PYTHON

KURS PROGRAMOWANIA NA PROSTYCH PRZYKŁADACH

Z TEJ KSIĄŻKI NAUCZYSZ SIĘ, JAK:

■ krok po kroku posługiwać się językiem Python – poznasz podstawy, najważniejsze polecenia, instrukcje i moduły ■ tworzyć programy i gry 2D oraz 3D ■ stosować programowanie obiektowe

Z TĄ KSIĄŻKĄ E-WYDANIE GRATIS

Poniżej znajduje się płyta z kodem bonusowym dającym dostęp do e-wydania tej książki w serwisie KŚ+ (www.ksplus.pl) oraz pliku ISO z cyfrową wersją płyty do pobrania.

NA PŁYCIE DVD

Płyta zawiera zestaw startowy niezbędnych narzędzi dla programistów rozpoczynających swoją przygodę z językiem Python: najlepsze środowiska programistyczne, edytory kodu źródłowego i pliki szkoleniowe do wskazówek przedstawionych w książce.

Jeżeli brakuje płyty, poinformuj sprzedawcę lub redakcję: redakcja@komputerswiat.pl

python

PŁYTA JEST DODATKIEM DO KSIĄŻKI

KOMPUTER ŚWIAT BIBLIOTECZKA

3|2019

- środowiska programistyczne
- edytory kodu źródłowego
- materiały szkoleniowe

Biblioteczka Komputer Świat

Kod bonusowy należy zarejestrować w KŚ+ (www.ksplus.pl)

KONRAD JAGACIAK

PYTHON

KURS PROGRAMOWANIA NA PROSTYCH PRZYKŁADACH

AUTOR: Konrad Jagaciak

REDAKTORZY PROWADZĄCY: Rafał Kamiński, Agnieszka Al-Jawahiri, Krzysztof Dziedzic

PRZYGOTOWANIE PŁYTY: Mariusz Michalski

PROJEKT OKŁADKI: Robert Dobrzyński

SKŁAD I ŁAMANIE: Mariusz Rybak

KOREKTA: Jolanta Rososińska

WYDAWCA: RINGIER AXEL SPRINGER POLSKA Sp. z o.o.
02-672 Warszawa, ul. Domaniewska 52
tel. 22 7786102
www.ringieraxelspringer.pl

ISBN: 978-83-8091-765-1

© Copyright by Ringier Axel Springer Polska Sp. z o.o.

Warszawa 2019

DYREKTOR WYDAWNICZY: Paweł Paczuski

BUSINESS PROJECT MANAGER: Paweł Bulwan

DRUK I OPRAWA: Drukarnia im. Adama Półtawskiego, Kielce

EGZEMPLARZE ARCHIWALNE:
www.literia.pl

prenumerata.axel@qg.com

E-WYDANIA: www.ksplus.pl

KONTAKT:
redakcja@komputerswiat.pl

INTERNET: komputerswiat.pl, ksplus.pl

Płyta DVD jest dodatkiem do książki

ringier
axel springer

Konrad Jagaciak
programista
prowadzący zajęcia
komputerowe
z młodzieżą

od autora

Python to jeden z moich ulubionych języków programowania. Osoby, które nie miały jeszcze styczności z innymi językami, powinny polubić Pythona za niezwykle prostą składnię, która zdecydowanie ułatwia naukę. To doskonały język do pierwszych programistycznych prób. W dodatku warto go wykorzystać jako język docelowy i rozwijać dalej dotyczące go umiejętności. Z tej książki dowiesz się, dlaczego dobrze jest postawić na Pythona, poznać go i dalej się go uczyć. Na kolejnych stronach przedstawiam Pythona na prostych przykładach – dzięki temu bez sztywnych definicji nauczysz się wykorzystywać niezbędne polecenia. Stworzysz zarówno programy tekstowe, jak i graficzne – w tym gry. Poznasz także bibliotekę Pythona, która pozwala na tworzenie gier z grafiką 3D. W książce znajdziesz również zadania (z rozwiązaniami i podpowiedziami), które możesz rozwiązać samodzielnie, aby dodatkowo poszerzyć i utrwalić swoją znajomość języka. Jeśli któreś z zadań okaże się dla ciebie zbyt skomplikowane, możesz zapoznać się z rozwiązaniami proponowanymi w rozdziale 6.

Zapraszam do lektury!

WSTĘP

Od autora . . . 3

1 DLACZEGO WARTO POZNAĆ PYTHONA?

Środowisko – skąd wziąć . . . 5
Instalacja środowiska . . . 6
Pip . . . 6
IDLE . . . 7

2 PODSTAWOWE POLECENIA

Konfiguracja edytora IDLE . . . 8
Praca z narzędziem IDLE . . . 9
Operacje wejścia/wyjścia . . . 10
Zmienne . . . 11
Pętle . . . 12
Instrukcja warunkowa . . . 14
Gra Loteria liczbowa . . . 16

3 PROGRAMY Z OKNEM GRAFICZNYM

Gra losowa . . . 23
Gra helikopterrr . . . 28
Biblioteka do tworzenia gier . . . 28
Okno gry modułu pygame . . . 29
Wstawianie napisów . . . 29
Pętla główna gry . . . 32
Wczytanie logo gry . . . 33
Tworzymy przeszkody . . . 35
Rysowanie przeszkód . . . 37
Tworzymy obiekty przeszkody . . . 39
Wywołujemy rysowanie przeszkód . . . 39
Wprawienie przeszkód w ruch . . . 41
Tworzymy helikopter . . . 43
Kolizje . . . 45
Przełączanie się pomiędzy menu a rozgrywką . . . 46
Ekran przegranej . . . 47
Punkty . . . 49
Zadanie 1 . . . 51

4 ZADANIA LOGICZNE

Podstawy modułu . . . 53
Gra Kółko i krzyżyk . . . 53
Tworzymy planszę . . . 53
Tworzymy krzyżyk . . . 55
Tworzymy kółko . . . 57
Testowanie . . . 58
Zadania 2, 3, 4 . . . 59

5 PROGRAMY Z GRAFIKĄ 3D

Poznajmy Panda3D . . . 61
Wyświetlenie bazowej sceny . . . 62
Wstawiamy aktora . . . 64
Wprawiamy pandę w ruch . . . 66

6 ROZWIĄZANIA ZADAŃ

Zadanie 1: Inwazja biedronek . . . 71
Zadanie 2: Graficzne kodowanie liczb . . . 82
Zadanie 3: Program sam wstawia znaki . . . 89
Zadanie 4: Blokada zajętego pola . . . 90

7 PODSUMOWANIE: CO WARTO WIEDZIEĆ O PYTHONIE

Program wczytujący trzy liczby i wyznaczający najmniejszą z nich . . . 93
Program wczytujący 10 liczb i określający, które z nich są parzyste, a które nie . . . 94
Najważniejsze polecenia z modułu math . . . 95
Jak zatrzymać na chwilę wykonywanie programu? . . . 96
Najważniejsze polecenia z modułu turtle . . . 97
Procedura do wytyczania żółwiem dowolnych wielokątów foremnych . . . 98
Program do przeliterowania dowolnego słowa . . . 98
Nie tylko IDLE . . . 100
Python nie tylko dla programistów . . . 101

DODATKI

Warto wiedzieć . . . 102
Programy na płycie . . . 104

1 Dlaczego warto poznać Pythona?

Dlaczego warto poznać Pythona? Bo Python jest wszędzie, a specjaliści od tego języka programowania mogą wszystko: tworzyć usługi takie jak Dropbox, serwisy jak Netflix, pracować z danymi i z internetem rzeczy. A zatem – zaczynajmy!

Jeśli czytasz tę książkę, to znaczy, że masz ochotę spróbować swoich sił w nauce programowania w Pythonie. I nie ma znaczenia, czy jest to twój pierwszy kontakt z programowaniem, czy znasz już inny język, a teraz chcesz nauczyć się kolejnego. Pierwszy krok w kierunku poznania Pythona już został wykonany.
Co tobą kierowało? Ty wiesz o tym najlepiej. Jeśli jednak wciąż masz wątpliwości, czy dalej podążać tą drogą, chciałbym przedstawić ci kilka argumentów przemawiających za tym, by zostać pythonowym programistą.
Pierwszy i chyba najważniejszy argument – to pewna i dobrze płatna praca. Pythonowi programiści są jednymi z najlepiej opłacanych specjalistów. Wysokie stawki nie biorą się z niczego – Python to język popularny, szeroko wykorzystywany, stąd popyt na osoby, które go znają. Zapotrzebowanie na specjalistów od Pythona rośnie z roku na rok.
A kto go wykorzystuje? Takie firmy, jak **Google**, **Dropbox**, **Spotify**, **Facebook**, **Instagram**, **Pinterest** czy **Netflix** używają Pythona w swoich aplikacjach. Dysk internetowy Dropbox w całości stworzono w Pythonie. Pozostali w swoich systemach wykorzystują moduły napisane w różnych językach, wśród których Python również się znajduje.

Jednak nie tylko giganci rynku IT wykorzystują ten język programowania. Jest on powszechnie stosowany w wielu mniejszych firmach, które zajmują się różnymi sektorami IT. Należą do nich: **cyberbezpieczeństwo**, **Web Development** (tworzenie stron internetowych) czy IoT (**Internet of Things** – internet rzeczy).

Python to język, który jest szeroko wykorzystywany także w nauce – szczególnie do celów statystycznych, ale też do obróbki danych na poziomie matematycznym (o tym, dlaczego Python idealnie nadaje się do celów matematycznych, dowiesz się z dalszej części tej książki).

Python jest językiem popularnym, a co za tym idzie – istnieje wiele materiałów pomocniczych, które pomogą go zrozumieć. Ta książka to dobry start w świecie Pythona. Nie sposób jednak zawrzeć w niej całą wiedzę, jaka może być potrzebna na dalszych etapach.

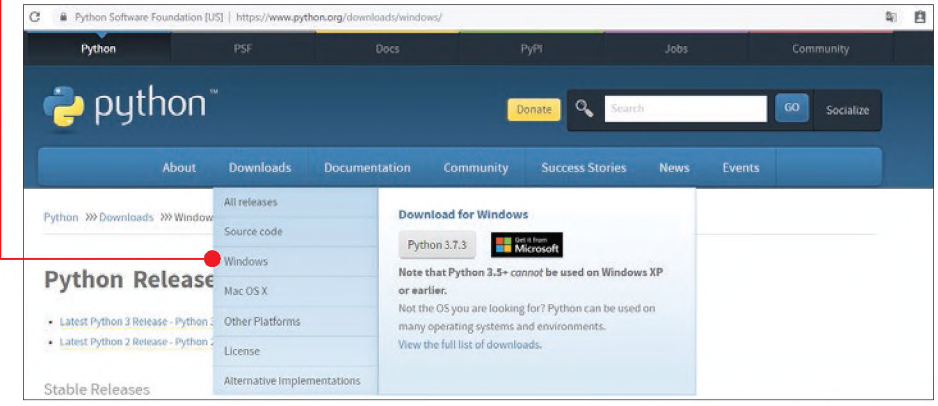
Kiedy dojdziesz już do wyższego poziomu wtajemniczenia, twoje pytania będą coraz bardziej szczegółowe, a odpowiedzi na nie – nieoczywiste. Na szczęście w przypadku popularnych języków, takich jak Python, prężnie działają internetowe społeczności chętne do pomocy.

Środowisko – skąd wziąć

Niezbędne do rozpoczęcia pracy z Pythonem narzędzia opisane w kolejnych rozdziałach można zainstalować z dołączonej do książki płyty (**Python 2** DVD-KOD: 013/014 32-/64-BIT, **Python 3** DVD-KOD: 015/016 32-/64-BIT) lub z KŚ+. Środowisko do nauki programowania w Pythonie można też pobrać z oficjalnej strony **python.org** – w wersji odpowiedniej dla danego systemu operacyjnego. Pobranie środowiska bezpośrednio ze strony internetowej python.org to dobra decyzja, ponieważ znajdziemy tam zawsze najnowszą jego wersję.

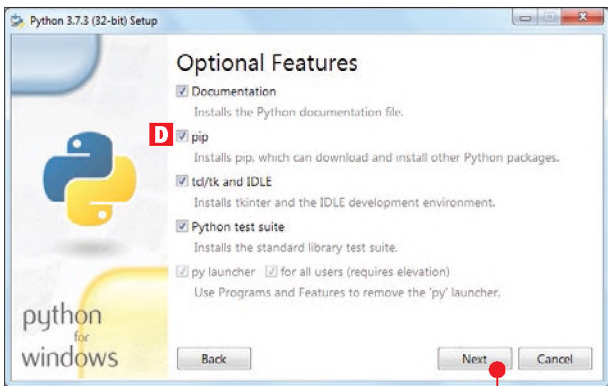
PYTHON 2 I PYTHON 3

Istnieją dwie ścieżki rozwojowe Pythona, które różnią się nieznacznie paroma szczegółami. To Python 2 i Python 3. Wersja z numerem 3 jest nowsza i to na jej rozwój kładziony jest większy nacisk. To ona wyznacza trendy na przyszłość i dlatego ta książka oparta jest właśnie na tej gałęzi Pythona. Wśród różnic pomiędzy wersjami należy wymienić brak niektórych bibliotek w wersji nowszej – nie zostały one jeszcze do niej przeniesione. Tendencja jest jednak taka, że z czasem na pewno do niej trafią.



Instalacja środowiska

Po uruchomieniu instalatora Pythona użytkownik ma do wyboru dwie opcje. Pierwsza (w przypadku gdy nie mamy jeszcze zainstalowanej żadnej wersji Pythona) to **Install Now** A lub (gdy mamy już zainstalowaną starszą wersję Pythona) **Upgrade Now**. Pozwala ona na standardową instalację środowiska.
Druga opcja – **Customize installation** B pozwala na zmianę ustawień instalacyjnych. To właśnie z tej opcji powinniśmy skorzystać. Zanim ją wybierzemy, spójrzmy jeszcze na opcje u dołu okna, które można zaznaczyć. Nas interesuje pozycja **Add Python 3.7 to PATH** C – powinna być zaznaczona. Bez aktywowania tej opcji niemożliwe może być skorzystanie z narzędzia **pip**, które znacznie ułatwia pracę z Pythonem. Po kliknięciu na **Customize installation** należy upewnić się, że opcja **pip** D i wszystkie pozostałe są zaznaczone, a następnie przejść dalej (**Next**).



Pip

Pip jest narzędziem do zarządzania pakietami Pythona. Pakiety to moduły – biblioteki funkcji. Do czego są przydatne? Język programowania sam w sobie nie zawiera wielu poleceń. Zbudowany jest z podstawowych instrukcji – jak pętle czy instrukcja warunkowa. Taki czysty język możemy wykorzystać na przykład do prostych obliczeń matematycznych. Stworzenie rozbudowanych projektów byłoby bardzo czasochłonne i skomplikowane. Tymczasem wiele mechanizmów, jakie mogą być nam potrzebne do tworzenia gier i aplikacji, zostało już wcześniej opracowanych. Chodzi tu zarówno o bardziej skomplikowane funkcje, jak choćby wczytanie i wyświetlenie pliku graficznego w oknie gry, jak i proste, jak obliczenie pierwiastka kwadratowego z danej liczby.
Python nie zawiera polecenia pozwalającego na obliczenie pierwiastka. Aby móc obliczyć pierwiastek kwadratowy ze 100, powinniśmy skorzystać z polecenia **sqrt(100)** – polecenie to znajduje się w module **math**. Aby skorzystać

z poleceń, jakie znajdują się w tym module, na początku skryptu należy napisać **from math import *** – co oznacza, że do skryptu mają zostać zaimportowane wszystkie polecenia z tego modułu. Gdybyśmy zamiast gwiazdki napisali **sqrt**, do skryptu z modułu **math** trafiłoby tylko to jedno polecenie. W analogiczny sposób w skrypcie dodajemy polecenia także z innych modułów. **Math** jest modułem bardzo popularnym – podstawowym. Takie moduły instalowane są wraz ze środowiskiem. Istnieje jednak bardzo dużo modułów, które nie są instalowane od razu i musimy instalować je sami. Właśnie do samodzielnej instalacji modułów służy pip. Na potrzeby realizacji zadań zawartych w tej książce również należy zainstalować pewne moduły. Jak to zrobić?

```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Wersja 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Windows\system32>pip install pygame
```
A B

1 Uruchamiamy Wiersz polecenia z uprawnieniami administratora.

2 Wpisujemy polecenie **pip install [nazwa modułu]** A – dla treningu możemy zainstalować moduł **pygame** B, z którego będziemy korzystać, realizując dalsze zadania. Pip najpierw pobierze moduł C (dlatego podczas tego procesu niezbędny jest dostęp do internetu), a następnie go zainstaluje.

```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Wersja 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Wszelkie prawa zastrzeżone

C:\Windows\system32>pip install pygame
Collecting pygame
  Downloading https://files.pythonhosted.org/packages/11/4f/2bbfdb52
c90e24bbb4b6970af66f0ebfe50bb32b67b51b39/pygame-1.9.6-cp36-cp36m-win
B)
    100% |████████████████████████████████| 4.0MB 80kB/s
Installing collected packages: pygame
Successfully installed pygame-1.9.6
You are using pip version 9.0.1, however version 19.1.1 is available
You should consider upgrading via the 'python -m pip install --upgra
```
C

IDLE

IDLE to nazwa prostego IDE, które jest instalowane razem z pakietem środowiska Python. Uruchamiamy je, wpisując **IDLE** w pole wyszukiwania Windows. To w tym programie będziemy pisać skrypty. Działa on na zasadzie konsoli – wpisane do niego polecenie wykonuje się od razu, a my możemy wpisywać kolejne linijki skryptu.

Aby na przykład sprawdzić, czy poprawnie udało się zainstalować moduł **pygame**, wpisujemy w IDLE polecenie **import pygame**. Pozwoli ono na korzystanie z poleceń modułu **pygame**. Jeśli moduł jest zainstalowany prawidłowo, o powodzeniu jesteśmy informowani komunikatem z nazwą modułu i numerem zainstalowanej wersji. A gdyby coś poszło nie tak, program poinformowałby nas o błędzie – **No module named „pygame”**.

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import pygame
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
```

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import pygame
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    import pygame
ModuleNotFoundError: No module named 'pygame'
>>>
```

2 Podstawowe polecenia

Głównym elementem, którym różnią się między sobą języki programowania z różnych rodzin, jest ich składnia. Jeśli chcesz poznać nowy język – musisz opanować jego składnię. Składnia Pythona jest intuicyjna – dzięki temu jest on doskonałym językiem do rozpoczęcia przygody z programowaniem. Poznajmy więc składnię Pythona

KONFIGURACJA EDYTORA IDLE

Wykonując wskazówki przedstawione w tym rozdziale, będziemy korzystać z edytora IDLE z pakietu Python. Jak go uruchomić, przeczytamy na stronie obok. Warto też zadbać o to, by praca w nim była dla nas wygodna, dostosowując odpowiednio rozmiar i rodzaj czcionki. To ważne w pracy programisty, który przez większość czasu patrzy na tekst. Aby otworzyć okno konfiguracji, należy z opcji **Options** w menu głównym wybrać pozycję **Configure IDLE**.

File Edit Format Run Options Window Help

```
def kolko(a,b
    pu()
    setx(a*bol
    sety(b*bok + bok/2 - 54/matl
```

Configure IDLE
Show Code Context
Zoom Height Alt+2

W nowym oknie możemy zmienić między innymi rodzaj i wielkość czcionki, a także inne ustawienia edytora (okno podzielone jest na zakładki, na których możemy edytować różne opcje).

Settings

Fonts/Tabs Highlights Keys General Extensions

Shell/Editor Font

Font Face:
Courier New
Courier New Baltic
Courier New CE
Courier New CE
Courier New CYR
Courier New Greek
Courier New TUR
Curlz MT
DejaVu Math TeX Gyre
DejaVu Sans
DejaVu Sans Condensed
DejaVu Sans Light
DejaVu Sans Mono
DejaVu Serif
DejaVu Serif Condensed

Size: 14 Bold

Indentation Width
Python Standard: 4 Spaces!
4
2 4 6 8 10 12 14 16

Font Sample (Editable)

<ASCII/Latin1>
AaBbCcDdEeFfGgHhIiJj
1234567890#:+=(){}[]

<IPA,Greek,Cyrillic>

<Hebrew, Arabic>

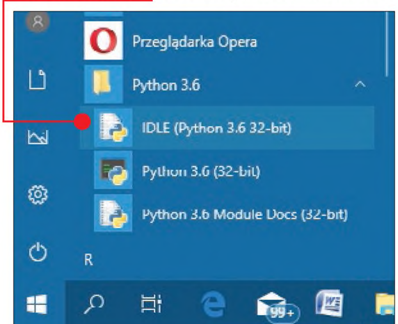
<Devanagari, Tamil>

Ok Apply Cancel Help

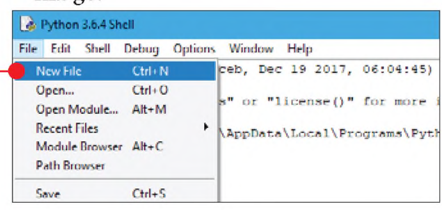
Praca z narzędziem IDLE

W tym rozdziale zapoznamy się z podstawowymi poleceniami Pythona, które nie wymagają do działania importu żadnych modułów.

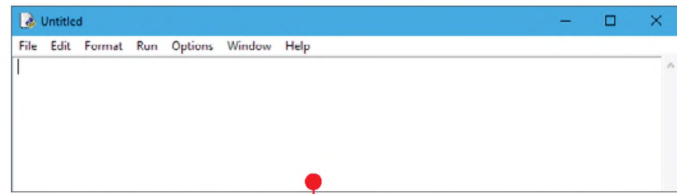
1 Uruchamiamy narzędzie **IDLE**. Domyślnie działaniem



przypomina ono konsolę. Każde polecenie, które wpiszemy, jest od razu wykonywane.
Rozwiązanie to jest dobre do prostych skryptów – na przykład matematycznych. Jeśli myślimy o tworzeniu większych projektów, takich jak gry, to wygodniejszym rozwiązaniem byłaby możliwość zapisania całego skryptu, a potem uruchomienia go.

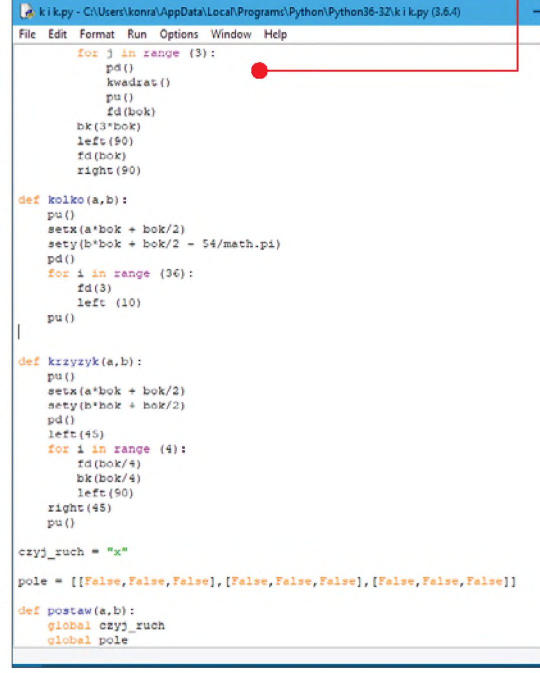


2 Na szczęście IDLE daje nam również taką możliwość. Narzędzie ma wbudowany edytor tekstowy – uruchomimy go, wybierając z menu głównego pozycję **File**, a następnie **New File**.



3 Okno z pozoru przypomina dobrze znany Notatnik. Nie ma zbyt wielu opcji – ale to ułatwia pracę początkującym programistom.

4 Tym, co ważne w tym edytorze programistycznym, jest kolorowanie składni. To rozwiązanie często stosowane w tego typu programach, ponieważ ułatwia tworzenie kodów. Już podczas pisania możemy zobaczyć, czy polecenia kolorują się odpowiednio, co świadczy o tym, że nie popełniliśmy błędu podczas pisania.

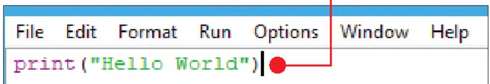


Operacje wejścia/wyjścia

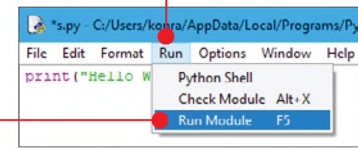
Operacje wejścia i wyjścia to podstawowy sposób komunikowania się programu tekstowego z użytkownikiem. Dlatego trzeba zapoznać się z dwoma poleceniami: **input()** – do odczytywania wartości podanych przez użytkownika, oraz **print()** – do wypisywania wartości, aby użytkownik mógł je przeczytać. Żeby lepiej zrozumieć działanie tych poleceń, warto zapoznać się z zagadnieniem zmiennych – patrz ramka **Zmienne**.

Polecenie print()

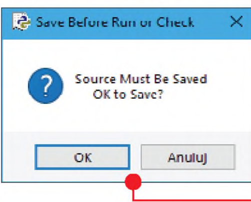
1 Używając polecenia **print()**, wpisujemy w nawiasie, co ma zobaczyć użytkownik. Jeśli chcemy wyświetlić napis Hello World, powinniśmy zastosować polecenie **print ("Hello World")**.



2 Aby wpisane przez nas polecenie (a później całe zestawy poleceń umieszczone w pliku) zostało wykonane, rozwijamy opcję **Run** z menu głównego.



3 Z rozwiniętej listy wybieramy **Run Module**. To pierwsza próba uruchomienia skryptu, musimy więc zapisać plik ze skryptem, korzystając z okna dialogowego. Przy kolejnych próbach uruchomienia skryptu też będzie wyświetlane okno z pytaniem o zapisanie zmian.



To dlatego, że wykonywany skrypt musi być zawsze zapisany w aktualnej wersji, aby można go było uruchomić.

4 Gdy nasz jednolinijkowy skrypt będzie już uruchomiony, w oknie IDLE powinno pokazać się sformułowanie ***Hello World***, które chcieliśmy wyświetlić.

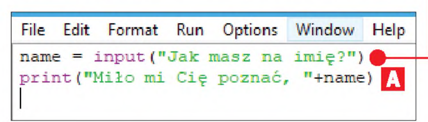
```
=== RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python36-32/s.py ===
Hello World
>>> |
```

Polecenie input()

Wiemy już, jak wypisywać dane. A jak spytać o coś użytkownika? Służy do tego polecenie **input()**, które łączy w sobie wypisywanie i odczytywanie informacji.

1 Aby pobrać dane od użytkownika, należy umieścić je w zmiennej. Linijkę kodu zaczynamy zatem od nazwy zmiennej, do której będziemy chcieli zapisać dane. Następnie po znaku równości używamy polecenia **input()**. W nawiasie należy wpisać informację dla użytkownika, która zostanie wyświetlona, gdy program będzie oczekiwał na podanie danych przez użytkownika.

2 Jeśli chcielibyśmy napisać program, który zapyta użytkownika, jak ma on na imię, możemy zrealizować to za pomocą polecenia **name = input("Jak masz na imię?")**.



3 Dalej nasz program mógłby odpowiedzieć użytkownikowi, wykorzystując jego imię w swojej wypowiedzi. Mógłby wypisać: ***Miło mi Cię poznać, ...***, w miejsce **...** wstawiając podane przez użytkownika imię. Aby wyświetlić samo sformułowanie ***Miło mi Cię poznać***, wystarczy umieścić

ZMIENNE

Zmienną w programowaniu najłatwiej porównać do pojemnika, w którym można przechowywać dane. Taki pojemnik ma swoją nazwę. Kiedy w skrypcie korzystamy z jego nazwy, odnosimy się do zawartości pojemnika. Wymyślmy pojemnik o nazwie **x**. Możemy w nim coś schować – na przykład liczbę 98. Gdybyśmy mieli zapisać to w Pythonie, użylibyśmy polecenia **x = 98**. Co oznacza, że stworzyliśmy zmienną o nazwie **x**, której nadaliśmy wartość **98**. Pojemniki mają to do siebie, że mogą mieć różne kształty i wymiary. A co za tym idzie, nie każdą zawartość da się umieścić w każdym pojemniku. Podobnie jest ze zmiennymi. Mamy naszą zmienną **x**, której wcześniej nadaliśmy wartość **98**, i inną zmienną – **y**, której wartość to jakieś słowo, na przykład **wyraz**. Spróbujmy zawartość pojemników **x** i **y** schować razem w pojemniku/zmiennej o nazwie **z**. Próba wykonania takiego skryptu powoduje wyświetlenie komunikatu o błędzie. W jego opisie możemy przeczytać: ***unsupported operand type(s) for +: 'int' and 'str'***. Oznacza to, że podjęliśmy próbę dodania do siebie dwóch zmiennych o różnych typach. Jedna z nich ma typ **int** (Integer), a druga **str** (String). Python należy do tych języków, w których nie musimy określać typu zmiennej, gdy ją tworzymy, inaczej niż choćby w C#, Javie czy C++. To wygodne rozwiązanie. Jednak to, że nie trzeba określać typu zmiennej, nie oznacza, że zmiennej nie jest nadawany typ. Python robi to sam, to znaczy nadaje zmiennej typ na podstawie umieszczonej w niej wartości. Naszym zmiennym **x** i **y** nadał odpowiednio typ **Integer** i **String**. Zmienne typu **Integer** służą do przechowywania liczb całkowitych, a zmienne typu **String** – do przechowywania słów (ciągów znaków). Gdybyśmy w zmiennej **y** umieścili liczbę – na przykład **12**, a następnie spróbowali w zmiennej **z** dodać do siebie **x** i **y** – program nie miałby z tym żadnego problemu, ponieważ **x** i **y** byłyby tego samego typu.

```
File  Edit  Format  Run
x = 98
y = "wyraz"
z = x + y
```

```
=== RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python36-32/s.py ===
Traceback (most recent call last):
  File "C:/Users/konra/AppData/Local/Programs/Python/Python36-32/s.py", line 3,
in <module>
    z = x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

je w poleceniu **print()**. Taki ciąg znaków jest przez program rozumiany tak samo jak zmienna typu **String**. Tak samo jest ze zmienną przechowującą imię użytkownika. Zmienne można do siebie dodawać. Jeśli w poleceniu **print()** dodamy do siebie dwa ciągi znaków, program wypisze je obok siebie. Zatem aby program zwrócił się do użytkownika podanym przez niego wcześniej imieniem, powinniśmy użyć polecenia **print("Miło mi Cię poznać, "+name)** **A**.

4 Uruchamiamy program poprzez opcję **Run Module**. Uzyskamy w ten sposób następujący efekt. Program wyświetla pytanie, czeka na podanie imienia i zatwierdzenie go klawiszem [enter], a następnie zwraca się do użytkownika, wykorzystując jego imię.

```
Type "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:\Users\konra\AppData\Local\Programs\Python\Python36-32\s.py ===
Jak masz na imię?Konrad
Miło mi Cię poznać, Konrad
>>>
```

Pętle

Ważnym elementem języków programowania są pętle. Służą one do wielokrotnego wykonywania zestawów instrukcji, które znajdują się wewnątrz nich. Jest kilka rodzajów pętli. Dwie najważniejsze to **for** i **while**. Czym się różnią? Pętla **for** wykonuje instrukcję określoną liczbę razy, natomiast pętla **while** wykonuje instrukcję, dopóki spełniony jest określony w niej warunek.

Pętla for

1 Zacznimy od tego, jak działa pętla **for**. Jeśli chcemy, aby program 5 razy napisał słowo ***stop*** – musimy stworzyć pętlę, która wykona się 5 razy. Piszemy więc: **for i in range(5):**, gdzie liczba podana w nawiasie mówi, ile razy ma się wykonać pętla.

```
File  Edit  Format  Run  Optio
for i in range(5):
```

2 Aby zapisać instrukcję, która ma się zapętlić – należy przejść do kolejnej linii kodu. IDLE automatycznie ustawi w tym momencie kursor z wcięciem. Po co? Aby wydzielić instrukcje, które mają wykonać się w pętli. Wszystkie linijki, które od tego momentu napiszemy z wcięciem – wykonają się 5 razy. Zapętlone będą wszystkie instrukcje aż do napotkania przez program linijki bez wcięcia.

```
*s.py - C:\Users\konra\AppData\Local\Programs\Pyth
File  Edit  Format  Run  Options  Window  Help
for i in range(5):
    |
```

3 Napiszmy zatem polecenie **print("stop")** zgodnie z naszym zamiarem pięciokrotnego wypisania tego słowa.

```
for i in range(5):
    print("stop")
```

4 Aby sprawdzić, czy wszystko zostało napisane poprawnie, uruchamiamy program. W konsoli powinno pojawić się pięciokrotnie słowo ***stop***.

```
Type "copyrigh
>>>
=== RESTART: C
stop
stop
stop
stop
stop
>>> |
```

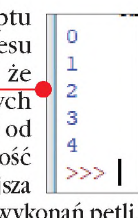
5 A jak sprawić, aby program poprzez pętlę napisał liczby od 1 do 100? Przyjrzyjmy się dokładnie stworzonej

```
for i in range(5):
    print("stop")
```

wcześniej pętli. Jest w niej litera **i**. Oznacza ona zmienną. Z każdym kolejnym wykonaniem się pętli wartość tej zmiennej zwiększa się o jeden. W instrukcjach wewnątrz pętli możemy korzystać z tej zmiennej. Zmieńmy zatem napisany wcześniej kod, tak by 5 razy wypisać wartość zmiennej **i**. W tym celu edytujemy polecenie **print**, usuwając słowo ***stop*** z nawiasu i zastępując je literą **i**.

```
for i in range(5):
    print(i)
```

6 Po uruchomieniu skryptu zobaczymy liczby z zakresu 0–4. Świadczy to o tym, że zmienna **i** podczas kolejnych wykonań pętli zaczyna rosnąć od zera, a osiągana przez nią wartość maksymalna jest o jeden mniejsza niż określona w kodzie liczba wykonań pętli.



7 Aby program napisał liczby od 1 do 100, pętla powinna wykonać się sto razy. Dlatego zmieniamy 5 na 100. Gdybyśmy

```
for i in range(100):
    print(i)
```

uruchomili skrypt w tej formie, program wypisałby liczby od 0 do 99. Aby zadanie było rozwiązane – każda z wypisanych liczb powinna być o 1 większa. A to znaczy, że zamiast wartości zmiennej **i** powinniśmy wy-

pisywać wartość **i+1** – i właśnie coś takiego powinno znaleźć się wewnątrz nawiasu w poleceniu **print**. Zadanie rozwiązane!

```
for i in range(100):
    print(i+1)
```

Uwaga! Zmienna, która rośnie w pętli **for**, nie musi nazywać się **i** – może mieć dowolną nazwę.

Pętla while

Przejdźmy teraz do drugiej ze wspomnianych pętli, czyli **while**. Wykonuje ona zestaw instrukcji, dopóki podany w niej warunek jest spełniany. Jak może wyglądać taki warunek? Na przykład: zmienna **a** jest mniejsza niż zmienna **b**.

1 Aby zapoznać się z działaniem pętli **while**, stwórzmy dwie zmienne: **a** o wartości 3 i **b** o wartości 10. Dalej napiszmy **while a<b:**, co będzie oznaczać, że napisane dalej instrukcje będą się wykonywać, dopóki wartość zmiennej **a** będzie mniejsza niż wartość zmiennej **b**. Wewnątrz pętli powinniśmy zmieniać wartość zmiennej **a** lub zmiennej **b** (albo obydwu) tak, aby warunek określony w pętli w pewnym momencie przestał być spełniany.

```
a=3
b=10
while a<b:
```

2 Instrukcje do wykonania wewnątrz pętli **while** zapisujemy z wcięciem, jak w wypadku pętli **for**. Napiszmy instrukcję **b = b+a** i **a = a*2+2**. Pierwsza z nich oznacza, że do obecnej wartości zmiennej **b** dodaje się wartość zmiennej **a**, a wynik staje się nową wartością zmiennej **b**. Druga z nich mówi, że nową wartością zmiennej **a** ma być jej stara wartość pomnożona przez 2 plus jeszcze 2. Dodajemy też polecenia **print(a)** i **print(b)**. Z każdym kolejnym przejściem pętli program wypisze wartości zmiennych **a** i **b**. Kiedy druga z wypisywanych liczb staje się mniejsza od pierwszej, program kończy wypisywać liczby.

```
a=3
b=10
while a<b:
    b = b+a
    a = a*2+2
    print(a)
    print(b)
```

```
8
13
18
21
38
39
78
77
>>>
```

3 Innym przykładem wykorzystania pętli **while** może być program pytający użytkownika o hasło aż do momentu, gdy ten poda je poprawnie. Aby napisać taki program, zaczynamy od stworzenia zmiennej **hasło**, której należy nadać wartość niebędącą poprawnym hasłem, na przykład: **hasło = "xxx"**.

4 Następnie tworzymy pętlę **while**, która będzie się wykonywać, dopóki hasło będzie inne niż poprawne. Wymyślmy dowolne hasło, które ma być poprawne, we wskazówce użyto słowa ***trzasło***. A to oznacza, że hasło ma być inne niż ***trzasło***. W Pythonie „inne niż" zapisujemy, używając wykrzyknika i znaku równości, tak więc: **while hasło-!="trzasło":**.

```
hasło = "xxx"
while hasło != "trzasło":
    hasło = input("Podaj hasło: ")
print("Udało Ci się")
```

5 Wewnątrz pętli należałoby pytać użytkownika o hasło poprzez polecenie **hasło = input("Podaj hasło: ")**.

6 Po pętli można użyć na przykład polecenia **print("Udało Ci się")**. Komunikat zostanie napisany dopiero wtedy, gdy pętla skończy się wykonywać, a więc gdy podane zostanie odpowiednie hasło.

```
Python 3.6.4 Shell
File  Edit  Shell  Debug  Options  Window
Python 3.6.4 (v3.6.4:d48eceb, De
 on win32
Type "copyright", "credits" or 
>>>
=== RESTART: C:\Users\konra\Appl
Podaj hasło: qwerty
Podaj hasło: haselko
Podaj hasło: masło
Podaj hasło: zgasło
Podaj hasło: trzasło
Udało Ci się
>>> |
```

Instrukcja warunkowa

Instrukcja warunkowa to kolejny podstawowy element języków programowania. Jest ona często nazywana ifem ze względu na słowo kluczowe **if**, które w niej występuje. W języku polskim oznacza ono jeżeli. Instrukcja warunkowa pozwala na wykonanie zestawu poleceń, jeżeli spełniony jest określony w niej warunek. Można zauważyć tu duże podobieństwo do pętli **while**, ale z taką różnicą, że instrukcje warunkowe nie są pętlami, więc umieszczone wewnątrz nich polecenia z zasady wykonują się jednokrotnie.
Instrukcję warunkową można rozbudowywać, dodając do niej kolejne sekcje **elif**, które sprawdzają inne warunki po sprawdzeniu warunków określonych wcześniej i mogą pozwolić na wykonanie innych zestawów poleceń. Innym sposobem rozbudowywania ifa jest dodanie sekcji **else** pozwalającej na określenie zestawu instrukcji, które wykonają się, gdy żaden z podanych warunków nie zostanie spełniony.
Jak to wygląda w praktyce? Przyjrzyjmy się temu na przykładzie prostego skryptu, którego zadaniem będzie zapytanie użytkownika o dwie liczby. A następnie program wyświetli informację, która z nich jest większa, a która mniejsza.

```
a = input("Podaj pierwszą liczbę: ")
b = input("Podaj drugą liczbę: ")
```

1 Zaczynamy od znanego nam już polecenia **input()** służącego do wczytywania danych od użytkownika. Używamy go dwukrotnie, wpisując podane przez użytkownika dane do dwóch zmiennych – **a** i **b**.

2 Dalej możemy już przejść do budowania instrukcji warunkowej. Robimy to, używając słowa kluczowego **if**, po którym piszemy warunek i stawiamy znak dwukropka. Naszym warunkiem będzie to, że wartość zmiennej **a** ma być większa niż wartość zmiennej **b**. Chcemy wykonać coś, tylko gdy podane stwierdzenie będzie prawdą. Piszemy zatem **if a > b:**.

```
a = input("Podaj pierwszą liczbę: ")
b = input("Podaj drugą liczbę: ")
if a > b:
```

3 Gdy podany przez nas warunek będzie spełniony, powinna zostać wypisana informacja o tym, że pierwsza z liczb jest większa. Osiągniemy to poprzez polecenie **print()**.

```
a = input("Podaj pierwszą liczbę: ")
b = input("Podaj drugą liczbę: ")
if a > b:
    print("Pierwsza z podanych liczb była większa")
```

Jak widzimy, kursor podczas pisania powinien ustawić się z wcięciem. Zasada wcięć obowiązuje w przypadku instrukcji warunkowej tak samo jak w przypadku pętli. To cecha charakterystyczna składni języka Python – występuje we wszystkich konstrukcjach mających „wnętrze". Musimy przyzwyczaić się do tego, aby pisząc w Pythonie, pilnować wcięć w kodzie.

4 Program powinien zachować się inaczej, gdy większą wartość będzie mieć zmienna **b**. Dlatego naszą instrukcję rozbudujemy o sekcję **elif**, w której sprawdzimy, czy zmienna **b** jest większa. Zrobimy to, dodając linijkę **elif b > a:**.

```
a = input("Podaj pierwszą liczbę: ")
b = input("Podaj drugą liczbę: ")
if a > b:
    print("Pierwsza z podanych liczb była większa")
elif b > a:
```

5 Jeżeli ten warunek będzie spełniony, należałoby wyświetlić informację – przez polecenie **print()** – o tym, że druga z poda-

nych przez użytkownika liczb była większa.

```
a = input("Podaj pierwszą liczbę: ")
b = input("Podaj drugą liczbę: ")
if a > b:
    print("Pierwsza z podanych liczb była większa")
elif b > a:
    print("Druga z podanych liczb była większa")
```

6 To jeszcze nie koniec naszej instrukcji warunkowej. Istnieje jeszcze jedna możliwość, którą trzeba uwzględnić. Użytkownik może przecież wpisać dwukrotnie taką samą liczbę. Dlatego do instrukcji dopiszemy sekcję **else:**.

```
a = input("Podaj pierwszą liczbę: ")
b = input("Podaj drugą liczbę: ")
if a > b:
    print("Pierwsza z podanych liczb była większa")
elif b > a:
    print("Druga z podanych liczb była większa")
else:
```

```
a = input("Podaj pierwszą liczbę: ")
b = input("Podaj drugą liczbę: ")
if a > b:
    print("Pierwsza z podanych liczb była większa")
elif b > a:
    print("Druga z podanych liczb była większa")
else:
    print("Podane liczby były sobie równe")
```

W niej możemy użyć polecenia **print()**, które wykona się w każdej innej sytuacji, gdy żaden z podanych wyżej dwóch warunków nie zostanie spełniony. Umieścimy tu informację, która ma zostać wyświetlona użytkownikowi.

7 Możemy teraz przetestować ten prosty program. Uruchamiamy go i podajemy dwie liczby **A**.

8 Aby utrwalić zdobytą do tej pory wiedzę, spróbujmy jeszcze tak zmienić kod programu, aby pytanie o dwie liczby i wyświetlanie informacji o tym, która z nich była większa, wykonało się 3 razy.

Uwaga! Niezależnie od realizacji tego zadania możemy zauważyć, że programowi zdarza się popełnić błąd **B**, mimo że logicznie wszystko jest poprawnie napisane. Wynika to z dużej awaryjności polecenia **input()**, które może sobie nie poradzić z poprawnym zaklasyfikowaniem podanej przez użytkownika liczby. Możemy programowi nieco pomóc, stosując **rzutowanie na typ integer**. W tym celu polecenia **input** umieszczamy w nawiasie i piszemy przed nimi nazwę typu, na który będziemy rzutować dane, czyli **int**.

```
a = int(input("Podaj pierwszą liczbę: "))
b = int(input("Podaj drugą liczbę: "))
```

Po wprowadzeniu zmian przetestujmy program raz jeszcze – wynik powinien być już prawidłowy **C**.

PODPOWIEDŹ

Zadanie można rozwiązać z wykorzystaniem pętli **for**.

```
 RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python36-32/liczby.py
Podaj pierwszą liczbę: 64
Podaj drugą liczbę: 128
Druga z podanych liczb była większa
>>> 
```
A

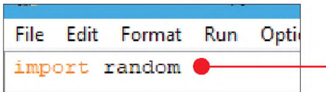
```
>>> ================================ RESTART ================================
>>> 
Podaj pierwszą liczbę: 34
Podaj drugą liczbę: 53
Pierwsza z podanych liczb była większa
>>> 
```
B

```
 RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python36-32/liczby.py
Podaj pierwszą liczbę: 34
Podaj drugą liczbę: 54
Druga z podanych liczb była większa
>>> 
```
C

GRA LOTERIA LICZBOWA

Aby lepiej utrwalić informacje poznane w tym rozdziale, a także nieco je rozszerzyć, zróbmy teraz prostą grę.
Zasada jej działania będzie wzorowana na popularnej loterii liczbowej, która polega na podaniu przez gracza zestawu sześciu liczb, tak by trafić jak najwięcej z sześciu wylosowanych.

1 Będziemy korzystać z losowania liczb. Potrzebne do tego polecenia nie znajdują się standardowo w języku Python. Żeby z nich skorzystać, musimy zaimportować moduł **random**. Od tego zaczynamy skrypt.



2 Nasza gra będzie pozwalała na wielokrotną rozgrywkę. Możemy stworzyć zmienną **gramy** i nadać jej wartość **tak**,

```
File Edit Format Run Option
import random

gramy = "tak"
```

a samą rozgrywkę umieścić wewnątrz pętli **while**, która będzie się wykonywać, dopóki zmienna **gramy** będzie miała wartość **tak**.

```
import random

gramy = "tak"

while gramy == "tak":
```

3 Skupmy się jeszcze na tym, co można zrobić przed pętlą **while**. Zarówno podane przez użytkownika, jak i wylosowane przez program liczby musimy jakoś przechować. Wiemy już, że dane możemy przechowywać w zmiennych. Tworzenie teraz dwunastu nowych zmiennych – bo tyle liczb musimy przechować – byłoby rozwiązaniem dość kiepskim. Są lepsze sposoby. Jednym z nich są **listy**. Lista to struktura danych, która pozwala na przechowywanie wielu elementów pod tą samą nazwą. Potrzebujemy dwóch takich dużych pojemników na dane: jednego przeznaczonego na liczby podane przez użytkownika i drugiego – na liczby wylosowane przez program. Jeszcze przed pętlą trzeba zatem zdefiniować dwie listy. Robimy to, przypisując do nazwy listy pusty kwadratowy nawias. Tworzymy tak listy **podane** i **wylosowane**.

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
```

4 Ponieważ gra polega na prawidłowym wytypowaniu sześciu liczb, zarówno losowanie, jak i pytanie gracza o kolejną liczbę powinno wykonać się 6 razy. Jakiej instrukcji należy użyć, aby coś wykonało się 6 razy? Oczywiście, potrzebna jest pętla **for**.

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
```

5 We wnętrzu tej pętli powinniśmy pytać gracza o kolejne liczby, a ich wartości dodawać do listy **podane**. W tym celu skorzystamy z polecenia **podane.append()** – to funkcja pozwalająca na dodawanie do listy elementów umieszczonych w nawiasie. Jeśli w nawiasie umieścimy polecenie **input()**, do listy trafią wartości podane przez gracza. Możemy jeszcze te wartości rzutować na typ **integer**. Używamy zatem polecenia **podane.append(int(input("Podaj liczbę numer**

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
```

"+str(i+1)+": "))). Jeśli odpowiednio sformułujemy to, co ma wyświetlić polecenie **input()** i wykorzystamy zmienną **i**, której wartość rośnie z kolejnymi przejściami pętli, program będzie mógł wypisywać, którą z kolei liczbę gracz ma podać.

6 W tej samej pętli możemy dodawać liczby do listy liczb wylosowanych. W tym celu użyjemy polecenia **wylosowane.append()**, jednak w tym wypadku w nawiasie musimy użyć polecenia do losowania liczb. Znajduje się ono w module **random**, który importowaliśmy

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
```

wcześniej. Potrzebne nam polecenie nazywa się **randint**, a w nawiasie podajemy zakres, z którego ma pochodzić wylosowana liczba. Powiedzmy, że będzie to liczba od 1 do 50.

UWAGA

Liczby losowe, o których mówimy, są tak naprawdę liczbami pseudolosowymi. Komputer nie jest w stanie przeprowadzić losowania. Liczby te są w rzeczywistości generowane przez program na podstawie pewnych warunków, które mają charakter możliwie zbliżony do losowości.

7 Kolejnym etapem rozwiązywania zadania powinno być sprawdzanie, ile liczb udało się graczowi trafić. Liczbę tę będziemy przechowywać w zmiennej **trafione**. Tworzymy ją i na początku dajemy jej wartość 0. Potem będziemy tę wartość zwiększać, jeśli któraś z liczb zostanie trafiona przez gracza.

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
```

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
    for z in podane:
```

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
    for z in podane:
        for j in wylosowane:
```

8 Jak będzie wyglądać logika tego sprawdzania? Powinniśmy brać liczbę z podanych przez gracza i porównywać ją z kolejnymi liczbami z listy wylosowanych. Jak to zrobić? Użyjemy do tego pętli **for** w nieco innej formie. Zapisanie **for z in podane:** sprawia, że pętla wykona się tyle razy, ile elementów jest na liście **podane**. Zmienna **z**, której użyliśmy w pętli, nie będzie natomiast przybierała kolejnych rosnących wartości od zera. Będzie ona przybierała wartości kolejnych elementów z listy **podane**.

9 Następnie każdą liczbę z podanych przez gracza powinniśmy porównywać z każdą kolejną wylosowaną. Dlatego w napisanej przez nas pętli tworzymy drugą pętlę **for** „przechodzącą" przez elementy listy **wylosowane**. Robimy to za pomocą polecenia **for j in wylosowane:**.

OPERATOR RÓWNOŚCI

Dla sprawdzenia, czy dwie wartości są sobie równe, a takie sprawdzenia często pojawiają się w pętli **while** czy w ifach, musimy zastosować operator porównania zbudowany z dwóch znaków równości. Jeden znak równości oznacza przypisanie wartości. Napisanie: **if z=j:** - oznaczałoby użycie operatora przypisania, czyli próbę nadania zmiennej **z** wartości zmiennej **j**, co skutkowałoby błędem programu (spowodwanym błędem składni).

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
    for z in podane:
        for j in wylosowane:
            if z == j:
```

10 Kolejnym krokiem powinno być teraz porównanie wartości, jakie zostaną umieszczona w **z** oraz w **j**. W tym celu w naszym przykładzie użyjemy ifa, czyli **if**.

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
    for z in podane:
        for j in wylosowane:
            if z == j:
                trafione = trafione + 1
```

11 Jeśli liczby te okażą się takie same, zwiększamy wartość zmiennej **trafione** o 1.

12 Dalej wychodzimy z pętli **for** i piszemy, co ma wykonywać się w pętli **while**. Zauważmy, że kolejne linijki mają już tylko jedno wcięcie. Poprzez **print** należałoby wpisać, ile było trafionych liczb.

13 Można również, korzystając z pętli **for**, wypisać liczby, które wylosował program.

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
    for z in podane:
        for j in wylosowane:
            if z == j:
                trafione = trafione + 1

    print("Twój wynik to: "+str(trafione))
```

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
    for z in podane:
        for j in wylosowane:
            if z == j:
                trafione = trafione + 1

    print("Twój wynik to: "+str(trafione))
    print("Wylosowane liczby:")
    for i in wylosowane:
        print(i)
```

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
    for z in podane:
        for j in wylosowane:
            if z == j:
                trafione = trafione + 1

    print("Twój wynik to: "+str(trafione))
    print("Wylosowane liczby:")
    for i in wylosowane:
        print(i)
    podane.clear()
    wylosowane.clear()
```

14 Aby możliwe było przeprowadzenie kolejnej rozgrywki, trzeba wyrzucić wszystkie elementy z naszych dwóch list. Robimy to, pisząc nazwę listy, a następnie stawiając kropkę i wywołując dla listy polecenie **clear()**.

15 Na sam koniec możemy nadać nową wartość zmiennej **gramy**, pytając o nią gracza. Jeśli wpisze on ***tak***, program uruchomi kolejną kolejkę rozgrywki. A jeżeli wpisze co innego, gra się zakończy.

```
import random

gramy = "tak"

podane = []
wylosowane = []

while gramy == "tak":
    for i in range(6):
        podane.append(int(input("Podaj liczbę numer "+str(i+1)+": ")))
        wylosowane.append(random.randint(1,50))
    trafione = 0
    for z in podane:
        for j in wylosowane:
            if z == j:
                trafione = trafione + 1

    print("Twój wynik to: "+str(trafione))
    print("Wylosowane liczby:")
    for i in wylosowane:
        print(i)
    podane.clear()
    wylosowane.clear()
    gramy = input("Czy chcesz zagrać jeszcze raz? (tak/nie) ")
```

```
*Python 3.6.4 Shell*
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python36-32/LOTERIALIZCBO
WA.py
Podaj liczbę numer 1: 4
Podaj liczbę numer 2: 32
Podaj liczbę numer 3: 24
Podaj liczbę numer 4: 13
Podaj liczbę numer 5: 43
Podaj liczbę numer 6: 7
Twój wynik to: 0
Wylosowane liczby:
34
12
22
37
26
1
Czy chcesz zagrać jeszcze raz? (tak/nie) tak
Podaj liczbę numer 1: |
```

16 Przetestujmy grę. Sprawdźmy, czy da się przeprowadzić rozgrywkę. Gra nie jest idealnym odzwierciedleniem loterii, w której z bębna wypadają kulki z liczbami. W naszym programie zarówno gracz, jak i program mogą powtarzać liczby. Może to powodować drobne problemy i błędy. Spróbujmy samodzielnie tak zmienić skrypt, aby wylosowanie dwukrotnie tej samej liczby było niemożliwe.

PODPOWIEDŹ

Zadanie można rozwiązać z wykorzystaniem pętli **while**, która będzie sprawdzać każdą kolejną wylosowaną liczbę i powtarzać losowanie, jeżeli liczba ta znajdowała się już na liście liczb wylosowanych wcześniej.

LISTY, LISTY LIST I TABLICE

Listy to bardzo przydatne struktury do przechowywania danych. Są jednak sytuacje, że listy to zbyt mało. Dane w rzeczywistości często zapisujemy w formie tabelarycznej. Języki programowania są na to przygotowane. W Pythonie możemy korzystać z tak zwanych **list list**. Oznacza to, że elementem jednej listy mogą być inne listy. Takie struktury nazywa się też macierzami i tablicami dwuwymiarowymi (lub wielowymiarowymi, bo elementem **listy list** może być lista – i tak dalej). W niektórych językach programowania listy i tablice to odrębne struktury, jednak w Pythonie nie rozróżnia się ich, nazw tych można używać zamiennie. Oto przykładowa deklaracja listy list wypełnionej zerami:
tabela = [[0,0],[0,0],[0,0]]

3 Programy z oknem graficznym

W poprzednim rozdziale zostały omówione podstawy języka Python. Za przykład posłużyły nam programy tworzone w trybie tekstowym. Teraz pora na stworzenie własnego programu z oknem graficznym. Będziemy musieli skorzystać z importu modułów

Python ma moduł, który pozwala na tworzenie interfejsu graficznego. To biblioteka **tkinter**. Za jej pomocą stworzony został choćby program IDLE, który standardowo dołączony jest do Pythona i z którego korzystamy we wskazówkach przedstawionych w książce.

Jak korzystać z tego modułu, nauczymy się na kolejnych stronach, tworząc według wskazówek własne gry.

GRA LOSOWA

Stworzymy teraz prostą grę losową. W oknie programu znajdzie się kilka przycisków, wszystkie będą wyglądały tak samo. Zadaniem gracza będzie trafienie jednego prawidłowego przycisku. Kliknięcie na każdy z pozostałych to będzie „pudło".

1 Pisanie rozpoczynamy od polecenia **from tkinter import ***, za pomocą którego importujemy do programu wszystkie polecenia z modułu **tkinter** do naszego programu.

```
File  Edit  Format  Run  Options  Window  Help
from tkinter import *
```

2 Drugim potrzebnym nam poleceniem będzie utworzenie obiektu, który będzie oknem graficznym. Takie okno musi mieć nazwę, do której przypiszemy polecenie tworzące okno graficzne. Jeśli chcemy, aby okno

```
from tkinter import *

t = Tk()
```

nazywało się **t**, piszemy **t = Tk()**, gdzie **Tk()** to nazwa specjalnej funkcji tworzącej obiekt okna graficznego. Jeśli teraz zapiszemy i uruchomimy ten dwulinijkowy skrypt, zobaczymy, że oprócz informacji napisanych w konsoli program ma okno graficzne.

```
from tkinter import *

t = Tk()
t.title("Wybierz przycisk")
```

3 Następnie możemy dokonać prostej konfiguracji tego okna. Okno składa się z sekcji graficznej i paska tytułowego. Zacznijmy od wyświetlenia odpowiedniego tekstu na pasku tytułowym. Skorzystamy z polecenia **t.title("Wybierz przycisk")**, gdzie **t** jest nazwą okna graficznego, a **"Wybierz przycisk"** to tekst, który chcemy umieścić na pasku.

4 Warto też zmienić rozmiar okna. Służy do tego polecenie **geometry()**. Korzystamy z niego, pisząc: **t.geometry("300x350")**, gdzie **t** to nazwa okna graficznego, a **"300x350"** to jego wymiary. Pierwsza liczba, przed x, to szerokość okna (w pikselach), a druga liczba, po x, to wysokość okna.

```
from tkinter import *

t = Tk()
t.title("Wybierz przycisk")
t.geometry("300x350")
```

5 Kolejnym krokiem będzie stworzenie przycisków, które umieścimy w oknie graficznym. W tym celu opracujemy nową procedurę, która będzie realizować to zadanie. Co przemawia za takim rozwiązaniem? To, że ułatwi ono późniejszy restart gry. Kiedy będziemy chcieli zagrać jeszcze raz, bez konieczności ponownego uruchamiania gry, program będzie mógł ponownie dodawać przyciski. Piszemy zatem procedurę **wstaw_przyciski()**.

```
def wstaw_przyciski():
```

6 Wewnątrz procedury stworzymy zmienną **ile_przyciskow**. W niej będzie

```
def wstaw_przyciski():
    ile_przyciskow = 8
```

przechowywana informacja o tym, jak dużo przycisków chcemy mieć w oknie gry. Zastosujemy zmienną w każdym miejscu kodu, w którym niezbędne będzie podanie liczby przycisków, oraz użyjemy odniesienia do niej. Dzięki temu bardzo łatwo będziemy mogli zmienić finalną liczbę przycisków w grze, ponieważ wystarczy w tym celu tylko nadać odpowiednią wartość zmiennej. Unikniemy więc wprowadzania zmian w kodzie w każdym miejscu, w którym odnosimy się do liczby przycisków.

7 Aby móc swobodnie zmieniać liczbę przycisków bez wprowadzania wielu zmian w kodzie, musimy mieć odpowiednią strukturę do przechowywania tych przycisków. W tym wypadku najlepiej sprawdzi się lista. Należy zwrócić uwagę na to, że to, co piszemy, znajduje się wewnątrz procedury. Gdybyśmy więc teraz zadeklarowali listę przycisków, byłaby ona dostępna tylko wewnątrz tej procedury. A to znaczy, że gdyby była inna procedura, która też powinna oddziaływać na przyciski, nie miałaby ona dostępu do tej listy. Jest na to sposób. Tworząc listę o nazwie **przyciski**, użyjmy wcześniej słowa **global**. W ten sposób stworzymy coś, co jest globalne i dostępne z poziomu innych procedur.

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
```

8 W kolejnej linii kodu powinniśmy wskazać, że nasz globalny obiekt o nazwie **przyciski** ma być listą. Robimy to, przypisując do niego pusty, kwadratowy nawias.

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
```

9 Tworzeniem przycisków zajmiemy się w pętli **for**, która wykona się tyle razy, ile wynosi wartość zmiennej **ile_przyciskow**.

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    for i in range (ile_przyciskow):
```

Jak jednak sprawić, by jeden z przycisków oznaczał zwycięstwo, a pozostałe nie? To musimy wskazać już na etapie ich tworzenia. Przycisk oznaczający zwycięstwo powinien być w każdej rozgrywce inny i nieprzewidywalny, czyli losowy. Logicznym rozwiązaniem wydaje się w takim razie stworzenie zmiennej, w której będziemy przechowywać pewną liczbę. Wartość tej liczby porównywać będziemy z wartością zmiennej **i**, która rośnie z kolejnymi przejściami pętli **for**.

10 Zmienną, w której przechowamy liczbę mówiącą o tym, który z przycisków prowadzi do wygranej, musimy utworzyć przed pętlą **for**. Zmienna, która rośnie od zera z kolejnymi przejściami pętli, maksymalnie osiągnie wartość o jeden mniejszą niż liczba wykonań pętli. Co za tym idzie, przyciski będą miały numery odpowiadające kolejnym wartościom tej zmiennej. Zatem losując numer dla przycisku prowadzącego do wygranej, musimy skorzystać z zakresu od zera do liczby o jeden mniejszej od zmiennej **ile_przyciskow**. Korzystamy więc z polecenia **dobry = random.randint(0,ile_przyciskow-1)**. Tworzymy w ten sposób nową zmienną o nazwie **dobry** i nadajemy jej losową wartość.

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    dobry = random.randint(0,ile_przyciskow-1)
    for i in range (ile_przyciskow):
```

11 Zostawienie programu w takim stanie skutkowałoby jednak błędem, ponieważ polecenie **random.randint()** jest dostępne tylko wtedy, kiedy dokonamy importu modułu **random** – dopiszmy to więc u góry skryptu.

```
File  Edit  Format  Run  Options  Window  Help
from tkinter import *
import random
t = Tk()
t.title("Wybierz przycisk")
t.geometry("300x350")
```

12 Wróćmy jeszcze do pętli **for** i tworzenia przycisków. To, jaki przycisk utworzymy, ma zależeć od wartości zmiennej **dobry**. Jeśli jest ona równa zmiennej **i**, tworzymy przycisk nieco inaczej niż w każdym innym wypadku. Porównanie wartości dwóch zmiennych zapisujemy ifem – instrukcją warunkową **if**.

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    dobry = random.randint(0,ile_przyciskow-1)
    for i in range (ile_przyciskow):
        if i == dobry:
```

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    dobry = random.randint(0,ile_przyciskow-1)
    for i in range (ile_przyciskow):
        if i == dobry:
            przyciski.append(Button(t,text = "kliknij mnie",command=trafiony))
```

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    dobry = random.randint(0,ile_przyciskow-1)
    for i in range (ile_przyciskow):
        if i == dobry:
            przyciski.append(Button(t,text = "kliknij mnie",command=trafiony))
        else:
```

13 Przejdźmy do napisania instrukcji, która wykona się, gdy warunek zostanie spełniony. Do stworzenia przycisku służy polecenie **Button(t, text = "kliknij mnie",command=trafiony)**. Litera **t** oznacza w nim nazwę okna gry. Wartość przypisana jako **text** to napis, jaki ma pojawić się na przycisku. Natomiast to, co przypisujemy do **command**, to nazwa procedury, która wykona się po kliknięciu na ten przycisk. Zgodnie z tym poleceniem niezbędne będzie stworzenie procedury o tej nazwie.

14 Aby przycisk stworzony poprzez opisaną linijkę trafił do naszej listy przycisków, powinniśmy dodać go tam przez polecenie **append**. Pełna linijka kodu tworząca przycisk i jednocześnie dodająca go do listy powinna wyglądać więc tak: **przyciski.append(Button(t, text = "kliknij mnie",command=trafiony))**.

15 Aby móc stworzyć przycisk inaczej, to znaczy gdy zmienna **i** ma inną wartość niż wcześniej wylosowana, do **if** należy dodać sekcję **else**.

16 Wewnątrz sekcji **else** ponownie tworzymy przycisk i dodajemy go do listy z przyciskami. Różnica polega na tym, że kliknięcie na ten przycisk ma skutkować wykonaniem innej procedury. Zatem przypisujemy do **command** inną nazwę. W ten sposób kończymy pętlę **for** odpowiedzialną za tworzenie nowych przycisków.

17 Nie jest to jednak jeszcze koniec samej procedury. **Tkinter** wymaga, aby utworzone przyciski odpowiednio umieścić w oknie. Wykorzystamy do tego polecenie **pack()**, które należy wywołać z osobna dla każdego przycisku. Dlatego stworzymy drugą pętlę **for**, która tym razem nie będzie

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    dobry = random.randint(0,ile_przyciskow-1)
    for i in range (ile_przyciskow):
        if i == dobry:
            przyciski.append(Button(t,text = "kliknij mnie",command=trafiony))
        else:
            przyciski.append(Button(t,text = "kliknij mnie",command=pudlo))
```

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    dobry = random.randint(0,ile_przyciskow-1)
    for i in range (ile_przyciskow):
        if i == dobry:
            przyciski.append(Button(t,text = "kliknij mnie",command=trafiony))
        else:
            przyciski.append(Button(t,text = "kliknij mnie",command=pudlo))
    for i in przyciski:
```
A

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    dobry = random.randint(0,ile_przyciskow-1)
    for i in range (ile_przyciskow):
        if i == dobry:
            przyciski.append(Button(t,text = "kliknij mnie",command=trafiony))
        else:
            przyciski.append(Button(t,text = "kliknij mnie",command=pudlo))
    for i in przyciski:
        i.pack(fill=BOTH,expand=YES)
```

wykonywać się w określonym zakresie liczbowym, ale dla wszystkich elementów listy **przyciski** powinna ona wyglądać następująco: **for i in przyciski:** **A** (patrz poprzednia strona).

18 Przyciski wyświetlą się pionowo, jeden pod drugim. Zmienna **i** w nowej pętli **for** będzie odnosiła się do kolejnych przycisków na liście. Zatem polecenie **pack()** należy uruchamiać dla obiektu **i**. W nawiasie tego polecenia wpisujemy też **fill=BOTH**, co oznacza, że przyciski mają swoją szerokością wypełnić całe okno gry, a także **expand=YES**, co oznacza, że przyciski po równo podzielą się całą wysokością okna gry, wypełniając ją.

19 Aby sprawdzić, czy wszystko zostało poprawnie napisane, po definicji procedury należałoby ją wywołać **B**. Jednak taka próba będzie w tym momencie skutkowała błędem, a to dlatego, że program nie zna jeszcze dwóch procedur, które przypisaliśmy do **command** podczas tworzenia przycisków.

20 Przed ich napisaniem trzeba zastanowić się, co mają robić. Dobrym rozwiązaniem byłoby pozbycie się z okna gry przycisków i wyświetlenie informacji o tym, czy trafiliśmy w dobry, czy zły przycisk. Zaczynamy od słowa **def** i podania nazwy tworzonej procedury.

```
def trafiony():
```

```
def wstaw_przyciski():
    ile_przyciskow = 8
    global przyciski
    przyciski = []
    dobry = random.randint(0,ile_przyciskow-1)
    for i in range (ile_przyciskow):
        if i == dobry:
            przyciski.append(Button(t,text = "kliknij mnie",command=trafiony))
        else:
            przyciski.append(Button(t,text = "kliknij mnie",command=pudlo))
    for i in przyciski:
        i.pack(fill=BOTH,expand=YES)

wstaw_przyciski() B
```

```
 RESTART: C:/Users/clientadmin/AppData/Local/Programs/Python/Python37-32/3przyci
ski.py
Traceback (most recent call last):
  File "C:/Users/clientadmin/AppData/Local/Programs/Python/Python37-32/3przycisk
i.py", line 45, in <module>
    wstaw_przyciski()
  File "C:/Users/clientadmin/AppData/Local/Programs/Python/Python37-32/3przycisk
i.py", line 19, in wstaw_przyciski
    przyciski.append(Button(t,text = "kliknij mnie",command=pudlo))
NameError: name 'pudlo' is not defined
>>>
```

21 Wewnątrz procedury tworzymy pętlę **for**, która będzie wykonywać się dla każdego z przycisków umieszczonych na liście.

```
def trafiony():
    for i in przyciski:
```

22 A wewnątrz tej pętli za pomocą polecenia **destroy** uruchamianego dla każdego elementu listy z przyciskami pozbywamy się po kolei każdego z przycisków.

```
def trafiony():
    for i in przyciski:
        i.destroy()
```

23 Musimy jednak mieć jakiś inny element okna, który wciąż będzie widoczny dla gracza i posłuży nam do wyświetlenia informacji o tym, czy trafił on w prawidłowy przycisk, czy nie. W tym celu możemy stworzyć etykietę. Wykorzystamy do tego polecenie **Label()**. Jednak aby z niego skorzystać, powinniśmy tę etykietę stworzyć pod jakąś nazwą. Niech będzie to po prostu **etykieta**. Napiszmy jednak: **global**

```
def trafiony():
    for i in przyciski:
        i.destroy()
    global etykieta
```

etykieta, aby była ona dostępna również poza definicją tej procedury. Dopiero w kolejnej linijce użyjemy polecenia **etykieta = Label(t, text = "Trafiłeś dobry przycisk")**, gdzie **t** oznacza nazwę okna, a **"Trafiłeś dobry przycisk"** to napis, który ma być na tej etykiecie umieszczony.

```
def trafiony():
    for i in przyciski:
        i.destroy()
    global etykieta
    etykieta = Label(t, text = "Trafiłeś dobry przycisk")
```

24 Etykietę, jak przyciski, za pomocą polecenia **pack()** umieszczamy w oknie gry. Powinno to wyglądać następująco: **etykieta.pack(fill=BOTH,expand=YES)**.

```
def trafiony():
    for i in przyciski:
        i.destroy()
    global etykieta
    etykieta = Label(t, text = "Trafiłeś dobry przycisk")
    etykieta.pack(fill=BOTH,expand=YES)
```

Dzięki zastosowaniu takich samych ustawień jak w wypadku przycisków etykieta ustawi się idealnie na środku okna.

25 W pisanej właśnie procedurze możemy umieścić też mechanizm restartu gry. Robimy to poleceniem **t.after(5000, restart)**, gdzie **t** oznacza nazwę okna,

```
def trafiony():
    for i in przyciski:
        i.destroy()
    global etykieta
    etykieta = Label(t, text = "Trafiłeś dobry przycisk")
    etykieta.pack(fill=BOTH,expand=YES)
    t.after(5000,restart)
```

a **5000** to czas wyrażony w milisekundach, po którym ma zostać zrestartowana nasza gra. Jednak słowo **restart** samo w sobie nie jest poleceniem ponownego uruchomienia gry. To nazwa procedury, która ma się wykonać po upływie wcześniej określonego czasu. Zatem program nie będzie działać poprawnie aż do czasu zdefiniowania przez nas procedury **restart** (lub innej, jeżeli użyjemy tu innej nazwy).

26 Dodajemy więc definicję procedury **restart**.

```
def restart():
```

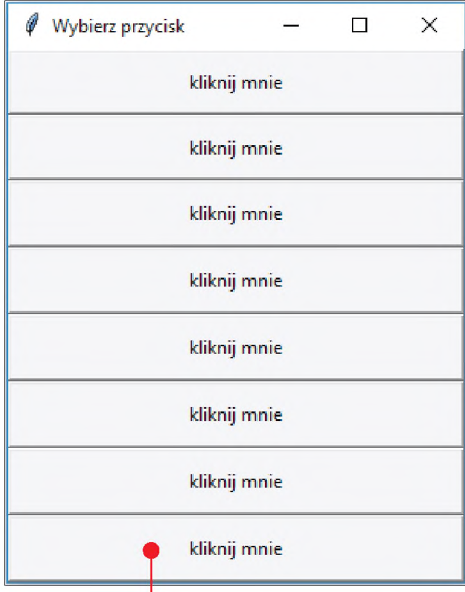
27 Ale czym właściwie ma być „restart" naszej gry? To usunięcie etykiety mówiącej o wyniku gry i ponowne wstawienie

```
def restart():
    etykieta.destroy()
```

przycisków. Poprzez polecenie **etykieta.destroy()** usuwamy etykietę z okna gry.

28 Poprzez uruchomienie procedury **wstaw_przyciski()** uruchamiamy kolejną rozgrywkę.

```
def restart():
    etykieta.destroy()
    wstaw_przyciski()
```



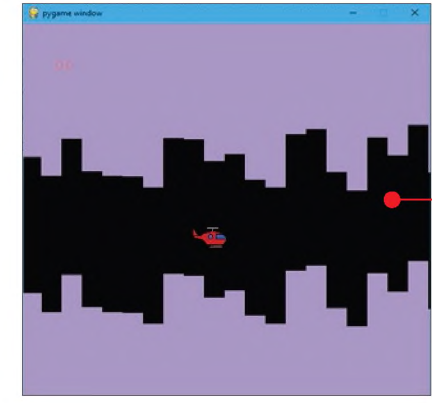
29 Do zakończenia całej gry brakuje nam jeszcze definicji procedury, która ma wykonywać się zawsze, gdy gracz naciśnie przycisk, który nie prowadzi do wygranej. Jest ona analogiczna do procedury **trafiony()**, musi mieć jednak inną nazwę, a tworzona przez nią etykieta powinna mieć inny napis – informujący o trafieniu w niewłaściwy przycisk.

30 Teraz próba uruchomienia skryptu nie będzie już powodować wyświetlania informacji o błędach. Powinno pokazać się okno gry. Przetestujmy jej działanie.

```
def pudlo():
    for i in przyciski:
        i.destroy()
    global etykieta
    etykieta = Label(t, text = "Trafiłeś zły przycisk")
    etykieta.pack(fill=BOTH,expand=YES)
    t.after(5000,restart)
```

GRA HELIKOPTERRR
Biblioteka do tworzenia gier

Znamy już podstawy wykorzystania modułu **tkinter** do tworzenia graficznych programów. Jednak **tkinter** wykorzystywany jest głównie do oprogramowania użytkowego. Gorzej sprawdza się w wypadku tworzenia gier. Dlatego jeśli myślimy o wykorzystywaniu Pythona do tworzenia gier, warto poznać moduł, który właśnie w tym celu został stworzony – **pygame**.
Nie jest on standardowo dołączony do Pythona, trzeba go zainstalować, aby móc programować za jego pomocą. Instalacja została



opisana w rozdziale **1**. Właśnie korzystając z tego modułu, zrealizujemy kolejne zadanie. Stworzymy grę, w której zadaniem gracza będzie sterowanie helikopterem lecącym wewnątrz jaskini. Gra będzie miała widok z boku, na środku okna gry znajdzie się grafika helikoptera, a u góry i na dole – nierówności jaskini.

Okno gry modułu pygame

1 Na początek trzeba użyć polecenia, za pomocą którego importujemy zainstalowany wcześniej moduł do programu. Możemy to zrobić na dwa sposoby: korzystając z polecenia **import pygame** lub **from pygame import*** – w tym wypadku dalej w kodzie polecenia z modułu pisalibyśmy bez podawania jego nazwy.

```
File  Edit  Format  Run  Options  Window  Help
import pygame
```

2 Aby móc swobodnie wykorzystywać różnego rodzaju polecenia z modułu **pygame**, należy wywołać dla niego **polecenie init()** poprzez linijkę **pygame.init()**. Bez tego część funkcji modułu nie będzie działać.

```
import pygame

pygame.init()
```

3 Ponieważ **pygame** podobnie jak **tkinter** daje nam do dyspozycji okno graficzne, trzeba określić wymiary tego okna. Musimy je podać podczas jego tworzenia. Inaczej niż w przypadku poprzedniej gry, tu w skrypcie będziemy na dalszych etapach odwoływać się do tych wymiarów. Aby było nam wygodnie z nich korzystać i równie wygodnie móc je zmienić, kiedy uznamy to za konieczne, utwórzmy dwie zmienne do ich przechowywania – **szer** i **wys**. Do nich wpisujemy odpowiednio szerokość i wysokość okna gry wyrażoną w pikselach, taką jaka nam odpowiada.

```
import pygame

pygame.init()

szer = 600
wys = 600
```

4 Mając już zapisane wymiary okna, możemy przejść do użycia polecenia, które je stworzy. Okno jak w przypadku **tkintera** musi mieć nazwę. Możemy nazwać je na przykład **screen**, a następnie poprzez polecenie **pygame.display.set_mode((szer,wys))** przypisać do tej nazwy okno gry o określonych wcześniej wymiarach.

```
szer = 600
wys = 600
screen = pygame.display.set_mode((szer,wys))
```

Wstawianie napisów

1 Mając już stworzone okno gry, możemy zacząć umieszczać w nim poszczególne elementy. Kilkakrotnie, w różnych miejscach okna będziemy potrzebowali wstawić tekst. W tym celu należy użyć kilku poleceń. Aby ułatwić sobie dalsze pisanie, możemy stworzyć procedurę **napisz** z parametrami, którą potem wykorzystamy do tworzenia napisów w oknie gry. Będzie ona pozwalała umieścić dowolny tekst dowolnej wielkości w dowolnym miejscu okna. Parametry, które będziemy podawać, wywołując procedurę,

powinniśmy zapisać w nawiasie podczas jej definiowania. Używamy zatem polecenia **def napisz(tekst, x, y, rozmiar):**.

```
def napisz(tekst, x, y, rozmiar):
```

2 Aby móc tworzyć napisy, niezbędne jest posiadanie obiektu, który taki napis wyrenderuje. Inaczej mówiąc, musimy mieć zapisaną jakąś czcionkę. Można użyć odniesienia do jednej z czcionek systemowych. Tworzymy obiekt poleceniem **cz = pygame.font.SysFont("Arial", rozmiar)**, gdzie **cz** jest nazwą nowego obiektu, **"Arial"** to nazwa czcionki, jakiej chcemy użyć, a **rozmiar** to nazwa ostatniego parametru, którego wartość podamy, wywołując procedurę.

```
def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
```

3 Następnie możemy przystąpić do renderowania napisu. Służy do tego polecenie **render()**, z którego korzystamy w następujący sposób: **rend = cz.render(tekst, 1, (255,100,100))**. W tej linijce kodu **rend** to nazwa nowego obiektu graficznego z napisem, **cz** to nazwa obiektu będącego czcionką,

```
def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
```

tekst to pierwszy parametr procedury, **1** to wartość argumentu typu prawda/fałsz. Jeśli mamy prawdę, czyli **1**, to utworzony napis będzie miał wygładzone krawędzie, a jeżeli wstawimy zamiast niej **0**, program nie wygładzi krawędzi w napisie.

4 Ostatni z argumentów - nawias z trzema liczbami **(255,100,100)** - to kolor, jakim chcemy stworzyć nasz napis. Kolor ten zapisany jest w notacji RGB. Opiera się ona na łączeniu trzech barw: czerwonej, zielonej i niebieskiej. W uproszczeniu wynika to z tego, że na wyświetlaczach w każdym pikselu mamy trzy diody w tych właśnie kolorach. Każda z nich może świecić z różną mocą wyrażoną liczbą od 0 do 255. Zależnie od tego, jak świecą diody, widzimy inny kolor. Jeśli wszystkie nie świecą - czyli (0,0,0) - mamy kolor czarny. Gdy wszystkie świecą maksymalnie mocno (255,255,255) - mamy kolor biały.
Skrót RGB pochodzi od nazw kolorów: **R**ed (czerwony), **G**reen (zielony), **B**lue (niebieski). I w takiej kolejności podajemy, jak mocno mają świecić poszczególne diody. Zatem dla przykładu: (0,255,0) oznacza kolor zielony, bo tylko ta dioda świeci, a pozostałe nie, więc zielony nie łączy się z inną barwą. Odpowiednio łącząc czerwony, zielony i niebieski - możemy uzyskać dowolny kolor.

5 W naszej procedurze **napisz** brakuje jeszcze jednego ważnego elementu - polecenia **screen.blit(rend, (x,y))** służącego do umieszczenia wyrenderowanego tekstu w oknie graficznym. W poleceniu tym

```
def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
    screen.blit(rend, (x,y))
```

screen oznacza nazwę okna gry, **rend** to stworzony linijkę wcześniej obiekt, a **x** i **y** to współrzędne, w których należy umieścić tekst. Podajemy je podczas wywoływania procedury.

6 Sprawdźmy teraz, czy wszystko działa. Wywołajmy procedurę **napisz**, podając w nawiasie parametry dotyczące tego, jaki tekst, gdzie i jakiej wielkości chcemy napisać: **napisz("Naciśnij spację, aby zacząć", 80, 150, 20)**.
Co oznaczają poszczególne pozycje? **"Naciśnij spację, aby zacząć"** - to tekst, który chcemy umieścić na ekranie. Ostatnia liczba w nawiasie, czyli **20**, to wielkość czcionki, jaką zostanie napisany ten tekst. A dwie po-

```
import pygame

pygame.init()

szer = 600
wys = 600
screen = pygame.display.set_mode((szer,wys))

def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
    screen.blit(rend, (x,y))

napisz("Naciśnij spację, aby zacząć", 80, 150, 20)
```

8 Jeśli chcemy zmienić cokolwiek, co jest narysowane w oknie gry tworzonej przez **pygame**, po wprowadzeniu zmian musimy odświeżyć ekran, aby zostały one wprowadzone w życie. W tym celu używamy polecenia **pygame.display.update()**.

zostałe liczby w nawiasie, czyli **80** i **150**? Są to współrzędne. Pierwsza z nich to **x** i mówi o tym, w jakiej odległości od lewej krawędzi okna gry chcemy zacząć umieszczać nasz tekst. A druga z nich, czyli **y**, decyduje, w jakiej odległości od góry okna zacznie się on wyświetlać. Odległości te wyrażone są w pikselach.

```
import pygame

pygame.init()

szer = 600
wys = 600
screen = pygame.display.set_mode((szer,wys))

def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
    screen.blit(rend, (x,y))

napisz("Naciśnij spację, aby zacząć", 80, 150, 20)

pygame.display.update()
```

7 Jeśli teraz uruchomimy skrypt, zobaczymy... czarny ekran. Nie zniechęcajmy się jednak, to nie jest wina błędu w kodzie. Brakuje po prostu jeszcze jednego ważnego polecenia.

9 Teraz próba uruchomienia skryptu powinna sprawić, że wyświetli się okno z napisem.

WYŚRODKOWANY NAPIS

Posiadając obiekt graficzny z renderowanym tekstem, możemy sprawdzić jego wymiary. Polecenie **render.get_rect()** zwraca obiekt **Rect** reprezentujący prostokąt, w jakim zmieścił się tekst. Z tego obiektu można odczytać atrybuty **width** i **height** – czyli szerokość i wysokość prostokąta. Czyli pod poleceniem **rend.get_rect().width** kryje się szerokość tekstu, a pod poleceniem **rend.get_rect().height** kryje się wysokość obiektu. Te dwie wartości możemy wykorzystać do wyśrodkowania napisu. W takim wypadku współrzędną **x** napisu obliczamy wzorem **(szer – rend.get_rect().width)/2** A, a współrzędną **y** wzorem **(wys – rend.get_rect().height)/2** B. Wywołując procedurę napisaną w ten sposób, uzyskamy napis na środku okna gry.

```
def napisz(tekst, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
A   x = (szer - rend.get_rect().width)/2
B   y = (wys - rend.get_rect().height)/2
    screen.blit(rend, (x,y))

napisz("Naciśnij spację, aby zacząć", 40)
```

Naciśnij spację, aby zacząć

Pętla główna gry

Okno, które stworzyliśmy w poprzedniej wskazówce, ma pewną dziwną cechę. Nie da się go normalnie zamknąć. Aby znikło, należy wyłączyć IDLE. Przycisk **x** w prawym górnym rogu nie działa. Dlaczego? Dlatego że to my powinniśmy go zaprogramować. A żeby to zrobić, musimy zapoznać się z zagadnieniem **eventów**, czyli zdarzeń. Zdarzenia to różne akcje możliwe do wykonania przez użytkownika. Zdarzeniem jest kliknięcie na przycisk **x** w rogu okna gry, ale też naciśnięcie klawisza na klawiaturze czy przycisku myszy oraz puszczenie ich, a nawet ruch kursora. Pygame ma listę eventów, które wystąpiły w danej chwili. Jeśli coś się zadziało, to trafia na listę, po czym za chwilę ją opuszcza, gdy przestaje się dziać.

1 Naszym zadaniem powinno być sprawdzanie listy eventów w poszukiwaniu tego zdarzenia, które mówi o próbie zamknięcia okna gry. Ponieważ lista zmienia zawartość, jej sprawdzanie powinno odbywać się regularnie, właściwie non stop. Aby powtarzać sprawdzanie, należy umieścić je w pętli. Gdybyśmy użyli do tego pętli **for**, musielibyśmy określić, ile razy ma się ono wykonać. Dlatego powinniśmy skorzystać z pętli **while**, która może wykonywać coś, dopóki spełniony jest określony w niej warunek. Pętlę **while** łatwo jednak napisać w taki sposób, aby wykonywała się zawsze aż do zamknięcia programu. Wystarczy zamiast warunku wpisać słowo **True** oznaczające prawdę. Pętlę umieszczamy przed wywołaniem procedury **napisz()**.

```
def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
    screen.blit(rend, (x,y))

while True:
```

2 Samo sprawdzanie powinno być stworzone poprzez pętlę **for**, ponieważ należy przechodzić po kolejnych elementach listy. Lista z występującymi aktualnie eventami to **pygame.event.get()**, zatem pętla **for** może wyglądać następująco: **for event in pygame.event.get():**.

```
while True:
    for event in pygame.event.get():
```

3 Następnie należy sprawdzić – poprzez **if** – czy typ eventu zgadza się z tym, którego poszukujemy, co zapisujemy linijką: **if event.type == pygame.QUIT**, bo **pygame.QUIT** to typ zdarzenia wskazujący na kliknięcie na przycisk **x** w rogu okna.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```

4 Zatem gdy zapisany warunek jest spełniony, należy wykonać dwa polecenia – **pygame.quit()**, które zamyka okno gry, i **quit()**, które dodatkowo zamknie IDLE.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

napisz("Naciśnij spację, aby zacząć", 80, 150, 20)

pygame.display.update()
```

5 Jeśli spróbujemy teraz uruchomić grę, ponownie zobaczymy czarny ekran – to dlatego, że wywołanie procedury **napisz** i odświeżenie ekranu znajdują się w skrypcie po pętli **while**, a nie w niej. Aby te dwie linijki się wykonały, pętla **while** musiałaby skończyć się wykonywać. Tak się jednak nie stanie, bo napisaliśmy ją w taki sposób, aby się nie kończyła podczas działania programu. Logicznym rozwiązaniem

jest umieszczenie dwóch wspomnianych linijek kodu wewnątrz pętli, czyli dodanie do nich jednego wcięcia. Dzięki temu ekran będzie na bieżąco aktualizowany, a my będziemy mogli wewnątrz pętli **while** decydować, co ma być wyświetlone na ekranie. Pętlę **while** można teraz nazywać **pętlą główną** naszej gry.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    napisz("Naciśnij spację, aby zacząć", 80, 150, 20)

    pygame.display.update()
```

Wczytanie logo gry

1 To, co ma być wyświetlone w oknie gry, powinno się zmieniać. Inny powinien być widok, gdy prowadzimy rozgrywkę, a inny, gdy dopiero chcemy ją zacząć. Dodany przez nas wcześniej napis nie powinien być widoczny w oknie podczas prowadzenia rozgrywki. To, co będzie pokazywane w oknie, możemy uzależniać od wartości jakiejś zmiennej. Utwórzmy zmienną **copokazuje** przed pętlą **while** i dajmy jej wartość początkową **"menu"**.

```
def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
    screen.blit(rend, (x,y))

copokazuje = "menu"

while True:
    for event in pygame.event.get():
```

2 Wewnątrz pętli **while** należałoby sprawdzać wartość tej zmiennej i jeśli jest ona równa **"menu"**, to dopiero wtedy powinniśmy umieszczać w oknie gry stworzony przez nas napis. Robimy to poprzez **if**, a do wywołania procedury **napisz()** dodajemy kolejne wcięcie, aby wykonywało się tylko wtedy, gdy warunek opisany w **if** jest spełniony.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
    if copokazuje == "menu":
        napisz("Naciśnij spację, aby zacząć", 80, 150, 20)
```

3 Nasza gra mogłaby wyświetlać coś jeszcze oprócz samego napisu mówiącego o tym, że aby rozpocząć rozgrywkę, należy wcisnąć spację. To mogłoby być logo naszej gry. Przygotujmy je w programie graficznym – dobrze sprawdza się tu **Piskel** dostępny online pod adresem **piskelapp.com** lub w wersji do instalacji na płycie dołączonej do książki (DVD-KOD: 011).

4 Przygotowaną grafikę umieszczamy w tym samym folderze, w którym mamy zapisany skrypt tworzonej właśnie gry. Do wczytania grafiki służy polecenie **pygame.image.load()** – w nawiasie należy podać ścieżkę do pliku graficznego. Unika się jednak podawania pełnych ścieżek do pliku. Lepszym rozwiązaniem jest trzymanie grafiki w folderze ze skryptem i z poziomu skryptu odniesienie się tylko do pliku znajdującego się w tej samej lokalizacji co skrypt.

5 Aby odnieść się do pliku, który znajduje się w tej lokalizacji co skrypt, używamy polece-

nia **os.path.join()**, w nawiasie podając nazwę pliku. Natomiast samo polecenie zwraca nam już ścieżkę do pliku. Więc to polecenie należałoby umieścić w nawiasie polecenia **pygame.image.load()**. Natomiast taki wczytany plik również powinniśmy zapisać w kodzie pod jakąś nazwą, na przykład **grafika** **A**.

```
if copokazuje == "menu":
    napisz("Naciśnij spację, aby zacząć", 80, 150, 20)
    grafika = pygame.image.load(os.path.join('logo.png'))
```

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
    if copokazuje == "menu":
        napisz("Naciśnij spację, aby zacząć", 80, 150, 20)
        grafika = pygame.image.load(os.path.join('logo.png'))
        screen.blit(grafika, (80,30))

    pygame.display.update()
```

6 To dlatego, że dalej musimy odnieść się do obiektu o nazwie **grafika**, aby umieścić go na ekranie. Do tego wykorzystujemy używane wcześniej polecenie **blit()** w formie **screen.blit(grafika, (80,30))**, dobierając takie współrzędne, aby logo dobrze wyglądało w oknie gry.

7 Próba uruchomienia programu w obecnej formie zwróci nam błąd. To dlatego, że korzystamy z polecenia **os.path.join()**, które znajduje się w module **os**. Aby móc z niego korzystać, musimy ten moduł importować.

```
*h.py - C:\Users\konra\Documents\h.py (3.7.3)*
File  Edit  Format  Run  Options  Window  Help
import pygame
import os
```

```
Python 3.7.3 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Inte
l)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=================== RESTART: C:\Users\konra\Documents\h.py ===================
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
Traceback (most recent call last):
  File "C:\Users\konra\Documents\h.py", line 23, in <module>
    grafika = pygame.image.load(os.path.join('logo.png'))
NameError: name 'os' is not defined
>>> |
```

Tworzymy przeszkody

Kolejnym krokiem w realizacji naszego zadania będzie napisanie, jak ma wyglądać rozgrywka. W naszym przykładzie na ekranie ma być widoczny helikopter i jaskinia. Jaskinia będzie pokazywana w przekroju, potrzebna będzie oddzielnie jej góra i dół. Zarówno góra, jak i dół mają być nieregularne (jak na rysunku na stronie 28). Jaskinię zbudujemy z ciągów prostokątów o losowych wymiarach. Aby pary prostokątów, dolny i górny będące w jednej linii, nie nachodziły na siebie, uznamy je za jedną przeszkodę, a między nimi ustalimy stały odstęp o takiej wielkości, by helikopter zmieścił się pomiędzy nimi. Takich par prostokątów będzie wiele. I tu przechodzimy do bardzo ważnego aspektu programowania w Pythonie, a mianowicie do programowania obiektowego. Python to język zorientowany obiektowo. Obiekt jest instancją klasy. Jeśli chcemy mieć obiekty reprezentujące poszczególne przeszkody, musimy mieć klasę, która definiuje taką przeszkodę. Tworząc klasy, zwracamy szczególną uwagę na to, co jest wspólne, a co różne dla obiektów. Zagadnienie programowania obiektowego najlepiej poznać na przykładzie.

1 Naszym przykładem będzie klasa **Przeszkoda**, której definicję zaczynamy od **class Przeszkoda():**. Piszemy ją przed pętlą główną gry.

```
copokazuje = "menu"

class Przeszkoda():

while True:
    for event in pygame.event.get():
```

2 Podstawą w klasie jest **konstruktor** – to taka metoda, za pomocą której tworzymy nowe obiekty. W Pythonie konstruktor ma nazwę **__init__**, zatem zapisujemy: **def __init__():**.

```
class Przeszkoda():
    def __init__():
```

PROGRAMOWANIE OBIEKTOWE

To sposób programowania, w którym programy definiuje się za pomocą **obiektów** – elementów łączących **stan** (czyli dane, nazywane najczęściej polami) i **zachowanie** (czyli procedury i funkcje, w programowaniu obiektowym nazywane metodami). Obiektowy program komputerowy rozumiany jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

3 Jeśli wywołując konstruktor, mamy tworzyć nową przeszkodę, powinniśmy mieć wpływ na jej ostateczny wygląd. Chcemy wskazać, gdzie ma się ona utworzyć, a także jakiego ma być rozmiaru. W tym celu musimy podać parametry do konstruktora, tak abyśmy te elementy mogli ustawiać, tworząc obiekt, czyli wywołując konstruktor. Piszemy zatem: **def __init__(x, szerokosc):**. To oznacza, że pierwszym argumentem podawanym przy tworzeniu obiektu będzie współrzędna **x**, na której ma pojawić się przeszkoda, a drugim szerokość tej przeszkody.

```
class Przeszkoda():
    def __init__(x, szerokosc):
```

4 Każdy obiekt jest widziany sam przez siebie pod nazwą **self**. Nazwy tej używa się wewnątrz klasy w tych miejscach, w których potrzebne jest odwołanie obiektu do niego samego. Nazwa **self** powinna być pierwszym argumentem wszystkich deklarowanych w klasie metod, w tym konstruktora. Musimy dopisać **self** jako pierwszy parametr konstruktora. W wywołanej metodzie reprezentuje ona aktualnie użyty obiekt.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
```

Wśród argumentów użytych podczas wywołania **self** nie ma odpowiednika. To oznacza, że nawet po wprowadzeniu tej zmiany pierwszym argumentem podawanym przy tworzeniu obiektu będzie współrzędna **x**, na której ma pojawić się przeszkoda, a drugim **szerokosc** tej przeszkody.

5 Zajmijmy się jeszcze danymi w treści konstruktora, które zostaną podane podczas jego uruchamiania. Obiekt klasy ma **pola** – są to jego właściwości, cechy szczególne. Dla naszej przeszkody taką właściwością, cechą szczególną, powinna być współrzędna **x**, na której przeszkoda się znajduje, i **szerokosc** przeszkody.
Aby tworzyć właściwości obiektu, w konstruktorze używamy słowa **self**, po nim stawiamy kropkę, a po kropce piszemy nazwę właściwości. Do tak utworzonego pola klasy powinniśmy przypisać wartość. Jeśli tworzymy pole **x**, to możemy przypisać do niego wartość bezpośrednio z parametru, poprzez **self.x = x**.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
```

6 Podobnie powinniśmy zrobić z szerokością.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
```

7 Nasza przeszkoda to tak naprawdę dwa prostokąty: jeden u góry, a drugi u dołu okna gry. Dlatego wszystkie pola klasy muszą charakteryzować te dwa prostokąty. Aby narysować prostokąt, trzeba znać jego lokalizację w oknie. Ponieważ prostokąty mają być w jednej linii, mają wspólną współrzędną **x**. Inaczej jest ze współrzędną **y** – jest inna dla górnego i dolnego prostokąta. Musimy zatem zadeklarować dwa oddzielne pola klasy dla współrzędnych **y** tych dwóch prostokątów: jedno dla **y** górnego prostokąta, drugie dla **y** dolnego prostokąta. To pierwsze **y** będzie nam znacznie łatwiej wyznaczyć. Ponieważ współrzędna **y** to odległość od górnej krawędzi okna gry, a prostokąt ma przylegać do krawędzi okna, odległość powinna wynosić **0**. Piszemy zatem **self.y_gora = 0**.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
```

8 Ponieważ kolejne przeszkody, jakie pojawiają się na ekranie, mają się od siebie różnić, powinny mieć różne wysokości. Najlepiej, aby ich wysokości były losowe.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
```

Tworzymy pole **wys_gora**, które będzie odpowiadać za wysokość górnego prostokąta, i przypisujemy do niego wartość zwróconą przez polecenie **random.randint(150,250)**.

9 Odległość pomiędzy górnym i dolnym prostokątem w przypadku każdej przeszkody powinna być taka sama. Stwórzmy pole **odstep** i wpiszmy do niego wartość **200**. W odległości tylu pikseli od górnego prostokąta powinien pojawić się dolny.

10 Dzięki temu wiemy już, jaka powinna być wartość współrzędnej **y** dla dolnego prostokąta: o odstęp większa niż wysokość górnego prostokąta. Tworzymy pole **y_dol** i wpisujemy działanie, które do

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
        self.odstep = 200
```

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
        self.odstep = 200
        self.y_dol = self.wys_gora + self.odstep
```

wysokości górnego prostokąta doda odstęp między prostokątami.

11 Dalej możemy obliczyć wysokość dolnego prostokąta i utworzyć dla niej odpowiednie pole, a obliczamy ją, sprawdzając, ile pikseli od współrzędnej **y** zostało do końca wysokości okna. Inaczej mówiąc, od wysokości okna należy odjąć współrzędną **y** dolnego prostokąta. Wysokość okna mamy zapisaną już wcześniej w zmiennej **wys**.

12 Pola klasy miały być cechami charakterystycznymi dla dwóch prostokątów. Wiemy, jakie są ich wymiary i lokalizacja, ale nie znamy jeszcze ostatecznego wyglądu prostokątów, bo nie mamy określonego ich koloru. Kolor także zapiszmy jako pole klasy. Do tego pola należy przypisać połączone w jeden nawias trzy liczby – kolor jest podany w systemie RGB, którego działanie zostało już w tym rozdziale wyjaśnione (patrz strona 30).

W ten sposób zdefiniowaliśmy konstruktor dla przeszkód. Nie jest to jednak cała definicja klasy. Przeszkodę na podstawie pól, które jej nadaliśmy, trzeba jeszcze narysować. W tym celu w klasie napiszemy odpowiednią metodę.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
        self.odstep = 200
        self.y_dol = self.wys_gora + self.odstep
        self.wys_dol = wys - self.y_dol
        self.kolor = (160,140,190)
```

Dodatkowo, musimy zastanowić się nad działaniem naszej gry. Nie chcemy, by helikopter poruszał się po osi X – cały czas powinien być na środku ekranu. A żeby powstało złudzenie jego ruchu, powinny przesuwać się przeszkody. Zatem niezbędne będzie też dopisanie metody odpowiadającej za ruch przeszkód.

Rysowanie przeszkód

Pygame ma odpowiednie polecenie, którym możemy posłużyć się w celu narysowania przeszkód.

1 Polecenia tego użyjemy w metodzie **rysuj**, którą musimy utworzyć. Należy pamiętać o odniesieniu się obiektu do samego siebie, czyli o wyrazie **self** w nawiasie. Jest on niezbędny w każdej metodzie klasy.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
        self.odstep = 200
        self.y_dol = self.wys_gora + self.odstep
        self.wys_dol = wys - self.y_dol
        self.kolor = (160,140,190)
    def rysuj(self):
```

2 Wewnątrz metody **rysuj** należy użyć polecenia do tworzenia graficznej repre-

```
def rysuj(self):
    pygame.draw.rect(screen, self.kolor, self.ksztalt_gora, 0)
```

(etykiety nad kodem: A – screen, B – self.kolor, C – self.ksztalt_gora, D – 0)

zentacji prostokąta. Nasze polecenie powinno mieć postać ●, gdzie:

A – oznacza odniesienie do okna graficznego
B – oznacza kolor, jakim rysujemy prostokąt
C – oznacza prostokąt, który chcemy narysować
D – oznacza grubość krawędzi, jaką ma mieć prostokąt. Zastosowanie liczby **0** sprawia, że prostokąt nie będzie narysowany z krawędzią, ale cały będzie wypełniony kolorem. Gdybyśmy podali liczbę większą od zera, prostokąt powstałby pusty w środku z ramką określonej grubości.

Punkty **A**, **B** i **D** powinny być dość jasne. Gorzej może być z punktem **C**. Oznacza on, że aby narysować prostokąt, trzeba mieć prostokąt – obiekt klasy **Rect** obecnej w **pygame**. Zatem jeśli chcemy użyć podczas rysowania **self.ksztalt_gora**, nasza klasa musi mieć pole o nazwie **ksztalt_gora**, które będzie obiektem klasy **Rect** odpowiadającym prostokątowi z górnej części przeszkody.

3 Dodajemy w konstruktorze pole **self.ksztalt_gora** i przypisujemy do niego obiekt poprzez polecenie **pygame.Rect(self.x, self.y_gora, self.szerokosc, self.wys_gora)** ●, gdzie w nawiasie należy podać kolejno współrzędną **x** prostokąta, współrzędną **y** prostokąta, jego szerokość i wysokość. Teraz mamy już górny prostokąt i jego reprezentację graficzną.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
        self.odstep = 200
        self.y_dol = self.wys_gora + self.odstep
        self.wys_dol = wys - self.y_dol
        self.kolor = (160,140,190)
        self.ksztalt_gora = pygame.Rect(self.x, self.y_gora, self.szerokosc, self.wys_gora)
```

4 Aby dopełnić przeszkodę, w konstruktorze powinniśmy utworzyć dolny prostokąt poleceniem **self.ksztalt_dol = pygame.Rect(self.x, self.y_dol, self.szerokosc, self.wys_dol)** ●. Następnie dodajemy jego reprezentację graficzną ● w procedurze **rysuj**.
Działanie napisanej procedury będziemy mogli przetestować, dopiero gdy zostaną utworzone obiekty przeszkody.

```
def rysuj(self):
    pygame.draw.rect(screen, self.kolor, self.ksztalt_gora, 0)
    pygame.draw.rect(screen, self.kolor, self.ksztalt_dol, 0)
```

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
        self.odstep = 200
        self.y_dol = self.wys_gora + self.odstep
        self.wys_dol = wys - self.y_dol
        self.kolor = (160,140,190)
        self.ksztalt_gora = pygame.Rect(self.x, self.y_gora, self.szerokosc, self.wys_gora)
        self.ksztalt_dol = pygame.Rect(self.x, self.y_dol, self.szerokosc, self.wys_dol)
```

Tworzymy obiekty przeszkody

Aby zobaczyć, czy dobrze napisaliśmy klasę **Przeszkoda**, spróbujemy stworzyć obiekty tej klasy. By łatwiej było nam się odnosić do wszystkich przeszkód jednocześnie, umieścimy je na liście.

1 Poleceniem **przeszkody = []** deklarujemy odpowiednią listę. Polecenia używamy po deklaracji klasy, przed pętlą **while**.

```
przeszkody = []

while True:
    for event in pygame.event.get():
```

2 Po zadeklarowaniu listy musimy zapełnić ją obiektami. Do dodawania elementów do listy służy polecenie **append()**, gdzie w nawiasie należy użyć konstruktora klasy **Przeszkoda**. Wywołujemy go, podając nazwę klasy, po której w kolejnym nawiasie wpisujemy parametry obiektu, który tworzymy, w tym wypadku będzie to współrzędna **x** przeszkody i jej szerokość.

3 Załóżmy, że chcemy wypełnić ekran dwudziestoma identycznej szerokości przeszkodami. Gdy napiszemy już, aby zaczęły się one poruszać, to gdy pierwsza z nich zacznie zjeżdżać z okna gry, w jej miejsce, z drugiej strony okna, powinna zacząć pojawiać się kolejna, w naszym przykładzie dwudziesta pierwsza przeszkoda. Tyle właśnie przeszkód może być maksymalnie jednocześnie widocznych. Dlatego tyle przeszkód należy utworzyć. Przeszkody tworzymy w pętli **for**, która ma się wykonać 21 razy.

```
przeszkody = []
for i in range(21):
```

4 Wewnątrz pętli **for** wykorzystujemy wspomniane wcześniej polecenie **append** w formie **przeszkody.append (Przeszkoda())**. Do uzupełnienia pozostaną nam wtedy już tylko parametry konstruktora. Działamy w pętli, mamy więc dostęp do zmiennej **i**. Musimy wykorzystać ją tak, aby każda kolejna tworzona przez nas przeszkoda miała inną współrzędną **x**, ale żeby kolejne przeszkody przylegały do siebie. Pewne jest to, że jeśli całą szerokość okna gry ma wypełnić 20 przeszkód, to szerokość pojedynczej przeszkody będzie wynosiła 1/20 szerokości okna gry, czyli **szer/20** – to druga z wartości, jaką podajemy.

```
przeszkody = []
for i in range(21):
    przeszkody.append(Przeszkoda())
```

```
przeszkody = []
for i in range(21):
    przeszkody.append(Przeszkoda(i*szer/20,szer/20))
```

Wróćmy do pierwszej. Każda kolejna z przeszkód jest o szerokość przeszkody bardziej odsunięta od lewej krawędzi okna gry od poprzedniej. Możemy zapisać to wzorem **i*szer/20**.

Wywołujemy rysowanie przeszkód

Dla każdego elementu listy przeszkody należy wywołać metodę **rysuj**.

1 Najpierw ustalmy, kiedy należy to zrobić, gdzie to zapisać? Na pewno wewnątrz pętli **while**. Ale nie wtedy, kiedy gra pokazuje menu. Zatem trzeba rysować przeszkody, gdy zmienna **copokazuje** ma wartość inną niż **"menu"**. Niech będzie to wartość **"rozgrywka"**. Do **if** sprawdzającego wartość tej

```
if copokazuje == "menu":
    napisz("Nacisnij spację, aby zacząć", 80, 150, 20)
    grafika = pygame.image.load(os.path.join('logo.png'))
    screen.blit(grafika, (80,30))
elif copokazuje == "rozgrywka":
```

zmiennej dodajemy sekcję **elif copokazuje == "rozgrywka"**.

2 Gdy opisany warunek jest spełniony, poprzez pętlę **for** będziemy przechodzić po wszystkich elementach tablicy przeszkody.

```
elif copokazuje == "rozgrywka":
    for p in przeszkody:
```

3 Dla poszczególnych jej elementów uruchamiamy metodę **rysuj()**.

```
elif copokazuje == "rozgrywka":
    for p in przeszkody:
        p.rysuj()
```

4 Jednak w obecnym stanie skryptu nie mamy możliwości ręcznego przełączenia wartości zmiennej **copokazuje**. Program cały czas pokazywałby menu. Dla testów zmieńmy wartość zmiennej przy jej deklaracji, dając jej wartość początkową **"rozgrywka"**. Wtedy menu się nie narysuje, a powinny od razu pokazać się przeszkody.

5 W klasie jedno z pól ma nadawaną wartość losową: **self.wys_gora = random.randint(150,250)**, a polecenie, którego używamy, znajduje się w module **random**. Tymczasem my nie importowaliśmy wcześniej tego modułu. Należy to zrobić, ponieważ w przeciwnym wypadku próba uruchomienia programu wywoła komunikat o błędzie.

```
def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
    screen.blit(rend, (x,y))

copokazuje = "rozgrywka"
```

6 Importujemy moduł **random**. Teraz już uda nam się uruchomić program – powinniśmy zobaczyć przekrój jaskini.

```
=================== RESTART: C:\Users\konra\Documents\h.py ===================
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
Traceback (most recent call last):
  File "C:\Users\konra\Documents\h.py", line 35, in <module>
    przeszkody.append(Przeszkoda(i*szer/20,szer/20))
  File "C:\Users\konra\Documents\h.py", line 22, in __init__
    self.wys_gora = random.randint(150,250)
NameError: name 'random' is not defined
>>>
```

Wprawienie przeszkód w ruch

Nasze przeszkody powinny zacząć się poruszać. Dzięki temu uzyskamy złudzenie, że umieszczony na środku ekranu helikopter, który będzie poruszał się tylko góra-dół, leci przez jaskinię.

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
        self.odstep = 200
        self.y_dol = self.wys_gora + self.odstep
        self.wys_dol = wys - self.y_dol
        self.kolor = (160,140,190)
        self.ksztalt_gora = pygame.Rect(self.x, self.y_gora, self.szerokosc, self.wys_gora)
        self.ksztalt_dol = pygame.Rect(self.x, self.y_dol, self.szerokosc, self.wys_dol)
    def rysuj(self):
        pygame.draw.rect(screen, self.kolor, self.ksztalt_gora, 0)
        pygame.draw.rect(screen, self.kolor, self.ksztalt_dol, 0)
    def ruch(self, v):
```

```
def ruch(self, v):
    self.x = self.x - v
    self.ksztalt_gora = pygame.Rect(self.x, self.y_gora, self.szerokosc, self.wys_gora)
    self.ksztalt_dol = pygame.Rect(self.x, self.y_dol, self.szerokosc, self.wys_dol)
```

1 Niezbędna jest definicja nowej metody w klasie **Przeszkoda**. Pozwoli ona na przesunięcie obiektu. Piszemy zatem **def ruch(self, v):**, gdzie **v** będzie oznaczać prędkość, z jaką ma poruszać się przeszkoda. Będziemy ją podawać w momencie wywoływania metody.

2 Ruch przeszkody to tak naprawdę przesunięcie jej w lewą stronę. A to znaczy, że zmniejsza się odległość przeszkody od lewej krawędzi okna. O tej odległości informuje nas pole **x** obiektu. Musimy zatem zmniejszyć **self.x** o wartość **v**.

```
def ruch(self, v):
    self.x = self.x - v
```

3 Po zmianie wartości pola **x** powinniśmy zmienić też prostokąty reprezentujące przeszkody, czyli **self.ksztalt_dol** i **self.ksztalt_gora**. Gdybyśmy nie nadali im nowych wartości, przeszkody dalej rysowałyby się tam, gdzie wcześniej, bo prostokąty nadal miałyby w sobie zapisany stary **x**. Zatem ponownie dajemy wartości tym dwóm polom, używając tych samych poleceń co w konstruktorze.

4 Tak stworzoną metodę możemy uruchomić. Piszemy **p.ruch(1)** przed wywołaniem rysowania przeszkód, czyli w pętli najpierw przesuwamy każdą przeszkodę o 1 piksel, a potem ją rysujemy.

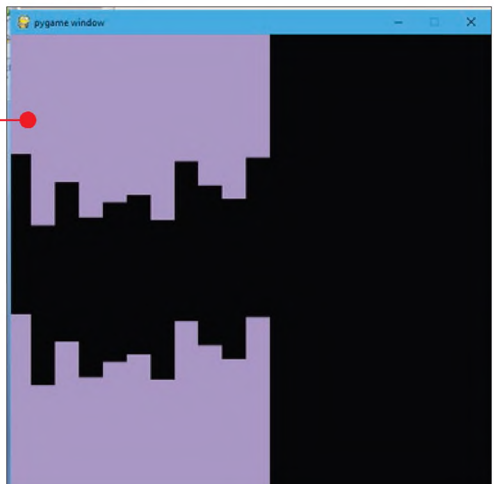
```
elif copokazuje == "rozgrywka":
    for p in przeszkody:
        p.ruch(1)
        p.rysuj()
```

5 Zwróćmy uwagę na jeszcze jeden szczegół. Rysowanie odbywa się w pętli. Przeszkody po przesunięciu wyrysują się w miejscu, w którym były już narysowane przeszkody przed przesunięciem. Każde kolejne wywołanie rysowania będzie nakładać nową warstwę na okno gry, które zaraz stanie się nieczytelne. Powinniśmy wymazywać to, co było narysowane wcześniej, z każdym przejściem pętli **while**. Dlatego przed **if** sprawdzającym, co należy umieścić w oknie gry, powinniśmy oczyś-

```
screen.fill((0,0,0))
if copokazuje == "menu":
    napisz("Nacisnij spację, aby zacząć", 80, 150, 20)
    grafika = pygame.image.load(os.path.join('logo.png'))
    screen.blit(grafika, (80,30))
elif copokazuje == "rozgrywka":
    for p in przeszkody:
        p.ruch(1)
        p.rysuj()
```

cić okno. Robimy to poleceniem **screen .fill((0,0,0))**, które wypełnia okno gry kolorem **(0,0,0)** wyrażonym w RGB. Jest to kolor czarny.

6 Po tych zmianach ruch przeszkód powinien już działać, co można zaobserwować po uruchomieniu gry. Przeszkody wyjeżdżają z okna gry, a w ich miejsce powinny pojawiać się kolejne przeszkody. Póki co, tych nie widać. Należy napisać skrypt, który – gdy przeszkoda opuści okno gry – w jej miejsce stworzy kolejną przeszkodę z drugiej strony okna gry, aby była ciągłość przeszkód.



7 Po pętli **for** do rysowania i przesuwania przeszkód piszemy kolejną pętlę **for**, której zadaniem będzie przejście po wszystkich przeszkodach.

```
elif copokazuje == "rozgrywka":
    for p in przeszkody:
        p.ruch(1)
        p.rysuj()
    for p in przeszkody:
```

8 Wewnątrz nowej pętli należy sprawdzać, czy współrzędna **x** przeszkody ma taką wartość, że całą szerokością przeszkoda znajduje się poza oknem gry. Zapisujemy to poleceniem **if p.x <= -p.szerokosc:**.

```
for p in przeszkody:
    if p.x <= -p.szerokosc:
```

9 Jeśli zapisany warunek jest prawdą, należy taką przeszkodę usunąć z listy poleceniem **przeszkody.remove(p)**.

```
for p in przeszkody:
    if p.x <= -p.szerokosc:
        przeszkody.remove(p)
```

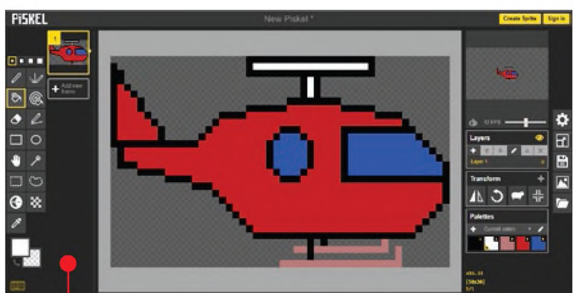
10 Następnie do listy z przeszkodami dodajemy nową przeszkodę poprzez polecenie **przeszkody.append((Przeszkoda(szer,szer/20)))**.

Po tak wprowadzonych zmianach ruch przeszkód powinien już być ciągły – gdy jedna opuści okno gry, pojawia się następna.

```
for p in przeszkody:
    if p.x <= -p.szerokosc:
        przeszkody.remove(p)
        przeszkody.append((Przeszkoda(szer,szer/20)))
```

Tworzymy helikopter

W naszym skrypcie nie napisaliśmy jeszcze nic o czymś bardzo ważnym, czyli o obiekcie, którym ma sterować gracz – o helikopterze. Aby go stworzyć, napiszmy oddzielną klasę.



1 Nową klasę nazwijmy **Helikopter** i zdefiniujmy ją poniżej klasy **Przeszkoda**.

```
class Helikopter():
```

2 Tworząc obiekt tej klasy, będziemy podawać tylko miejsce, w którym mamy go umieścić, czyli **x** i **y**. Takie parametry powinien mieć konstruktor tej klasy.

```
class Helikopter():
    def __init__(self, x, y):
```

3 Na podstawie tych parametrów powinniśmy utworzyć pola klasy, do których trafią wartości z parametru.

```
class Helikopter():
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

4 A teraz musimy na chwilę zostawić nasz skrypt. Helikopter musi mieć grafikę. Trzeba więc przygotować plik graficzny z rysunkiem helikoptera. Możemy skorzystać z programu **Piskel**. Pamiętajmy, że odstęp pomiędzy przeszkodami (dolną i górną) ustawiliśmy na 200 pikseli, dlatego grafika helikoptera nie może być zbyt duża. Sprawdźmy wymiary grafiki. Jej wielkość zapiszmy w polach klasy **Helikopter**. W naszym przykładzie plik ma 30 pikseli wysokości i 50 pikseli szerokości.

ROZMIAR OBRAZKA

Aby sprawdzić rozmiary obrazka, otwieramy folder z grafiką i klikamy na nią prawym przyciskiem myszki. Z menu kontekstowego wybieramy **Właściwości**. W nowym oknie przechodzimy do zakładki **Szczegóły** – tu odczytamy szerokość i wysokość grafiki.

```
class Helikopter():
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.wysokosc = 30
        self.szerokosc = 50
```

5 Aby móc wygodnie sprawdzać, czy helikopter koliduje z przeszkodami, klasa ta powinna mieć też pole z obiektem **Rect**, tak jak przeszkoda, **pygame** ma bowiem mechanizm sprawdzania, czy dwa obiekty typu **Rect** kolidują ze sobą.

6 Ostatnim polem tej klasy powinno być odniesienie do pliku graficznego z helikopterem. Tu mechanizm jest taki sam, jak w przypadku pliku z logo gry. Plik umieszczamy oczywiście w folderze ze skryptem i używamy polecenia odnoszącego się do lo-

```
class Helikopter():
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.wysokosc = 30
        self.szerokosc = 50
        self.ksztalt = pygame.Rect(self.x, self.y, self.szerokosc, self.wysokosc)
```

```
def ruch(self, v):
    self.y = self.y + v
    self.ksztalt = pygame.Rect(self.x, self.y, self.szerokosc, self.wysokosc)
```

```
class Helikopter():
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.wysokosc = 30
        self.szerokosc = 50
        self.ksztalt = pygame.Rect(self.x, self.y, self.szerokosc, self.wysokosc)
        self.grafika = pygame.image.load(os.path.join('helikopter.png'))
```

kalizacji skryptu i w niej szukającego pliku o określonej nazwie.

7 Klasa **Helikopter** również powinna mieć metody do rysowania i przesuwania obiektu. Zacznijmy od metody do rysowania. W jej definicji nie rysujemy prostokątów, jak w wypadku przeszkód, tylko umieszczamy na ekranie plik graficzny. Robimy to analogicznie, jak postępowaliśmy z logo gry. W tym celu używamy polecenia **screen.blit()**.

```
def rysuj(self):
    screen.blit(self.grafika, (self.x,self.y))
def ruch(self, v):
    self.y = self.y + v
```

8 Metoda **ruch()** dla helikoptera również powinna pobierać parametr **v** mówiący o prędkości, a także o kierunku ruchu obiektu. W metodzie tej należy dodawać wartość **v** do pola **y**. Gdy **v** będzie dodatnie, **y** zwiększy się i helikopter będzie niżej. Gdy **v** będzie ujemne – **y** się zmniejszy, a helikopter przesunie się w górę.

9 Aby później poprawnie wykrywać kolizje pomiędzy przeszkodami a helikopterem po przesunięciu helikoptera, trzeba zaktualizować miejsce, w którym znajduje się pole **ksztalt** z obiektem **Rect**.

```
przeszkody = []
for i in range(21):
    przeszkody.append(Przeszkoda(i*szer/20,szer/20))

gracz = Helikopter(250,275)
```

10 Klasa **Helikopter** jest już gotowa – możemy utworzyć jej obiekt. Robimy to poniżej miejsca, w którym stworzyliśmy obiekty przeszkody. Obiekt klasy **Helikopter** nazywamy **gracz**, a wywołując konstruktor, podajemy współrzędne, gdzie chcemy umieścić helikopter.

11 Od razu tworzymy też zmienną **dy** o wartości początkowej **0**. Wykorzystamy ją do określania kierunku, w któ-

```
przeszkody = []
for i in range(21):
    przeszkody.append(Przeszkoda(i*szer/20,szer/20))

gracz = Helikopter(250,275)

dy = 0
```

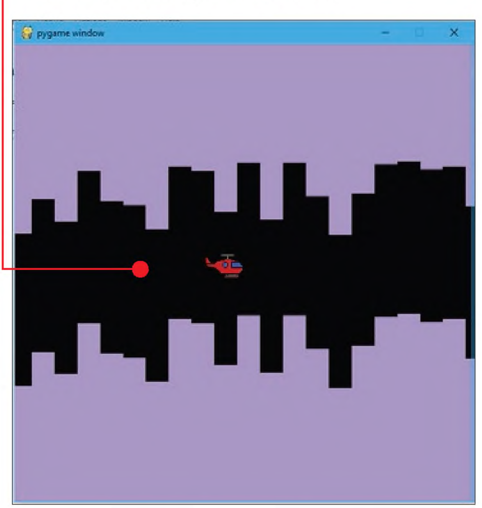
```
class Helikopter():
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.wysokosc = 30
        self.szerokosc = 50
        self.ksztalt = pygame.Rect(self.x, self.y, self.szerokosc, self.wysokosc)
        self.grafika = pygame.image.load(os.path.join('helikopter.png'))
    def rysuj(self):
        screen.blit(self.grafika, (self.x,self.y))
```

rym ma przesuwać się helikopter.

```
elif copokazuje == "rozgrywka":
    for p in przeszkody:
        p.ruch(1)
        p.rysuj()
    for p in przeszkody:
        if p.x <= -p.szerokosc:
            przeszkody.remove(p)
            przeszkody.append((Przeszkoda(szer,szer/20)))
    gracz.rysuj()
    gracz.ruch(dy)
```

12 Dla obiektu **gracz** należy wywołać metody **rysuj** i **ruch**. Robimy to w pętli **while**, gdy zmienna **copokazuje** ma wartość **"rozgrywka"**.

13 Po uruchomieniu skryptu widzimy już helikopter w oknie gry, tylko że... nie możemy nim sterować.



14 Aby dodać sterowanie helikopterem za pomocą strzałek, musimy powrócić do naszej pętli **for**, która przechodzi po wszystkich wyłapanych podczas działania gry zdarzeniach. Jeśli wykryje zdarzenia wciśnięcia klawisza, konieczna będzie reakcja. Zdarzenie wciśnięcia klawisza ma typ **KEYDOWN**, zatem piszemy **if event.type == pygame.KEYDOWN:**.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
        if event.type == pygame.KEYDOWN:
```

15 Kiedy warunek jest spełniony, musimy sprawdzić, który z klawiszy został naciśnięty. Piszemy w tym celu dwie instrukcje **if**: jedną do sprawdzenia, czy była to strzałka w górę – poleceniem **if event.key == pygame.K_UP:**, oraz drugą dla strzałki w dół. Jeśli wciśniętym klawiszem była strzałka w górę, dajemy **dy** wartość ujemną, na przykład -1. A jeżeli strzałka w dół – wartość dodatnią, na przykład 1.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        pygame.quit()
        quit()
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_UP:
            dy = -1
        if event.key == pygame.K_DOWN:
            dy = 1
```

Kolizje

Gra powinna być przerywana, gdy helikopter uderzy w przeszkodę.

1 Dodajemy do klasy **Przeszkoda** nową metodę **kolizja()**. W parametrze powinna ona przyjmować obiekt typu **Rect** – będziemy tam podawać obiekt typu **Rect** z obiektu gracza.

```
def kolizja(self, player):
```

```
class Przeszkoda():
    def __init__(self, x, szerokosc):
        self.x = x
        self.szerokosc = szerokosc
        self.y_gora = 0
        self.wys_gora = random.randint(150,250)
        self.odstep = 200
        self.y_dol = self.wys_gora + self.odstep
        self.wys_dol = wys - self.y_dol
        self.kolor = (160,140,190)
        self.ksztalt_gora = pygame.Rect(self.x, self.y_gora, self.szerokosc, self.wys_gora)
        self.ksztalt_dol = pygame.Rect(self.x, self.y_dol, self.szerokosc, self.wys_dol)
    def rysuj(self):
        pygame.draw.rect(screen, self.kolor, self.ksztalt_gora, 0)
        pygame.draw.rect(screen, self.kolor, self.ksztalt_dol, 0)
    def ruch(self, v):
        self.x = self.x - v
        self.ksztalt_gora = pygame.Rect(self.x, self.y_gora, self.szerokosc, self.wys_gora)
        self.ksztalt_dol = pygame.Rect(self.x, self.y_dol, self.szerokosc, self.wys_dol)
    def kolizja(self, player):
        if self.ksztalt_gora.colliderect(player) or self.ksztalt_dol.colliderect(player):
            return True
        else:
            return False
```

2 Wewnątrz tej metody sprawdzamy, czy **ksztalt_dol** albo **ksztalt_gora** koliduje z **Rect** z argumentu. Robimy to poleceniem **colliderect()** wywołanym dla każdego z tych dwóch prostokątów. Polecenie to umieszczamy w **if**. Jeśli napisany warunek będzie prawdą, nasza metoda, działając jak funkcja, powinna zwrócić wartość **True**. Jeżeli kolizja nie wystąpi, metoda powinna zwrócić wartość **False**.

3 Z metody tej korzystamy w pętli **while**, po przesunięciu i narysowaniu przeszkód. Wykorzystujemy ją w instrukcji **if**,

```
elif copokazuje == "rozgrywka":
    for p in przeszkody:
        p.ruch(1)
        p.rysuj()
        if p.kolizja(gracz.ksztalt):
            copokazuje = "koniec"
```

umieszczając ją w warunku. Jeśli metoda zwraca **True** – **if**, przejdzie do polecenia, które zostanie napisane, czyli do **copokazuje = "koniec"**. Dzięki temu gra przestanie wyświetlać rozgrywkę. Pojawi się pusty ekran, bo nigdzie nie napisaliśmy, co ma się wyświetlać, gdy **copokazuje** ma taką wartość.

Przełączanie się pomiędzy menu a rozgrywką

Póki co, nasz program wyświetla tylko to, co ręcznie ustawimy w zmiennej **copokazuje**. Z poziomu gry nie możemy przełączać się pomiędzy menu a rozgrywką. Musimy zatem dopisać, aby gracz widzący menu po naciśnięciu spacji, o czym informuje napis, mógł przejść do rozgrywki.

1 W tym celu powinniśmy powrócić do miejsca w kodzie, w którym poprzez pętlę **for** przechodziliśmy po zdarzeniach. Mamy tam już zaprogramowane wyłapywanie zdarzenia mówiącego o naciśnięciu jakiegoś klawisza, a gdy to nastąpi, sprawdzamy dwoma ifami, czy była to strzałka w górę,

czy w dół. Podobnego ifa należy dopisać, aby sprawdzić, czy naciśnięto spację, która w **pygame** oznaczona jest jako **K_SPACE**. Piszemy zatem **if event.key == pygame.K_SPACE:** ●.

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_UP:
        dy = -1
    if event.key == pygame.K_DOWN:
        dy = 1
    if event.key == pygame.K_SPACE:
```

2 Naciśnięcie spacji powinno powodować start gry, gdy widoczne jest menu, bądź jej restart, kiedy **copokazuje** wskazuje na **"koniec"**. Naciśnięcie spacji nie powinno restartować gry, gdy rozgrywka trwa, dlatego gdy opisany wcześniej warunek był spełniony, powinniśmy dodać kolejny **if**, który sprawdzi następny warunek, czyli czy zmienna **copokazuje** ma inną wartość niż **"rozgrywka"**. Zapisujemy to instrukcją **if copokazuje != "rozgrywka":** ●.

```
if event.key == pygame.K_SPACE:
    if copokazuje != "rozgrywka":
```

```
if event.key == pygame.K_SPACE:
    if copokazuje != "rozgrywka":
        gracz = Helikopter(250,275)
```

3 Gdy kolejny warunek również jest spełniony, możemy przejść do restartu gry. Na nowo tworzymy obiekt **gracz** poleceniem **gracz = Helikopter(250,275)** ●, w naswiasie podając współrzędne **x** i **y** lokalizacji, w jakiej ma się pojawić helikopter.

4 Następnie resetujemy do zera wartość zmiennej **dy** ●, co sprawi, że dopóki gracz nie naciśnie strzałki w górę lub w dół, helikopter nie będzie się poruszać.

```
if event.key == pygame.K_SPACE:
    if copokazuje != "rozgrywka":
        gracz = Helikopter(250,275)
        dy = 0
```

5 Jednak poleceniem decydującym o tym, że rozgrywka pojawi się na ekranie, jest zmiana **copokazuje** na **"rozgrywka"** ●.

```
if event.key == pygame.K_SPACE:
    if copokazuje != "rozgrywka":
        gracz = Helikopter(250,275)
        dy = 0
        copokazuje = "rozgrywka"
```

6 Aby w pełni przetestować to, co napisaliśmy, należy też zmienić wartość początkową zmiennej **copokazuje** na **"menu"** ●. Gotowe. Gdy program jest uruchomiony, naciśnięcie spacji, kiedy widoczne jest menu bądź gdy widzimy pusty ekran po przegranej, powinno już powodować przejście do rozgrywki.

```
def napisz(tekst, x, y, rozmiar):
    cz = pygame.font.SysFont("Arial", rozmiar)
    rend = cz.render(tekst, 1, (255,100,100))
    screen.blit(rend, (x,y))

copokazuje = "menu"
```

Ekran przegranej

Po przegranej rozgrywce gracz nie powinien zobaczyć pustego, czarnego ekranu. Tylko co w takim razie należy mu wyświetlić? Najlepiej logo gry i informację, jak ją uruchomić ponownie, czyli coś bardzo podobnego do menu początkowego, jedynie napis nie powinien mówić o rozpoczęciu, ale o ponownej rozgrywce.

1 Do **if**, w którym sprawdzamy wartość zmiennej **copokazuje**, w pętli **while** trzeba dodać nową sekcję **elif**. W niej powinniśmy sprawdzić, czy **copokazuje** ma wartość **"koniec"** **A** (patrz kolejna strona).

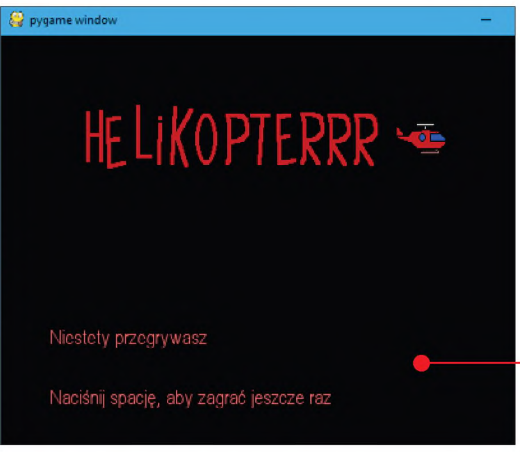
2 Następnie możemy przekopiować linijki odpowiedzialne za wstawianie grafiki

```
if copokazuje == "menu":
    napisz("Naciśnij spację, aby zacząć", 80, 150, 20)
    grafika = pygame.image.load(os.path.join('logo.png'))
    screen.blit(grafika, (80,30))
elif copokazuje == "rozgrywka":
    for p in przeszkody:
        p.ruch(1)
        p.rysuj()
        if p.kolizja(gracz.ksztalt):
            copokazuje = "koniec" A
    for p in przeszkody:
        if p.x <= -p.szerokosc:
            przeszkody.remove(p)
            przeszkody.append((Przeszkoda(szer,szer/20)))
    gracz.rysuj()
    gracz.ruch(dy)
elif copokazuje == "koniec":
```

```
elif copokazuje == "koniec":
    grafika = pygame.image.load(os.path.join('logo.png'))
    screen.blit(grafika, (80,30))
```

```
elif copokazuje == "koniec":
    grafika = pygame.image.load(os.path.join('logo.png'))
    screen.blit(grafika, (80,30))
    napisz("Niestety przegrywasz" , 50, 290,20)
```

logo ●, których użyliśmy we wcześniejszej części **if** – gdy **copokazuje** miało wartość **"menu"**.



3 Kiedy logo jest już umieszczone w kodzie, możemy przejść do wykorzystania procedury **napisz()** i wypisania kilku informacji dla gracza. Pierwsza to taka, że niestety przegrał. Możemy zrobić to poleceniem: **napisz("Niestety przegrywasz", 50, 290, 20)** ●, gdzie liczby **50** i **290** to odpowiednio współrzędna **x** i **y** miejsca, z którego zaczynamy umieszczanie napisu, a **20** to rozmiar czcionki. Oczywiście możemy użyć innych liczb w swoim skrypcie.

4 W innej linijce, poprzez kolejne wywołanie procedury **napisz** ●, możemy umieścić informację o tym, że gracz powinien nacisnąć spację, aby zagrać jeszcze raz.

5 Przetestujmy grę. Teraz, gdy helikopter zderzy się z przeszkodą, powinien być wyświetlony ekran przegranej zawierający odpowiednie informacje ●.

```
elif copokazuje == "koniec":
    grafika = pygame.image.load(os.path.join('logo.png'))
    screen.blit(grafika, (80,30))
    napisz("Niestety przegrywasz" , 50, 290,20)
```

Punkty

Nasza gra działa. Można przeprowadzić rozgrywkę, a po przegranej zagrać ponownie. Jednak jeśli chcielibyśmy porównać wyniki z dwóch przeprowadzonych przez gracza rozgrywek, jest to niemożliwe. Gracz nie otrzymuje żadnych punktów. Musimy to zmienić.

1 Deklarujemy zmienną **punkty**. Możemy to zrobić już w pętli **while**, w miejscu, gdzie nadajemy zmiennej **copokazuje** wartość **"rozgrywka"**. Linijkę niżej piszemy, że zmienna **punkty** ma dostać wartość **0** - dzięki temu po każdym restarcie gry punkty będą się zerowały.

```
if event.key == pygame.K_SPACE:
    if copokazuje != "rozgrywka":
        gracz = Helikopter(250,275)
        dy = 0
        copokazuje = "rozgrywka"
        punkty = 0
```

2 Wartość tej zmiennej powinna rosnąć, ale tylko wtedy, gdy helikopter się porusza. Gdy helikopter nie jest w ruchu - na początku gry - nie powinny być dodawane punkty. Możemy do tego mechanizmu wykorzystać zmienną **dy**. Ma ona wartość **0**, gdy helikopter się nie porusza, a gdy jest w ruchu - wartość **1** lub **-1** (zależnie od strony, w którą leci). W module **math** znajdziemy polecenie do obliczania wartości bezwzględnej z liczby. Wartością bezwzględną z **dy**, gdy helikopter się porusza, zawsze będzie **1**. Zatem możemy zwiększać liczbę punktów o wartość bezwzględną z **dy**. W tym celu musimy zaimportować moduł **math**, aby mieć dostęp do potrzebnego nam polecenia.

```
import pygame
import os
import random
import math
```

3 Kiedy dodawać punkty? Na pewno nie powinno się to dziać za często. Zwiększanie wartości zmiennej **punkty** bezpośrednio w pętli **while** byłoby zbyt częste. Rozsądne wydaje się takie rozwiązanie, że zmienna **punkty** będzie rosła za każdym razem, gdy jakaś przeszkoda opuszcza okno gry, a w jej miejsce tworzona jest nowa. To właśnie tam powinniśmy napisać, że **punkty = punkty + math.fabs(dy)** - **fabs** to funkcja służąca do obliczania wartości bezwzględnej z podanej liczby.

```
for p in przeszkody:
    if p.x <= -p.szerokosc:
        przeszkody.remove(p)
        przeszkody.append((Przeszkoda(szer,szer/20)))
        punkty = punkty + math.fabs(dy)
```

4 Aktualną liczbę punktów uzyskaną przez gracza powinniśmy wypisywać w oknie gry, gdy trwa rozgrywka. Przejdźmy więc do tego miejsca w kodzie, gdzie napisaliśmy, co ma być na ekranie podczas rozgrywki. Trzeba tam dodać jeszcze jedną linijkę kodu: **napisz(str(punkty), 50, 50, 20)**, gdzie korzystając z polecenia **napisz**, umieszczamy na ekranie wartość zmiennej **punkty**, ale tym razem rzutowaną na **String**.

```
elif copokazuje == "rozgrywka":
    for p in przeszkody:
        p.ruch(1)
        p.rysuj()
        if p.kolizja(gracz.ksztalt):
            copokazuje = "koniec"
    for p in przeszkody:
        if p.x <= -p.szerokosc:
            przeszkody.remove(p)
            przeszkody.append((Przeszkoda(szer,szer/20)))
            punkty = punkty + math.fabs(dy)
    gracz.rysuj()
    gracz.ruch(dy)
    napisz(str(punkty), 50, 50,20)
```

```
=================== RESTART: C:\Users\konra\Documents\h.py ===================
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
Traceback (most recent call last):
  File "C:\Users\konra\Documents\h.py", line 101, in <module>
    napisz(punkty, 50, 50,20)
  File "C:\Users\konra\Documents\h.py", line 14, in napisz
    rend = cz.render(tekst, 1, (255,100,100))
TypeError: text must be a unicode or bytes
>>>
```

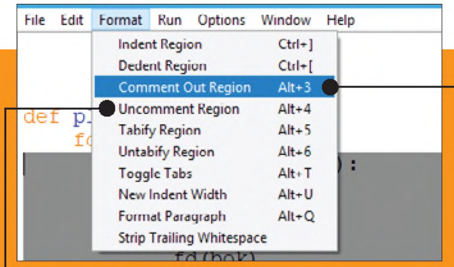
Gdybyśmy nie napisali **str(punkty)**, a tylko **punkty** – program mógłby wyświetlić błąd.
To dlatego, że wewnątrz procedury **napisz** korzystamy z mechanizmów, które pozwalają wyświetlać jedynie ciągi znaków. A zmienna **punkty** ma typ **Integer**, dlatego próbując ją wyświetlić, trzeba przerobić ją na ciąg znaków. Liczba przerobiona na ciąg znaków nie jest rozumiana jako liczba, tylko jako tekst składający się z cyfr.

5 Aby dopełnić nasz skrypt przed ostatecznymi testami, należałoby jeszcze pokazać wynik gracza na ekranie końca gry. To również robimy poleceniem **napisz**, jednak rozszerzamy nieco tekst, dodając przed wartością zmiennej **punkty** napis **"Twój wynik to: "**. Gra jest już gotowa. Możemy przystąpić do intensywnych testów. Jeśli wszystko zostało napisane zgodnie z tym poradnikiem, gra powinna działać, a efekt powinien być taki, jaki zaprezentowano w treści zadania.

```
elif copokazuje == "koniec":
    grafika = pygame.image.load(os.path.join('logo.png'))
    screen.blit(grafika, (80,30))
    napisz("Niestety przegrywasz" , 50, 290,20)
    napisz("Nacisnij spację, aby zagrać jeszcze raz", 50, 350, 20)
    napisz("Twój wynik to: " + str(punkty), 50, 320,20)
```

KOMENTARZE

Komentarze to fragmenty kodu przeznaczone tylko dla programisty, które nie mają wpływu na działanie aplikacji. W Pythonie komentarze poprzedzamy znakiem #. Linijka kodu poprzedzona tym znakiem przyjmuje inny kolor w edytorze. W komentarzach programiści zapisują często informacje, które mogliby zapomnieć, a które są ważne dla dalszej pracy ze skryptem. Czasem w komentarz zamieniane są fragmenty kodu, których działanie chcemy wyłączyć na pewien czas, a potem przywrócić. Edytor IDLE daje możliwość masowego zmieniania fragmentu skryptu w komentarz i usuwania takiego komentarza.



1 Aby zmienić w komentarz zaznaczony fragment skryptu, z menu głównego rozwijamy opcję **Format** i wybieramy **Comment Out Region**.

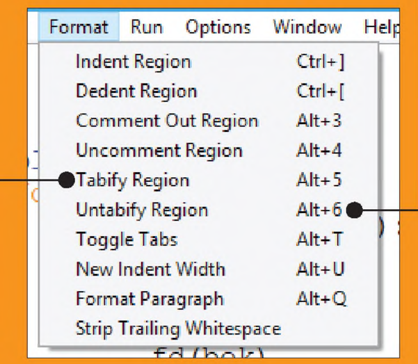
2 By usunąć znak komentarza z zaznaczonego fragmentu kodu, także rozwijamy opcję **Format** z menu głównego, ale tym razem wybieramy **Uncomment Region**.

WCIĘCIA

Wcięcia w Pythonie to coś, na co musimy zwracać szczególną uwagę. „Wnętrza" ifa, pętli, procedur czy wszystkich innych wydzielonych fragmentów kodu pisane są z zachowaniem zasad dotyczących wcięć w kodzie. Czasem zdarza się, że musimy do istniejącego skryptu dopisać fragment wymuszający na nas dodanie wcięcia w napisanych już wcześniej linijkach. Edytor IDLE jest na to przygotowany. Pozwala na masowe dodawanie i usuwanie wcięć w zaznaczonych linijkach kodu.

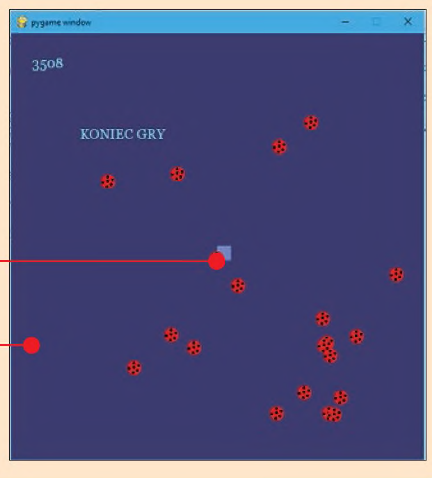
1 Aby dodać wcięcia w zaznaczonym fragmencie skryptu, z menu głównego rozwijamy opcję **Format** i wybieramy **Tabify Region**.

2 By usunąć wcięcia z zaznaczonego fragmentu kodu, także rozwijamy opcję **Format** z menu głównego, ale tym razem wybieramy **Untabify Region**.



Zadanie 1: Inwazja biedronek

Po zapoznaniu się z działaniem **pygame** warto poświęcić trochę czasu na samodzielną pracę. Jest ona dobrym sposobem na szlifowanie wiedzy. Aby poćwiczyć, spróbujmy zrobić grę pod tytułem **Inwazja biedronek**, w której w losowych miejscach pojawiają się biedronki. Powinny one poruszać się z losową prędkością, w losowych kierunkach i móc odbijać się od ścian. Zadaniem gracza niech będzie takie poruszanie kwadratem pomiędzy biedronkami, aby nie zderzyć się z żadną z nich. Efekt powinien być podobny do tego, co widać na rysunku.
Gra ma liczyć punkty – im dłużej gracz unika kolizji z biedronkami, tym więcej powinien mieć punktów. Po kolizji gra powinna się zatrzymać.



PODPOWIEDŹ

Aby uzyskać wiele biedronek, dobrze jest napisać w tym celu klasę, a potem stworzyć jej obiekty (dokładne rozwiązanie zadania znajdziemy w rozdziale **6**).

4 Zadania logiczne

Poznajmy moduł Pythona o nazwie turtle, czyli popularny w szkołach, bardzo ważny dla wszystkich uczniów – żółw. Wskazówki z tego rozdziału pomogą się z nim zaprzyjaźnić

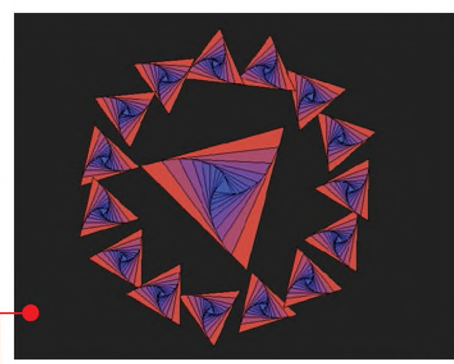
Python jest językiem programowania szeroko wykorzystywanym w edukacji. Duża w tym zasługa jednego z modułów tego języka – **turtle**. Wywodzi się on z języka **Logo** i pozwala na sterowanie wyświetlaną na ekranie grafiką żółwia.

Niegdyś Logo stanowiło podstawę edukacji programistycznej w szkołach, a do dziś w podstawie programowej dla szkół podstawowych znajdują się sformułowania mówiące o tym, że w ramach umiejętności informatycznych uczniowie powinni umieć zaprogramować obiekt tak, aby poruszał się po ekranie w określony przez nich sposób. Umiejętności te są również ważne w wielu konkursach informatycznych. Programistyczne zadania logiczne to doskonały sposób na utrwalenie różnego rodzaju pętli i innych podstawowych instrukcji języka.

W tym rozdziale zapoznamy się z modułem **turtle** na przykładzie gry kółko i krzyżyk, popularnej w wersji papierowej. Stworzymy skrypt, który narysuje planszę tej gry, a także pozwoli na tworzenie przez gracza kółek i krzyżyków w odpowiednich miejscach planszy poprzez definiowane do tego procedury.

Podstawy modułu

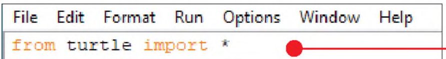
Po uruchomieniu modułu na środku ekranu pojawia się grafika żółwia (symbolizuje go strzałka). Za pomocą podstawowych poleceń – jak **fd()** i **bk()** – przesuwamy grafikę odpowiednio w przód i w tył (o określoną liczbę kroków). Przesuwając się, żółw pozostawia po sobie ślad w postaci kreski. Domyślnie żółw kieruje się w prawą stronę. Zatem użycie polecenia **fd()** przesunie go w tym kierunku. Moduł zawiera też polecenia **left()** i **right()**, dzięki którym możemy obracać obiekt, którym sterujemy, o określoną liczbę stopni – odpowiednio w lewą i prawą stronę.
Do naszej dyspozycji są też polecenia **pu()** i **pd()**. Użycie pierwszego „podnosi pisak", co znaczy, że żółw nie będzie zostawiał za sobą śladów aż do momentu użycia drugiego z poleceń – które „opuszcza pisak". Te podstawowe polecenia są wystarczające, aby rozpocząć pracę z modułem **turtle**, jednak warto pamiętać, że zawiera on jeszcze znacznie więcej ciekawych poleceń i pozwala na tworzenie ciekawie wyglądających projektów (patrz też ramka na stronie 58).



GRA KÓŁKO I KRZYŻYK

Tworzymy planszę

1 Aby móc w ogóle zacząć korzystać z modułu **turtle**, należy go importować. Robimy to poleceniem **from turtle import *** . Dzięki temu będziemy mogli używać każdego polecenia z **turtle** bez podawania dodatkowo nazwy tego modułu, jak miałoby to miejsce, gdybyśmy moduł importowali poprzez polecenie **import turtle**.



2 Zastanówmy się, jak ma być zbudowana plansza naszej gry. W wersji papierowej plansza to dwie pionowe kreski przecinające się z dwiema poziomymi kreskami. W naszej grze może być nieco inaczej: nasza plansza może mieć formę kratki zbudowanej z dziewięciu kwadratów – trzech kolumn po trzy wiersze. W tym celu będziemy potrzebowali procedury, w której zapiszemy sobie rysowanie jednego kwadratu. Potem zapętlimy wykonywanie tej procedury, aby powstało dziewięć kwadratów, które ułożą się w planszę.

UWAGA!
Nie wpisujmy skryptu w konsoli. Lepiej skorzystać z opcji tworzenia nowego dokumentu (jak w poprzednich rozdziałach), w którym zapiszemy cały skrypt. To ułatwi późniejsze wprowadzanie zmian, jeśli zechcemy nieco przebudować swoją grę.

3 Zaczynamy zatem od definicji procedury o nazwie **kwadrat**. Jako figura geometryczna kwadrat jest dość powtarzalny – wszystkie boki i kąty pomiędzy nimi są takie same. Ma cztery takie same boki. Żółw może narysować go, idąc przed siebie o jakąś stałą

liczbę kroków, a następnie skręcać o 90 stopni. Czynność tę należałoby powtórzyć cztery razy. Jak to zapisać? Należy skorzystać z pętli **for**, która wykona się **4** razy.

4 W pętli używamy polecenia **fd()**, w nawiasie, jako parametr wpisując liczbę, która będzie bokiem naszego kwadratu. Następnie używamy polecenia **left(90)**, aby żółw skręcił o 90 stopni w lewą stronę.

```
from turtle import *

bok = 80

def kwadrat():
    for i in range(4):
        fd(bok)
        left(90)
```

5 Ponieważ wielkość boku naszego kwadratu będzie często powtarzała się w kodzie, zapiszmy ją do zmiennej. To dobry nawyk, aby tak robić. Gdybyśmy w przyszłości chcieli zmienić wielkość boku, zmienimy go tylko w jednym miejscu, gdzie tworzymy i nadajemy wartość zmiennej, a nie w każdym miejscu w kodzie, gdzie należy z niej skorzystać. Przed definicją procedury **kwadrat**, tworzymy więc zmienną **bok** i nadajemy jej wartość **80**.

6 Kolejna powinna być definicja procedury **plansza**, w której treści zapętlimy wykonywanie procedury **kwadrat**. W treści tej procedury powinny znaleźć się dwie pętle **for**. Jedna będzie odpowiadała za rysowanie jednego „wiersza" planszy, czyli trzech kwadratów w poziomie. A samo rysowanie „wiersza" powinno być także zapętlone i wykonane trzy razy – aby powstały trzy wiersze kwadratów. Zatem jedna pętla **for** powinna być wewnątrz drugiej pętli.

```
def plansza():
    for i in range (3):
        for j in range (3):
```

7 W wewnętrznej pętli **for** powinniśmy rysować kwadrat, a następnie przesuwać się do takiego miejsca, aby kolejne przejście pętli narysowało kwadrat obok poprzedniego. Definicja procedury **kwadrat** została stworzona tak, że żółw po narysowaniu kwadratu znajduje się w lewym dolnym rogu figury. Jest to miejsce, w którym zaczynał on rysowanie kwadratu. Aby kolejne wywołanie procedury **kwadrat()** skutkowało pojawieniem się figury po prawej stronie tej już istniejącej, żółw powinien znajdować się w prawym dolnym rogu kwadratu. A żeby się tam znalazł, z obecnej pozycji powinien przemieścić się o wielkość boku kwadratu przed siebie. Robimy to poleceniem **fd(bok)**.

```
def plansza():
    for i in range (3):
        for j in range (3):
            kwadrat()
            fd(bok)
```

8 Dobrą praktyką przy korzystaniu z modułu **turtle** jest to, aby pisak żółwia był opuszczony tylko wtedy, gdy poruszamy się, aby coś narysować. A przemieszczając się do pozycji, w której chcemy narysować kolejny kwadrat, mimo że idziemy wciąż po krawędzi poprzedniego kwadratu, nie chcemy nic nowego stworzyć. Powinniśmy zatem podnieść pisak. Zrobimy to w ten sposób, że wewnątrz pętli, przed wywołaniem procedury **kwadrat()**, użyjemy polecenia **pd()**, aby włączyć rysowanie, a zaraz po jej wywołaniu – polecenia **pu()**, żeby rysowanie wyłączyć.

```
def plansza():
    for i in range (3):
        for j in range (3):
            pd()
            kwadrat()
            pu()
            fd(bok)
```

9 W obecnej formie nasza procedura **plansza()** narysowałaby rząd dziewięciu kwadratów obok siebie. Musimy więc ją nieco przebudować. Po narysowaniu trzech kwadratów, czyli po wykonaniu wewnętrznej pętli **for**, należałoby ustawić żółwia w takim miejscu, aby zaczął rysować kolejny „wiersz"

```
Python 3.6.4 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python36-32/k i k.py =
>>> plansza()
```

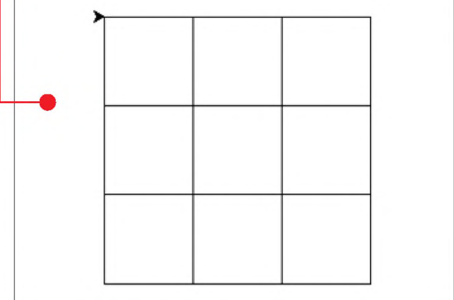
kwadratów. Dobrym miejscem byłby lewy górny róg pierwszego kwadratu z „wiersza". Aby tam dotrzeć, żółw powinien przemieścić się o trzykrotność boku kwadratu do tyłu (**bk(3*bok)**), a następnie o wielkość

```
def plansza():
    for i in range (3):
        for j in range (3):
            pd()
            kwadrat()
            pu()
            fd(bok)
        bk(3*bok)
        left(90) A
        fd(bok) B
        right(90) C
```

jednego boku kwadratu do góry. Ponieważ w tym momencie żółw zwrócony jest w prawą stronę, powinniśmy obrócić go o 90 stopni w lewo **A**, przejść o wielkość boku do przodu **B** i na koniec obrócić się o 90 stopni w prawą stronę **C**, aby kolejny kwadrat narysował się w odpowiednim miejscu (miejsce mamy wyznaczone zgodnie z założeniem, że żółw „patrzy" w prawą stronę).

10 Rozwiązanie należy przetestować. Zapisujemy i uruchamiamy skrypt. Jeśli pojawią się błędy, poprawiamy je. Jeżeli program nie zgłosił błędów, wpisujemy w konsoli **plansza()**, aby wywołać procedurę.

11 Jeśli wszystko przebiegło dobrze, żółw powinien stworzyć prawidłowy rysunek. Pamiętajmy, że wielkość planszy zależy od wielkości boku kwadratu, możemy zmienić wartość nadaną zmiennej bok, jeżeli plansza ma zły rozmiar.



Tworzymy krzyżyk

Tworząc procedurę odpowiedzialną za rysowanie krzyżyka, należy zastanowić się, jak będziemy grać w naszą grę.
Gracze będą na zmianę wywoływać procedurę **krzyzyk** i procedurę **kolko**, które w parametrach powinny mieć możliwość podania kratki, w której ma być postawiony znak. Aby stworzyć taki mechanizm, niezbędne będzie skorzystanie z dwóch poleceń modułu **turtle**, których jeszcze nie używaliśmy. To polecenia **setx()** i **sety()**, pozwalające na przeniesienie żółwia w dowolne miejsce na ekranie poprzez podanie współrzędnych.
Miejsce, w którym rozpoczynamy rysowanie, to punkt **(0,0)**. Współrzędna **x** rośnie w prawą stronę, a współrzędna **y** rośnie w górę (zupełnie jak w matematyce, ale ina-

czej niż w przypadku okna gry tworzonego przez moduł **pygame**).

1 Tworzymy definicję procedury **krzyzyk** w taki sposób, aby przy wywoływaniu należało podać w parametrze dwie liczby – **a** i **b**. Liczba **a** będzie reprezentowała kolumnę, w której stawiamy krzyżyk, a liczba **b** będzie mówiła o wierszu, w którym stawiamy krzyżyk.

```
def krzyzyk(a,b):
```

2 Wewnątrz procedury, zanim przystąpimy do rysowania, powinniśmy przemieszczać żółwia w odpowiednie miejsce. Użyjemy zatem poleceń **setx()** i **sety()**. Aby żółw nie zostawiał śladów podczas przemieszczania się do wybranego przez gracza pola, powinniśmy zacząć od podniesienia pisaka. Opuścimy go dopiero po zmianie współrzędnych żółwia. Zapiszmy zatem zestaw poleceń w procedurze:

```
def krzyzyk(a,b):
    pu()
    setx()
    sety()
    pd()
```

3 Na razie polecenia do zmiany współrzędnych nie mają jeszcze podanych odpowiednich liczb w parametrze. Powinniśmy je uzyskać na podstawie informacji ze zmiennych **a** i **b**, czyli parametrów procedury. Aby móc cokolwiek uzupełnić, ustalmy, jakie wartości może przyjąć **a** i **b**. Rzędy i wiersze powinny być numerowane od dołu do góry i od lewej do prawej strony – zaczynając od zera. Numerację reprezentuje ramka.

0,2	1,2	2,2
0,1	1,1	2,1
0,0	1,0	2,0

4 Współrzędne **x** i **y** żółwia powinniśmy ustawiać takie, aby znalazł się on na środku pola wybranego przez gracza. W przypadku współrzędnej **x** – dla pola oznaczonego zerem – będzie to x równy połowie boku. Dla pola oznaczonego jedynką – to bok plus połowa wielkości boku, a dla pola oznaczonego dwójką – to dwukrotność boku plus połowa boku. Zależność tę można opisać wzorem: **a*bok + bok/2**. I właśnie ten wzór powinniśmy wpisać w nawias przy poleceniu **setx()**.

```
def krzyzyk(a,b):
    pu()
    setx(a*bok + bok/2)
    sety(b*bok + bok/2)
    pd()
```

5 Analogicznie do współrzędnej **x** możemy wyznaczyć współrzędną **y**, z tym że do jej obliczenia we wzorze powinniśmy skorzystać z wartości parametru **b**.

6 Kiedy żółw jest już na środku wybranego pola, możemy przystąpić do rysowania, należy więc opuścić pisak. Aby powstał symbol x, a nie plus, obracamy żółwia o 45 stopni w lewą stronę **A**. Następnie poprzez pętlę **for**, która wykonuje się czterokrotnie **B**, przesuwamy go do przodu o jedną czwartą boku, po czym o taką samą odległość cofamy. Potem wykonujemy obrót o 90 stopni w lewą lub prawą stronę. Aby zakończyć rysowanie w pozycji, w jakiej je zaczęliśmy, na koniec obracamy jeszcze żółwia w prawą stronę o 45 stopni **C** i żeby być w zgodzie z przyjętymi zasadami, podnosimy pisak.

```
def krzyzyk(a,b):
    pu()
    setx(a*bok + bok/2)
    sety(b*bok + bok/2)
    pd()
    left(45) A
    for i in range (4): B
        fd(bok/4)
        bk(bok/4)
        left(90)
    right(45) C
    pu()
```

Tworzymy kółko

Kółko stworzymy podobnie jak krzyżyk. Okrągły kształt, jaki powinniśmy uzyskać, nie będzie jednak idealnym okręgiem. Możemy przyjąć, że kółko to trzydziestosześciobok. Aby go narysować, nie powinniśmy ustawiać żółwia na środku kratki, ale w takim miejscu, aby ten środek okrążyć, rysując figurę.

1 Schemat jest podobny jak w wypadku krzyżyka: podniesienie pisaka, przemieszczenie żółwia i opuszczenie pisaka - to na początek.

```
def kolko(a,b):
    pu()
    setx()
    sety()
    pd()
```

2 Współrzędną **x** możemy obliczyć według tego samego wzoru, co tworząc krzyżyk.

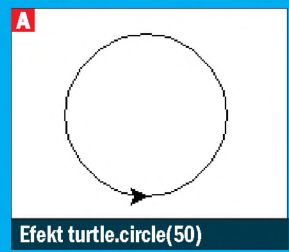
```
def kolko(a,b):
    pu()
    setx(a*bok + bok/2)
    sety()
    pd()
```

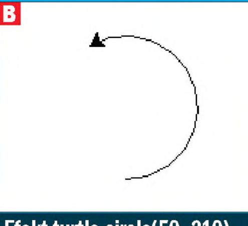
3 Zakładamy, że figura powstanie przez pętlę, to znaczy będziemy przesuwać żółwia do przodu i skręcać w lewą stronę. Współrzędna **y** musi być mniejsza niż na środku wybranego pola. Mniejsza o promień kółka. A promień możemy obliczyć ze wzoru na obwód koła **(2*pi*r)**. Kiedy kółko jest tworzone przez 36 linii długości trzech kroków żółwia, to jego obwód to 108 jednostek (36*3). Po przekształceniu wzoru dochodzimy do tego, że promień naszego kółka to 54/pi. Nie musimy tego sami obliczać poprzez podstawianie przybliżonej wartości liczby pi. Python zna tę liczbę i policzy to za nas. Wzór na współrzędną **y** w przypadku kółka powinien zatem wyglądać tak:
b*bok + bok/2 – 54/math.pi.

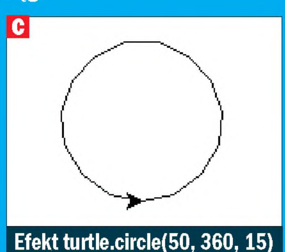
```
def kolko(a,b):
    pu()
    setx(a*bok + bok/2)
    sety(b*bok + bok/2 - 54/math.pi)
    pd()
```

INNY SPOSÓB NA KÓŁKO

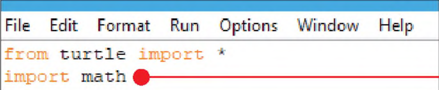
Przedstawione rozwiązanie ma cel edukacyjny. Innym sposobem wytyczenia okręgu w grze jest wykorzystanie polecenia **turtle.circle(x)** A, gdzie argument **x** oznacza promień figury. Polecenie to można wywoływać także z dwoma argumentami: **turtle.circle(x,y)** B. Żółw wytycza wtedy łuk, gdzie **y** oznacza kąt tego łuku. Trzecim sposobem jest wykonanie polecenia z trzema argumentami: **turtle.circle(x,y,z)** C. W takim wypadku **z** oznacza liczbę boków, jaką ma mieć wytyczana figura, ponieważ efekt polecenia **circle** to tak naprawdę (jak w rozwiązaniu przedstawionym we wskazówce powyżej) wielobok, który kształtem przypomina okrąg.


A Efekt turtle.circle(50)


B Efekt turtle.circle(50, 210)


C Efekt turtle.circle(50, 360, 15)

4 Aby Python wiedział, co znaczy **math.pi**, powinniśmy zaimportować moduł **math** – u góry skryptu dodajemy **import math** ●.



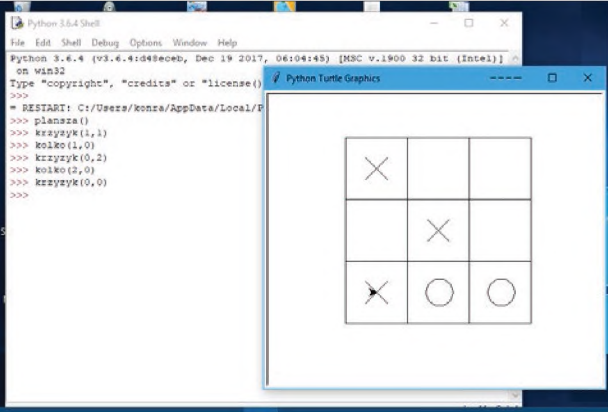

```
def kolko(a,b):
    pu()
    setx(a*bok + bok/2)
    sety(b*bok + bok/2 - 54/math.pi)
    pd()
    for i in range (36):
        fd(3)
        left (10)
```

5 Wróćmy do polecenia **kolko()** – należy stworzyć w nim pętlę **for**, która wykona się 36 razy. W niej żółw przesunie się przed siebie o trzy kroki i skręci w lewo o 10 stopni ●. Po wykonaniu pętli pamiętajmy o tym, aby podnieść pisak **pu ()** ●.

```
    for i in range (36):
        fd(3)
        left (10)
    pu()
```

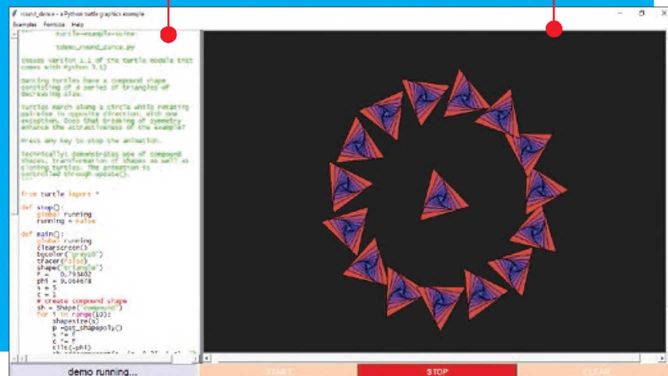
Testowanie

Uruchamiamy teraz skrypt i wpisujemy w konsoli polecenie **plansza()**. Następnie razem z partnerem do gry wpisujemy na zmianę polecenia **kolko()** i **krzyzyk()**. Gra nie jest mocno zautomatyzowana, nie ma mechanizmu sprawdzania wygranej, jednak praktycznie niczym nie różni się od rozgrywki na kartce.



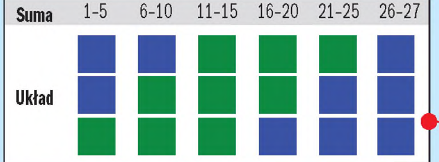
MOŻLIWOŚCI ŻÓŁWIA

Jeśli chcemy poznać więcej możliwości modułu **turtle** i zobaczyć przykładowe programy napisane z jego wykorzystaniem, pomoże nam program IDLE. Z menu głównego rozwijamy pozycję **Help**, a następnie wybieramy opcję **Turtle Demo**. Zobaczymy nowe okno. Z jego menu wybieramy **Examples**, aby wyświetlić listę przykładów. Po wybraniu jednego z nich zobaczymy skrypt ●, a także jego działanie ●.

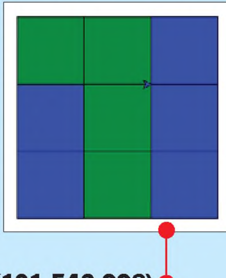


Zadanie 2: Graficzne kodowanie liczb

Zdefiniuj procedurę **zakoduj()**, która w parametrze będzie przyjmowała trzy liczby składające się z trzech cyfr. Zależnie od sumy każdej z tych liczb, powinien powstać inny rysunek. Rezultatem wykonania procedury jest rysunek przedstawiający kratkę 3 na 3 kwadraty. Każda kolumna tej kratki odpowiada kolejnej liczbie z parametru procedury.



Zależnie od sumy cyfr danej liczby z parametru, rysuje się inny układ kwadratów w słupku. Możliwe układy przedstawia tabela.



Przykład:
Wywołanie **koduj(101,543,998)**

PODPOWIEDŹ: Aby rozwiązać zadanie, musimy wiedzieć, jak wypełniać obiekty kolorem. Niezbędne do tego jest poznanie trzech kolejnych poleceń. Pierwsze z nich to **begin_fill()**. Od momentu jego użycia żółw będzie rejestrował przemierzaną przez siebie trasę. Trasa ta powinna wytyczać kształt, który chcemy wypełnić kolorem. Drugie z niezbędnych poleceń to **fillcolor()**, gdzie w nawiasie podajemy nazwę koloru, jakim wytyczony kształt ma zostać wypełniony. Nazwy kolorów wpisujemy w języku angielskim, w cudzysłowie. Ostatnie z nowych poleceń to **end_fill()**. Jego użycie kończy rejestrowanie ścieżki wytyczonej przez żółwia i zamyka kształt – a następnie wypełnia go wcześniej wybranym kolorem.

Zadanie 3: Program sam wstawia znaki

Przebuduj skrypt z zadania opisanego w tym rozdziale w taki sposób, aby gracze nie wywoływali bezpośrednio procedur **kolko()** i **krzyzyk()**, a jedynie procedurę **postaw()** – w której będą określać współrzędne, gdzie ma być postawiony nowy znak, a program sam będzie wstawiał tam zamiennie kółka i krzyżyki.

PODPOWIEDŹ: Dobrym rozwiązaniem byłoby tu użycie dodatkowej zmiennej **czyj_ruch** przechowującej informację o tym, czy należy teraz postawić kółko czy krzyżyk.

Zadanie 4: Blokada zajętego pola

Kontynuuj zmiany w skrypcie – tym razem dodaj do niego mechanizm, który uniemożliwia postawienie znaku w polu, w którym postawiono już inny znak.

PODPOWIEDŹ: Do rozwiązania tego zagadnienia należy wykorzystać tablicę dwuwymiarową.

Dokładne rozwiązania zadań znajdziemy w rozdziale **6**

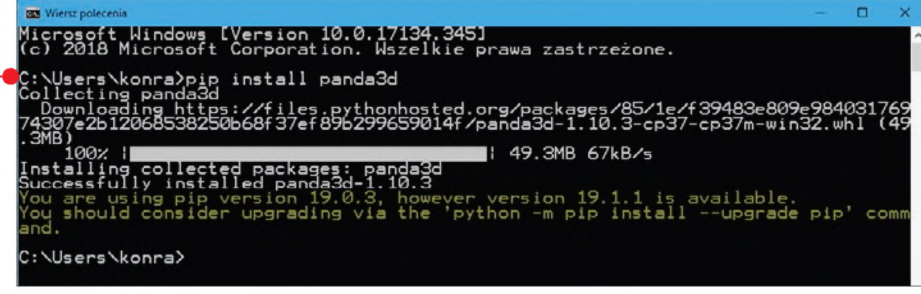
5 Programowanie z grafiką 3D

Poznaliśmy już podstawy Pythona i sposoby korzystania z kilku popularnych modułów tego języka. Osoby chcące związać się z branżą tworzenia gier komputerowych, powinny zainteresować się także możliwością wykorzystania w projektach grafiki 3D. Również i takie moduły są w Pythonie

Poznajmy Panda3D

Jednym z modułów do tworzenia grafiki 3D jest **Panda3D** (**DVD-KOD: 008/009** 32-/64-BIT). To w zasadzie silnik gier, który dostarcza zestaw grafik 3D, ale też wiele innych funkcji wykorzystywanych podczas tworzenia gier, jak detekcja kolizji, fizyka obiektów, obsługa audio, wsparcie rozwiązań sieciowych czy podstawowa sztuczna inteligencja. Warto wiedzieć, że silnik ten jest dziełem jednego z oddziałów wytwórni Disneya, którego zadaniem było tworzenie atrakcji 3D dla parków rozrywki. Z czasem Panda3D zaczęła być wykorzystywana także do tworzenia gier, takich jak na przykład **Toontown Online**, **Pirates of the Caribbean** czy **Signal Ops**.

Aby skorzystać z tego modułu, możemy zainstalować go, wpisując w **cmd** polecenie **pip install panda3d**. Pobrana zostanie wtedy najnowsza wersja modułu. Możemy też zainstalować Panda3D z płyty dołączonej do książki. Na płycie oprócz samego modułu



portal_culling.py - C:\Users\konra\Downloads\panda3d-1.10.3-samples\panda3d-1.10.3\samples\culling\portal_culling.py (3.7.3)

File Edit Format Run Options Window Help

```
# Some config options which can be changed.
ENABLE_PORTALS = True # Set False to disable portal culling and see FPS drop!
DEBUG_PORTALS = False # Set True to see visually which portals are used

# Load PRC data
from panda3d.core import loadPrcFileData
if ENABLE_PORTALS:
    loadPrcFileData('', 'allow-portal-cull true')
    if DEBUG_PORTALS:
        loadPrcFileData('', 'debug-portal-cull true')
loadPrcFileData('', 'window-title Portal Demo')
loadPrcFileData('', 'sync-video false')
loadPrcFileData('', 'show-frame-rate-meter true')
loadPrcFileData('', 'texture-minfilter linear-mipmap-linear')

# Import needed modules
import random
from direct.showbase.ShowBase import ShowBase
from direct.gui.OnscreenText import OnscreenText
from panda3d.core import PerspectiveLens, NodePath, LVector3, LPoint3, \
    TexGenAttrib, TextureStage, TransparencyAttrib, CollisionTraverser, \
    CollisionHandlerQueue, TextNode, CollisionRay, CollisionNode


def add_instructions(pos, msg):
    """Function to put instructions on the screen."""
    return OnscreenText(text=msg, style=1, fg=(1, 1, 1, 1), shadow=(0, 0, 0, 1),
                        parent=base.a2dTopLeft, align=TextNode.ALeft,
                        pos=(0.08, -pos - 0.04), scale=.05)

def add_title(text):
```

Panda3D znajdziemy też zestaw przykładowych projektów – **Panda3D Samples** (DVD-KOD: 010) A, B (patrz poprzednia strona) dostarczanych przez twórców tego silnika, aby jego użytkownicy mogli się z nim zapoznać. Dzięki temu, że są to pliki o rozszerzeniu ***.py**, możemy zarówno prześledzić skrypt, jak i obejrzeć efekty.

Wyświetlenie bazowej sceny

Silniki gier różnią się między sobą. Każdy zawiera specyficzne polecenia. Panda3D to jeden z wielu dostępnych silników. Warto poznać choćby kilka podstawowych poleceń, aby zorientować się, czy to właśnie ten silnik, z którego chcemy korzystać i którego obsługi powinniśmy się nauczyć.
Panda3D dostarcza nam możliwość wczytywania modeli 3D, z których możemy zbudować świat. Zawiera także gotowy model środowiska. Po wczytaniu go już w kilku linijkach możemy zobaczyć, jak może wyglądać baza do stworzenia własnej gry z grafiką 3D.

1 Zaczynamy od importu odpowiedniego modułu. I tu ważna uwaga. Korzystając z **pygame**, musieliśmy importować całe **pygame**. Natomiast Panda3D jest na tyle dużym narzędziem, że tworząc skrypty, możemy importować jedynie część dostępnych mechanizmów. Aby wyświetlić gotowy, przykładowy

```
File Edit Format Run Options Window Help
from direct.showbase.ShowBase import *
```

świat, wystarczy import jednego modułu za pomocą polecenia: **from direct.showbase. ShowBase import *** .

2 W importowanej części skryptu znajduje się definicja klasy **ShowBase**. Jest to klasa bazowa, na podstawie której musimy stworzyć klasę dziedziczącą, służącą do wyświetlenia modeli 3D. Mechanizm dziedziczenia to jeden z filarów programowania obiektowego. Gdy mamy do stworzenia wiele podobnych do siebie klas, możemy założyć klasę bazową, zawierającą definicje tych elementów klasy, które będą się powtarzać. Potem możemy przypisywać tę klasę bazową do tworzonych później klas dziedziczących, a w ich treści znajdą się już elementy, które zostały zdefiniowane w klasie bazowej. Aby do klasy przypisać klasę bazową, po jej nazwie dodajemy nawias, w którym wpisujemy nazwę klasy bazowej, w naszym przykładzie to **ShowBase** .

```
from direct.showbase.ShowBase import *

class MyApp(ShowBase):
```

```
class MyApp(ShowBase):

    def __init__(self):
```

3 W naszej klasie tworzymy konstruktor, czyli metodę **__init__** , w taki sposób, aby przy tworzeniu obiektu tej klasy nie było konieczności podawania żadnych parametrów.

4 Następnie w konstruktorze wywołujemy **konstruktor** obiektu klasy bazowej.

```
def __init__(self):
    ShowBase.__init__(self)
```

5 Przez polecenie **self.scene = self.loader.loadModel(”models/environment”)** wczytujemy model środowiska, który chcemy wyświetlić.

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
```

6 Potem poleceniem **self.scene.reparentTo(self.render)** dodajemy wczytany model do drzewa obiektów, jakie mają być wyświetlone.

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
```

7 Dalej, poprzez polecenia **setScale** i **setPos** , ustalamy odpowiednio wielkość wczytanego obiektu 3D i jego pozycję w oknie.

8 Po takiej definicji klasy wystarczy utworzyć jej obiekt **A** i wywołać dla niego metodę **run** **B**, której nie musimy pisać samodzielnie (dzięki zastosowaniu dziedziczenia).

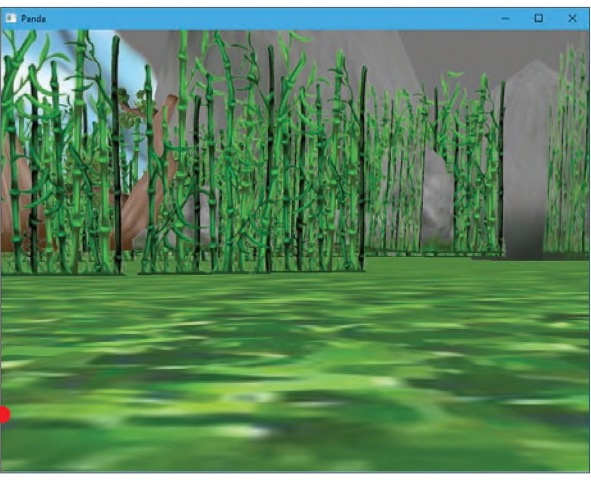
```
from direct.showbase.ShowBase import *

class MyApp(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)
        self.scene = self.loader.loadModel("models/environment")
        self.scene.reparentTo(self.render)
        self.scene.setScale(0.10, 0.10, 0.10)
        self.scene.setPos(-8, 42, 0)


app = MyApp() A
app.run() B
```

9 Zapis takiego skryptu i próba jego uruchomienia powinny skutkować otwarciem okna z modelami 3D wyświetlanymi w nim.

Jeśli chcesz zobaczyć coś więcej niż obiekty aktualnie wyświetlone w oknie gry, nie ma z tym problemu. Przykładowa scena zawiera jeszcze kilka elementów, świat nie jest ograniczony wymiarami okna, może być znacznie większy. Panda3D zawiera mechanizm, który pozwala nam na proste poruszanie się po świecie. Możemy robić to za pomocą myszy:

- wciśnięcie lewego przycisku myszy i ruch kursora przesuwa widok na boki i góra-dół;
- wciśnięcie prawego przycisku myszy i ruch kursora przesuwa widok w przód i tył;
- wciśnięcie rolki myszy i ruch kursora pozwala rotować widok wokół punktu startowego.

Widok wyraźnie się zmienia.



Wstawiamy aktora

W grach 3D aktorami nazywamy obiekty, które umieszcza się na scenie. Panda3D ma zdefiniowaną klasę pozwalającą tworzyć takie obiekty, a także modele 3D, które możemy wykorzystywać. W naszym przykładzie umieścimy na scenie pandę – zgodnie z nazwą silnika.

1 Aby móc korzystać z obiektów klasy **Actor**, trzeba importować odpowiedni moduł. Robimy to, dodając u góry skryptu linijkę: **from direct.actor.Actor import Actor**.

```
File  Edit  Format  Run  Options  Window  Help
from direct.showbase.ShowBase import *
from direct.actor.Actor import Actor
```

2 Dalej przechodzimy do konstruktora stworzonej wcześniej klasy **MyApp**. W nim dodajemy polecenie **self.pandaActor = Actor("models/panda-model",{"walk":**

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
```

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
```

"models/panda-walk4"}), którym tworzymy pole klasy o nazwie **pandaActor**. Jest ono samo w sobie obiektem klasy **Actor**. W parametrze konstruktora podczas tworzenia obiektu odnosimy się do modelu pandy będącego w zasobach silnika i przypisujemy mu animację chodzenia, która jest w zasobach.

3 Poprzez funkcję **setScale**, podobnie jak dla sceny, tak i dla aktora, możemy ustalić rozmiar obiektu. Wszystkie wartości w parametrze wpisujemy z reguły takie same. Odpowiadają one skalowaniu obiektu na każdej z trzech osi w trójwymiarowym układzie współrzędnych. Jeśli na każdej z nich będziemy skalować model tak samo, zachowa on oryginalne proporcje, a co za tym idzie, będzie dobrze wyglądał.

4 Również tak jak w przypadku sceny musimy użyć polecenia **reparentTo**, aby obiekt mógł być renderowany na scenie.

5 Uruchamiamy skrypt i szukamy pandy na scenie – trzeba nieco przesunąć i obrócić widok z kamery

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
    self.pandaActor.reparentTo(self.render)
```

Wprawiamy pandę w ruch

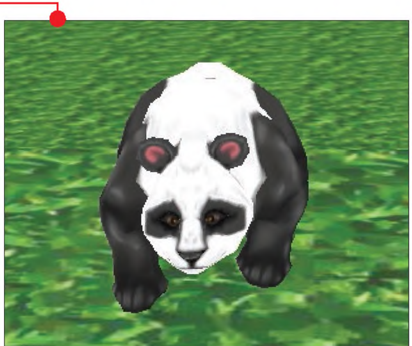
1 Jako wygląd pandy przepisaliśmy jej model wykonujący animację chodzenia. Aby animacja była widoczna, należy ją zapętlić. Wtedy po jednym kroku będą następować kolejne. Robimy to, dodając do konstruktora polecenie **self.pandaActor.loop("walk")**.

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
    self.pandaActor.reparentTo(self.render)
    self.pandaActor.loop("walk")
```

2 Panda wykonuje już animację chodzenia, co widać po tym, jak przebiera nogami, jednak obiekt się nie przemieszcza. Aby móc zacząć pisać jego przemieszczanie, należy ustalić, co panda powinna robić. Możemy napisać jej ruch w taki sposób, aby przeszła z jednego punktu do drugiego, potem zawróciła i przemieściła się z powrotem do punktu, z którego zaczynała iść, po czym kolejny raz się obróciła i w ten sposób osiągnęła lokalizację i rotację, jaką miała na początku. Wtedy powinna powtórzyć sekwencję ruchów, czyli przemieszczenie, obrót, przemieszczenie i obrót.

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
    self.pandaActor.reparentTo(self.render)
    self.pandaActor.loop("walk")
    przemieszczenie1 = self.pandaActor.posInterval(13,Point3(0, -5, 0),startPos=Point3(0, 5, 0))
```

3 Mamy dwa przemieszczenia i dwa obroty, które musimy zdefiniować. Pierwsze przemieszczenie definiujemy poleceniem: **przemieszczenie1 = self.pandaActor.posInterval(13,Point3(0, -5, 0),startPos=Point3(0, 5, 0))**, gdzie:
13 – oznacza czas, w jakim przemieszczenie ma się wykonać,



```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
    self.pandaActor.reparentTo(self.render)
    self.pandaActor.loop("walk")
    przemieszczenie1 = self.pandaActor.posInterval(13,Point3(0, -5, 0),startPos=Point3(0, 5, 0))
    obrot1 = self.pandaActor.hprInterval(3,Point3(-180, 0, 0),startHpr=Point3(0, 0, 0))
```

Point3(0, -5, 0) – to miejsce docelowe, punkt w trójwymiarowym układzie współrzędnych, **startPos=Point3(0, 5, 0)** – oznacza pozycję startową obiektu, z której ma zacząć przemieszczać się do pozycji docelowej.

4 Bardzo podobnie do przemieszczenia programujemy obrót. Robimy to poleceniem: **obrot1 = self.pandaActor.hprInterval(3,Point3(-180, 0, 0),startHpr=Point3(0, 0, 0))**, gdzie:
3 – to czas, w jakim obrót ma się wykonać,
Point3(-180, 0, 0) – wskazuje, jak ma rotować się obiekt na poszczególnych osiach; liczba -180 wskazuje na rotację wokół osi Y (czyli prawo/lewo),
startHpr=Point3(0, 0, 0) – wskazuje, jaka ma być początkowa rotacja, z której będziemy obracać obiekt do tej docelowej.

5 Po wykonaniu pierwszego obrotu powinno nastąpić drugie przemieszczenie. Różni się ono od pierwszego tym, że punktem startowym drugiego jest punkt docelowy pierwszego, a punktem docelowym drugiego – punkt startowy pierwszego przemieszczenia. Piszemy zatem: **przemieszczenie2 = self.pandaActor.posInterval(13,Point3(0, 5, 0),startPos=Point3(0, -5, 0))**.

6 To samo dotyczy obrotu. Tu również, tworząc drugi obrót, rotację docelową i startową zamieniamy miejscami względem pierwszego obrotu. Polecenie powinno wyglądać więc tak: **obrot2 = self.pandaActor.hprInterval(3,Point3(0, 0, 0),startHpr=Point3(-180, 0, 0))**.

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
    self.pandaActor.reparentTo(self.render)
    self.pandaActor.loop("walk")
    przemieszczenie1 = self.pandaActor.posInterval(13,Point3(0, -5, 0),startPos=Point3(0, 5, 0))
    obrot1 = self.pandaActor.hprInterval(3,Point3(-180, 0, 0),startHpr=Point3(0, 0, 0))
    przemieszczenie2 = self.pandaActor.posInterval(13,Point3(0, 5, 0),startPos=Point3(0, -5, 0))
```

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
    self.pandaActor.reparentTo(self.render)
    self.pandaActor.loop("walk")
    przemieszczenie1 = self.pandaActor.posInterval(13,Point3(0, -5, 0),startPos=Point3(0, 5, 0))
    obrot1 = self.pandaActor.hprInterval(3,Point3(-180, 0, 0),startHpr=Point3(0, 0, 0))
    przemieszczenie2 = self.pandaActor.posInterval(13,Point3(0, 5, 0),startPos=Point3(0, -5, 0))
    obrot2 = self.pandaActor.hprInterval(3,Point3(0, 0, 0),startHpr=Point3(-180, 0, 0))
```

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
    self.pandaActor.reparentTo(self.render)
    self.pandaActor.loop("walk")
    przemieszczenie1 = self.pandaActor.posInterval(13,Point3(0, -5, 0),startPos=Point3(0, 5, 0))
    obrot1 = self.pandaActor.hprInterval(3,Point3(-180, 0, 0),startHpr=Point3(0, 0, 0))
    przemieszczenie2 = self.pandaActor.posInterval(13,Point3(0, 5, 0),startPos=Point3(0, -5, 0))
    obrot2 = self.pandaActor.hprInterval(3,Point3(0, 0, 0),startHpr=Point3(-180, 0, 0))
    self.pandaIdzie = Sequence(przemieszczenie1, obrot1, przemieszczenie2, obrot2,name="pandaIdzie")
```

```
def __init__(self):
    ShowBase.__init__(self)
    self.scene = self.loader.loadModel("models/environment")
    self.scene.reparentTo(self.render)
    self.scene.setScale(0.10, 0.10, 0.10)
    self.scene.setPos(-8, 42, 0)
    self.pandaActor = Actor("models/panda-model",{"walk": "models/panda-walk4"})
    self.pandaActor.setScale(0.005, 0.005, 0.005)
    self.pandaActor.reparentTo(self.render)
    self.pandaActor.loop("walk")
    przemieszczenie1 = self.pandaActor.posInterval(13,Point3(0, -5, 0),startPos=Point3(0, 5, 0))
    obrot1 = self.pandaActor.hprInterval(3,Point3(-180, 0, 0),startHpr=Point3(0, 0, 0))
    przemieszczenie2 = self.pandaActor.posInterval(13,Point3(0, 5, 0),startPos=Point3(0, -5, 0))
    obrot2 = self.pandaActor.hprInterval(3,Point3(0, 0, 0),startHpr=Point3(-180, 0, 0))
    self.pandaIdzie = Sequence(przemieszczenie1, obrot1, przemieszczenie2, obrot2,name="pandaIdzie")
    self.pandaIdzie.loop()
```

7 Przemieszczenia i obroty trzeba jeszcze połączyć w sekwencję. Robimy to poleceniem: **self.pandaIdzie = Sequence(przemieszczenie1, obrot1, przemieszczenie2, obrot2,name="pandaIdzie")**.

8 Następnie zapętlamy wykonywanie tej sekwencji.

9 Próba uruchomienia skryptu w tym momencie będzie skutkowała błędem o opisie **name 'Point3' is not defined**. **Point3** jest klasą z silnika. Aby móc korzystać z tego typu obiektów, należy importować odpowiedni moduł. Robimy to, dodając u góry skryptu **from panda3d.core import Point3**.

```
from direct.showbase.ShowBase import *
from direct.actor.Actor import Actor
from panda3d.core import Point3
```

10 Kolejna próba uruchomienia skryptu również będzie skutkowała błędem, który tym razem jest opisany: **name 'Sequence' is not defined**, co oznacza, że połączenie przemieszczeń i obrotów w sekwen-

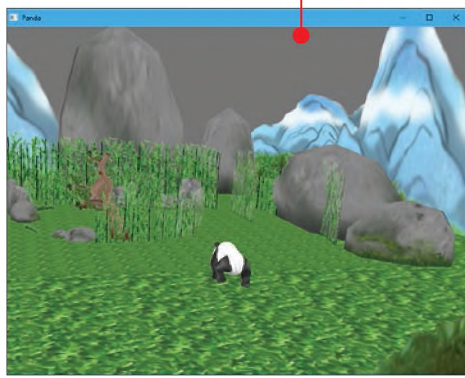
```
Traceback (most recent call last):
  File "C:\Users\konra\Documents\Rozdział 5 - wczytywanie gotowej sceny.py", line 26, in <module>
    app = MyApp()
  File "C:\Users\konra\Documents\Rozdział 5 - wczytywanie gotowej sceny.py", line 19, in __init__
    przemieszczenie1 = self.pandaActor.posInterval(13,Point3(0, -5, 0),startPos=Point3(0, 5, 0))
NameError: name 'Point3' is not defined
>>>
```

```
 RESTART: C:\Users\konra\Documents\Rozdział 5 - wczytywanie gotowej sceny.py
Traceback (most recent call last):
  File "C:\Users\konra\Documents\Rozdział 5 - wczytywanie gotowej sceny.py", line 26, in <module>
    app = MyApp()
  File "C:\Users\konra\Documents\Rozdział 5 - wczytywanie gotowej sceny.py", line 23, in __init__
    self.pandaIdzie = Sequence(przemieszczenie1, obrot1, przemieszczenie2, obrot2,name="pandaIdzie")
NameError: name 'Sequence' is not defined
>>>
```

```
from direct.showbase.ShowBase import *
from direct.actor.Actor import Actor
from panda3d.core import Point3
from direct.interval.IntervalGlobal import Sequence
```

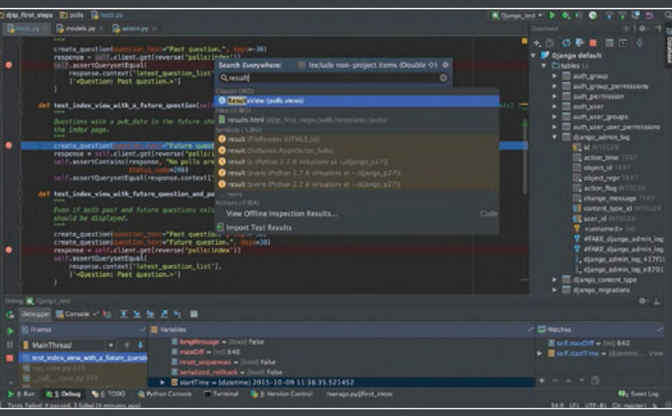
cję, gdy używamy obiektu klasy **Sequence**, jest możliwe tylko po imporcie odpowiedniego modułu. Robimy to poleceniem **from direct.interval.IntervalGlobal import Sequence**.

11 Tym razem próba uruchomienia skryptu powinna skutkować już wyświetleniem poprawnego okna, w którym zobaczymy chodzącą pandę.



PYCHARM – BARDZIEJ ZAAWANSOWANE IDE

Poznajemy stopniowo coraz bardziej zaawansowane polecenia, a nasza znajomość Pythona będzie rosła dalej. IDLE to IDE dobre dla początkujących użytkowników. Kiedy uznamy, że narzędzie to nie jest już dla nas wystarczające, możemy zacząć korzystać z bardziej zaawansowanych. Przykładem takiego bardziej zaawansowanego IDE może być **PyCharm** (DVD-KOD: 012) firmy **JetBrains**. Pozwala on między innymi na edycję i analizę kodu źródłowego, uruchamianie testów jednostkowych. Zapewnia integrację z systemem kontroli wersji. Zawiera graficzny debugger. Godne uwagi jest to, że wspiera programowanie i tworzenie aplikacji internetowych w popularnym frameworku **Django**.



6 Rozwiązania zadań

W tym rozdziale możemy sprawdzić rozwiązania zadań z wcześniejszych części książki. Pamiętajmy jednak – liczy się efekt! Do dobrego rozwiązania można dojść na różne sposoby

Sprawdźmy teraz, czy nasze pomysły na rozwiązania zadań zaprezentowanych w poprzednich rozdziałach są podobne do pomysłów autora tej książki. W przypadku rozwiązywania zadań programistycznych najważniejszy jest efekt. Jeśli udało nam się go osiągnąć – rozwiązanie jest prawidłowe, nawet jeśli inne niż zaproponowane na kolejnych stronach. To co tu znajdziemy, to tylko propozycje i podpowiedzi.

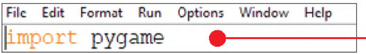
Zadanie 1: Inwazja biedronek

W treści zadania opisana została gra Inwazja biedronek. Zaproponowane rozwiązanie to tylko jedna z możliwości realizacji tego zadania. Jest ich znacznie więcej. Jeśli uda nam się stworzyć grę, która spełnia wymogi zadania, ale napisana jest inaczej niż w przykładzie – tym lepiej!

Import niezbędnych modułów

Znamy założenia gry – już na tym etapie możemy przewidzieć, jakie moduły będą nam potrzebne.

1 Zaczynamy od importu modułu **pygame** – w końcu to podstawowy moduł do budowy gier, bez którego niewiele moglibyśmy zrealizować.



2 Wiemy też, że biedronki mają pojawiać się w losowych miejscach, poruszać w losowym kierunku i z losową prędkością. To oznacza, że będziemy potrzebować losowych (pseudolosowych) liczb. Te uzyskamy, korzystając z poleceń dostępnych w module **random**, który też powinniśmy importować.

```
import pygame
import random
```

3 Ponadto wiemy, że niezbędne będzie wgranie grafiki biedronki. Po stworzeniu gry Helikopterrr wiemy, że grafikę należy umieścić w folderze ze skryptem. Aby to zrobić, należy odnieść się do lokalizacji skryptu, co jest możliwe, kiedy korzystamy z polecenia, które znajduje się w module **os** – też więc trzeba go importować.

```
import pygame
import random
import os
```

Tworzenie okna gry

Po imporcie modułów, a w szególności modułu **pygame**, możemy przejść do utworzenia okna gry.

1 Żeby móc korzystać z okien graficznych i innych poleceń modułu **pygame**, trzeba użyć dla niego polecenia **init**, które poznaliśmy przy okazji realizacji gry Helikopterrr.

```
import pygame
import random
import os

pygame.init()
```

2 Przed utworzeniem okna gry powinniśmy zapisać jego wymiary do zmiennych. Przydadzą nam się później do zaprogramowania odbijania się biedronek od krawędzi okna gry. Tworzymy zmienne **szer_okna** i **wys_okna**, do których przypisujemy preferowane wymiary okna gry.

```
import pygame
import random
import os

pygame.init()

szer_okna = 600
wys_okna = 600
```

3 W następnym kroku używamy polecenia **screen = pygame.display.set_mode((szer_okna, wys_okna))**, które tworzy okno gry o określonych przez nas wymiarach odczytanych przez polecenie ze zmiennych.

```
szer_okna = 600
wys_okna = 600
screen = pygame.display.set_mode((szer_okna, wys_okna))
```

Tworzymy klasę dla przeciwników, czyli biedronek

Podobnie jak w przypadku gry o helikopterze zarówno dla przeciwników, jak i gracza moglibyśmy napisać klasy. Ale jest też możliwe takie rozwiązanie, w którym klasę stworzymy jedynie dla biedronek, a z zaprogramowaniem mechanizmów gracza poradzimy sobie bez tworzenia dodatkowej klasy. Przystępujemy zatem do tworzenia jedynej nowej klasy w naszym projekcie.

1 Nasza klasa będzie nazywać się **Biedronka**.

```
pygame.init()

szer_okna = 600
wys_okna = 600
screen = pygame.display.set_mode((szer_okna, wys_okna))

class Biedronka():
```

2 Definiowanie klasy jak zawsze zaczynamy od utworzenia konstruktora. Możemy od razu zastanowić się, jakie dane powinniśmy podać podczas tworzenia obiektu, czyli jakie parametry powinien mieć konstruktor. Większość właściwości poszczególnych biedronek, które są różne dla każdej z nich, to powinny być wartości losowe (kierunek ruchu, prędkość, pozycja). Nie będziemy ich zatem nadawać poprzez parametry konstruktora, ale będziemy losować dla nich wartości wewnątrz konstruktora. Pozostałe właściwości biedronek mogą być dla nich wspólne. To oznacza, że jedyną wartością, jaką musimy podać w parametrze przy definicji konstruktora, jest **self**, co jest obowiązkiem w każdej metodzie.

```
class Biedronka():
    def __init__(self):
```

3 Do przechowania pozycji obiektu powinniśmy stworzyć w klasie dwa pola – **x** i **y**. Korzystając z polecenia **randint**, znajdującego się w module **random**, nadajemy im wartości losowe (pseudolosowe) z takiego zakresu, aby zmieściły się w oknie gry. Na przykład mogą to być wartości z zakresu od 50 do 550 (kiedy wymiary okna to 600 na 600 pikseli). Zostawiamy wtedy pewien margines przy brzegach ekranu gry. Innym sposobem jest użycie małego zakresu, tak aby biedronki wygenerowały się blisko siebie i dopiero potem rozbiegły po całym oknie.

```
class Biedronka():
    def __init__(self):
        self.x = random.randint(250,500)
        self.y = random.randint(250,500)
```

4 Z losowych właściwości biedronki pozostały nam jeszcze kierunek i prędkość poruszania się. Będą nam do nich potrzebne dwa pola, jednak obydwa będą dotyczyły prędkości. Sam kierunek powinien wynikać właśnie z prędkości. Prędkość rozbijamy na dwie składowe – prędkość na osi X i prędkość na osi Y. W każdym kroku biedronka będzie przesuwać się o pewną odległość na jednej i drugiej osi. Te prędkości będą losowe i mogą przybierać zarówno ujemne, jak i dodatnie wartości (na osi X biedronka może przesuwać się zarówno w prawą, jak i w lewą stronę, a na osi Y zarówno w górę, jak i w dół). Nazwijmy te dwa pola **vx** i **vy** i nadajmy im losowe wartości z przedziału od -4 do 4. Może się zdarzyć, że wypadnie 0, a wtedy na danej osi biedronka nie będzie się poruszać i pozostanie jej tylko ruch w pionie lub poziomie. Gdyby zdarzyło się tak, że wylosują się dwa zera, wtedy biedronka pozostanie nieruchoma.

```
class Biedronka():
    def __init__(self):
        self.x = random.randint(250,500)
        self.y = random.randint(250,500)
        self.vx = random.randint(-4,4)
        self.vy = random.randint(-4,4)
```

5 Kolejną właściwością, czyli polem w klasie **Biedronka**, powinna być grafika biedronki, jaką umieścimy w grze. Korzystamy z polecenia **self.grafika = pygame.image.load(os.path.join('biedronka.png'))**, gdzie **biedronka.png** to nazwa pliku, który przechowujemy w tej samej lokalizacji

```
class Biedronka():
    def __init__(self):
        self.x = random.randint(250,500)
        self.y = random.randint(250,500)
        self.vx = random.randint(-4,4)
        self.vy = random.randint(-4,4)
        self.grafika = pygame.image.load(os.path.join('biedronka.png'))
```

```
class Biedronka():
    def __init__(self):
        self.x = random.randint(250,500)
        self.y = random.randint(250,500)
        self.vx = random.randint(-4,4)
        self.vy = random.randint(-4,4)
        self.grafika = pygame.image.load(os.path.join('biedronka.png'))
        self.wielkosc = 20 A
```

```
class Biedronka():
    def __init__(self):
        self.x = random.randint(250,500)
        self.y = random.randint(250,500)
        self.vx = random.randint(-4,4)
        self.vy = random.randint(-4,4)
        self.grafika = pygame.image.load(os.path.join('biedronka.png'))
        self.wielkosc = 20
    def rysuj(self):
```

co skrypt. Możemy użyć wspomnianego polecenia, a konkretnie zawartego wewnątrz niego: **os.path.join**, ponieważ wcześniej importowaliśmy moduł **os**, którego częścią jest to polecenie.

6 Sprawdź wielkość przygotowanej przez siebie grafiki biedronki. Nie powinna być ona zbyt duża. W naszym przykładzie grafika ma rozmiar 20 na 20 pikseli, a biedronka jest tylko czerwonym kółkiem z czarnymi kropkami. Musimy zapisać rozmiar grafiki do odpowiedniego pola w klasie. Ponieważ we wskazówce korzystamy z kwadratowej grafiki, wystarczy jedno pole, które nazywamy **wielkosc** i nadajemy mu wartość **20** A, bo taka jest szerokość i jednocześnie wysokość pliku graficznego. Deklaracją tego pola kończymy pisanie konstruktora klasy **Biedronka**.

7 Możemy teraz przystąpić do tworzenia pozostałych metod klasy. Będziemy potrzebować ich kilku, w naszym przykładzie stworzymy trzy metody. Pierwsza będzie służyła do rysowania biedronki, czyli umieszczenia pliku graficznego w określonych współrzędnych. Zadaniem drugiej będzie przemieszczanie biedronki, a trzeciej – sprawdzanie, czy biedronka uderzyła w gracza.

8 Tworzymy definicję metody **rysuj**, pamiętając o tym, aby w parametrze tej metody umieścić odniesienie obiektu do samego siebie, czyli **self**.

9 W treści metody potrzebujemy jedynie jednego polecenia, które umieszcza grafikę na ekranie. To polecenie **blit**, które wykorzystywaliśmy w grze o helikopterze zarówno do wstawiania grafik, jak i napisów. Tym razem wpisujemy je w formie

```
def rysuj(self):
    screen.blit(self.grafika, (self.x,self.y))
```

screen.blit(self.grafika, (self.x,self.y)). Oznacza to, że odnosimy się do właściwości obiektu i z nich odczytujemy plik graficzny do wstawienia, a także punkt, w którym wstawiamy grafikę. Należy pamiętać, że **x** mówi o odległości grafiki od lewej krawędzi okna gry, a **y** o odległości grafiki od górnej krawędzi okna gry. Zatem punkt **(x, y)** tworzony przez te dwa pola klasy tożsamy jest z lewym górnym rogiem grafiki, którą wstawiamy. To będzie ważne szczególnie w wypadku tworzenia procedury poruszającej biedronką – przecież ma się ona odbijać od krawędzi okna gry. Do tego niezbędna jest znajomość współrzędnych, na których znajdują się krawędzie grafiki.

10 I właśnie do definicji metody odpowiedzialnej za ruch biedronki powinniśmy teraz przejść. A konkretniej – stworzyć definicję metody o nazwie **ruch**, której jedynym potrzebnym nam parametrem będzie **self**. Wszystkie inne niezbędne dane możemy odczytać z pól klasy, ale też ze zmiennych zadeklarowanych globalnie, poza klasą. Musimy przecież znać wymiary okna, aby wiedzieć, gdzie biedronka ma się odbić i zmienić kierunek ruchu.

```
def rysuj(self):
    screen.blit(self.grafika, (self.x,self.y))
def ruch(self):
```

11 Jednak zanim przejdziemy do odbijania, powinniśmy zająć się przemieszczaniem. Zadeklarowaliśmy wcześniej dwa pola klasy, które łączą w sobie prędkość i kierunek obiektu. Powinniśmy zmieniać wartość pola **x**, dodając do niego wartość pola **vx**. W ten sposób, gdy **vx** będzie dodatnie, to na osi X biedronka przesunie się w prawą stronę, a gdy będzie miało wartość ujemną, to po dodaniu wartość pola **x** się zmniejszy, czyli biedronka na tej osi przesunie się w stronę lewą. Piszemy zatem **self.x = self.x + self.vx**.

```
def ruch(self):
    self.x = self.x + self.vx
```

12 Podobny zabieg powinniśmy wykonać z polami dotyczącymi osi Y. Tym razem piszemy **self.y = self.y + self.vy**,

```
def ruch(self):
    self.x = self.x + self.vx
    self.y = self.y + self.vy
```

co oznacza, że gdy **vy** będzie miało wartość dodatnią, to na osi Y biedronka przesunie się w dół (bo zwiększy się odległość grafiki od górnej krawędzi okna gry), a gdy dodamy do **y** ujemną wartość **vy**, to tak naprawdę wykonamy odejmowanie, czyli zmniejszymy odległość grafiki od górnej krawędzi okna gry.

SPÓJNIKI AND, OR

Warunki w ifach możemy łączyć ze sobą różnymi spójnikami. Jeśli mamy instrukcje, które powinny wykonać się, gdy dwa warunki są spełnione, możemy te warunki połączyć spójnikiem **and** (co po angielsku oznacza i). Wtedy jeden i drugi warunek musi być spełniony, aby „wnętrze ifa" się wykonało. A gdy tylko jeden warunek może być spełniony, aby wykonało się „wnętrze ifa", możemy takie warunki połączyć spójnikiem **or** (czyli albo).

13 Tworząc ify sprawdzające, gdzie jest biedronka i czy należy odbić ją od krawędzi okna gry, skorzystamy ze spójnika **or**. Kiedy biedronka dotrze do ściany, powinna się od niej odbić. W logice naszego programu odbicie się od ściany to zmiana strony, w którą porusza się biedronka. Jeśli **vx** jest dodatnie, biedronka porusza się w prawą stronę. Jeżeli **vx** jest ujemne, biedronka przemieszcza się w stronę lewą. Gdy dotyka krawędzi, czy to lewej, czy prawej, musimy wykonać zmianę strony, w którą się porusza. A to znaczy, że **vx**, które było wcześniej dodatnie, musi stać się ujemne, a jeśli było ujemne, musi stać się dodatnie. Możemy to uzyskać, mnożąc wartość pola **vx** przez -1, co z liczby ujemnej zrobi dodatnią, a z dodatniej ujemną.
Jednak zanim napiszemy odpowiednią instrukcję, wróćmy do warunków. Dla lewej krawędzi musimy sprawdzić, czy lewa krawędź biedronki dotyka lewej krawędzi okna gry. Jeśli **x** biedronki będzie mniejszy lub równy 0, należy odbić się od lewej krawędzi. Jeżeli **x** biedronki powiększony o jej szerokość (czyli wartość pola **wielkosc**) będzie większy od szerokości okna gry, również odbijamy biedronkę. Te dwa warunki w ifie zapisujemy w następujący sposób: **if self.x <= 0 or self.x >= szer_okna - self.wielkosc:**.

```
def ruch(self):
    self.x = self.x + self.vx
    self.y = self.y + self.vy
    if self.x <= 0 or self.x >= szer_okna - self.wielkosc:
```

```python
def ruch(self):
    self.x = self.x + self.vx
    self.y = self.y + self.vy
    if self.x <= 0 or self.x >= szer_okna - self.wielkosc:
        self.vx = self.vx * -1
```

14 Gdy któryś z tych warunków jest spełniony, wykonujemy omówione wcześniej polecenie **self.vx = self.vx * -1**, które zmienia kierunek ruchu na osi X.

15 Analogicznie powinno wyglądać poruszanie i odbijanie się biedronek na osi Y. Tworzymy zatem następnego ifa: **if self.y <= 0 or self.y >= wys_okna - self.wielkosc:**. W pierwszym warunku sprawdza on, czy biedronka dotyka górnej krawędzi okna, a w drugim dolnej.

```python
def ruch(self):
    self.x = self.x + self.vx
    self.y = self.y + self.vy
    if self.x <= 0 or self.x >= szer_okna - self.wielkosc:
        self.vx = self.vx * -1
    if self.y <= 0 or self.y >= wys_okna - self.wielkosc:
```

```python
def ruch(self):
    self.x = self.x + self.vx
    self.y = self.y + self.vy
    if self.x <= 0 or self.x >= szer_okna - self.wielkosc:
        self.vx = self.vx * -1
    if self.y <= 0 or self.y >= wys_okna - self.wielkosc:
        self.vy = self.vy * -1
```

16 Gdy jeden z tych warunków będzie spełniony, powinna wykonać się instrukcja: **self.vy = self.vy * -1**, zmieniająca kierunek ruchu na osi Y. W ten sposób kończymy definicję metody odpowiedzialnej za ruch biedronki.

17 W klasie **Biedronka** pozostaje nam jeszcze jedna metoda do napisania – nazwijmy ją **kolizja**.

```python
def kolizja(self):
```

18 Metoda ta będzie sprawdzać, czy biedronka jest w kolizji z graczem. Inaczej niż w wypadku helikoptera i przeszkód, gdzie mieliśmy dwa obiekty zawierające pola typu **Rect** (czyli obiekt, dla którego korzystaliśmy z polecenia **colliderect**), nasza **Biedronka** nie ma pola typu **Rect**. Nie możemy więc przeprowadzić sprawdzania kolizji w ten sam sposób. Przyjmijmy jednak, że **Rect** pojawi się w skrypcie i będzie reprezentował gracza. Dla takiego obiektu możemy użyć polecenia **collidepoint** – pozwala ono sprawdzić, czy prostokąt koliduje z jednym określonym punktem. Możemy przyjąć, że w naszej grze o biedronkach punkt znajdujący się na samym środku biedronki to punkt kolizji. Aby uznać, że biedronka zderzyła się z graczem, **Rect** reprezentujący gracza powinien być w kolizji ze środkiem biedronki.

```python
def kolizja(self, player):
```

19 Obiekt typu **Rect** będziemy przekazywać do metody poprzez parametr. Nazwijmy ten obiekt **player** i poprzez tę nazwę będziemy odnosić się do niego w definicji metody.

20 Pola **x** i **y** w klasie **Biedronka** odnoszą się do punktu w lewym górnym rogu grafiki. Nas interesuje teraz punkt na środku grafiki. W metodzie powinniśmy obliczyć zatem współrzędne nowego punktu. Środek grafiki na osi X nazwijmy **x_środek**. Jego wartość można obliczyć, dodając do wartości pola **x** połowę szerokości grafiki, co zapisujemy poleceniem: **x_środek = self.x + self.wielkosc/2**.

```python
def kolizja(self, player):
    x_środek = self.x+self.wielkosc/2
```

21 W ten sam sposób powinniśmy obliczyć środek grafiki biedronki na osi Y. Piszemy zatem: **y_środek = self.y + self.wielkosc/2**.

```python
def kolizja(self, player):
    x_środek = self.x+self.wielkosc/2
    y_środek = self.y+self.wielkosc/2
```

22 Ze wspomnianego wcześniej polecenia **collidepoint** korzystamy

```
def kolizja(self, player):
    x_środek = self.x+self.wielkosc/2
    y_środek = self.y+self.wielkosc/2
    if player.collidepoint(x_środek, y_środek):
        czcionka = pygame.font.SysFont("Georgia", 20)
        napis = czcionka.render("KONIEC GRY", 1, (123, 213, 231))
```

```
def kolizja(self, player):
    x_środek = self.x+self.wielkosc/2
    y_środek = self.y+self.wielkosc/2
    if player.collidepoint(x_środek, y_środek):
        czcionka = pygame.font.SysFont("Georgia", 20)
        napis = czcionka.render("KONIEC GRY", 1, (123, 213, 231))
        screen.blit(napis, (100,130))
```

w instrukcji warunkowej **if** w następujący sposób: **if player.collidepoint(x_środek, y_środek):**.

```
def kolizja(self, player):
    x_środek = self.x+self.wielkosc/2
    y_środek = self.y+self.wielkosc/2
    if player.collidepoint(x_środek, y_środek):
```

23 Co powinno być „wnętrzem ifa"? Moglibyśmy skorzystać z polecenia **return** i zwracać informację o kolizji. Możemy też zadziałać inaczej. Jednym z możliwych rozwiązań jest umieszczenie w oknie gry napisu: ***Koniec gry***. W tym celu musimy mieć obiekt będący czcionką, który tworzymy poleceniem **czcionka = pygame.font.SysFont("Georgia", 20)**.

```
def kolizja(self, player):
    x_środek = self.x+self.wielkosc/2
    y_środek = self.y+self.wielkosc/2
    if player.collidepoint(x_środek, y_środek):
        czcionka = pygame.font.SysFont("Georgia", 20)
```

24 Za pomocą stworzonego obiektu i jego funkcji **render** tworzymy graficzną reprezentację napisu.

25 Następnie umieszczamy ją w oknie gry poprzez wielokrotnie wykorzystywane już polecenie **blit**.

26 W tej propozycji rozwiązania zadania nie jest to jeszcze koniec metody. Zakładamy bowiem, że ruch biedronek (i jakichkolwiek innych elementów w oknie gry) będzie odbywał się tylko wtedy, gdy wcześniej nie wystąpiła kolizja. A to znaczy, że z poziomu tej metody powinniśmy też móc zatrzymać wszystko, co dzieje się w oknie gry. W tym celu należy stworzyć zmienną, niech nazywa się **gramy**. Początkowo powinna ona mieć wartość **True**, a metoda **kolizja** powinna zmienić tę wartość na **False**. Aby metoda mogła modyfikować wartość takiej zmiennej, powinna być ona zapisana w niej jako globalna.

```
def kolizja(self, player):
    x_środek = self.x+self.wielkosc/2
    y_środek = self.y+self.wielkosc/2
    if player.collidepoint(x_środek, y_środek):
        czcionka = pygame.font.SysFont("Georgia", 20)
        napis = czcionka.render("KONIEC GRY", 1, (123, 213, 231))
        screen.blit(napis, (100,130))
        global gramy
```

27 Dopiero potem możemy takiej globalnej zmiennej nadać wartość **False**.

```
def kolizja(self, player):
    x_środek = self.x+self.wielkosc/2
    y_środek = self.y+self.wielkosc/2
    if player.collidepoint(x_środek, y_środek):
        czcionka = pygame.font.SysFont("Georgia", 20)
        napis = czcionka.render("KONIEC GRY", 1, (123, 213, 231))
        screen.blit(napis, (100,130))
        global gramy
        gramy = False
```

W ten sposób zdefiniowaliśmy całą klasę **Biedronka**.

Tworzymy obiekty klasy Biedronka

Gdy klasa jest już w całości stworzona, możemy przejść do inicjalizowania jej obiektów. Ponieważ biedronek będzie całkiem sporo, a my będziemy musieli odnosić się do każdej z nich, by je przemieszczać, rysować i sprawdzać, czy kolidują z graczem, powinniśmy „trzymać" obiekty na liście.

1 Tworzymy zatem listę **przeciwnicy**.

```
przeciwnicy = []
```

2 Następnie będziemy musieli dodawać kolejne obiekty do tej listy. Tworzymy pętlę **for** – niech wykona się ona tyle razy, ile biedronek chcemy mieć w naszej grze, na przykład 5 czy 10.

```
przeciwnicy = []

for i in range (5):
```

3 Wewnątrz pętli **for** poprzez polecenie **append** dodajemy kolejne elementy do listy. Nasze elementy to obiekty typu **Biedronka**, które tworzymy, wywołując konstruktor, a ten nie wymaga od nas podawania żadnych parametrów.

```
przeciwnicy = []

for i in range (5):
    przeciwnicy.append(Biedronka())
```

Pętla główna gry

Aby sprawdzić działanie utworzonych przez nas obiektów, powinniśmy zacząć już pisanie pętli głównej gry. Podobnie jak w grze o helikopterze będzie to pętla **while**, która ma wykonywać się zawsze.

```
while True:
```

1 Zanim przejdziemy do wywoływania ruchu i rysowania biedronek, możemy zająć się innym ważnym elementem pętli głównej gry. Chodzi o przycisk zamykający okno, który standardowo w oknie gry **pygame** nie ma przypisanego działania. Podobnie jak w przypadku gry z helikopterem dodajemy pętlę **for**, która przechodzi po wszystkich występujących w grze zdarzeniach.

```
while True:
    for event in pygame.event.get():
```

2 Następnie sprawdzamy, czy każde kolejne zdarzenie to naciśnięcie przycisku **x** w prawym górnym rogu okna gry. Robimy to poleceniem **if event.type == pygame.QUIT:**.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```

3 Gdy opisany wcześniej warunek jest spełniony, wyłączamy grę: **pygame** (**pygame.quit()**) i wyłączamy **IDLE(quit()**).

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
```

```
if event.type == pygame.QUIT:
    pygame.quit()
    quit()
```

4 Dalej w pętli głównej tworzymy instrukcję **if**, która będzie sprawdzać wartość globalnej zmiennej **gramy**. Warunek powinien być spełniony, tylko gdy będzie ona miała wartość **True**. Na razie zmienna

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    if gramy == True:
```

ta jeszcze nie istnieje, dlatego przed pętlą główną gry należy ją utworzyć z wartością początkową **True**.

```
gramy = True

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
```

5 Wykorzystując doświadczenia z projektu z helikopterem, wiemy, że przed wyryso-

waniem obiektów należy ukryć to, co narysowaliśmy w oknie gry wcześniej. Najprostsze jest wypełnienie całego okna gry jednolitym kolorem, co robimy poleceniem **screen.fill((50,50,100))**, gdzie w nawiasie podajemy kolor wypełnienia (w systemie RGB).

```
if gramy == True:
    screen.fill((50,50,100))
```

6 W kolejnym kroku tworzymy pętlę **for**, która przechodzi po wszystkich elementach listy **przeciwnicy**.

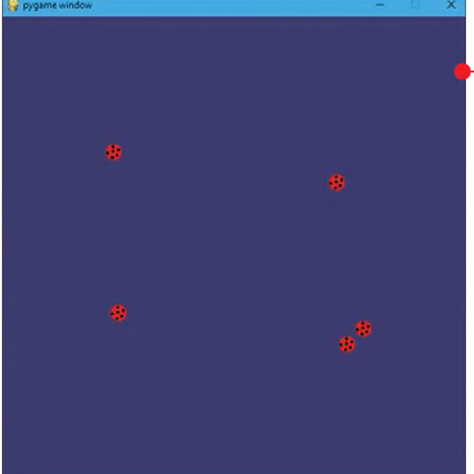
```
if gramy == True:
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
```

```
if gramy == True:
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
```

7 Dla każdego obiektu wywołujemy **ruch**, a także **rysowanie**.

```
if gramy == True:
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
```

8 Na końcu dodajemy jeszcze odświeżanie ekranu, aby móc zobaczyć, jak dodane przez nas zmiany wpłynęły na obraz w oknie gry. Służy do tego polecenie **pygame.display.update()**.




```
if gramy == True:
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
    pygame.display.update()
```

Po uruchomieniu gry zobaczymy jej okno z dość szybko poruszającymi się biedronkami. Aby poruszały się wolniej, pomiędzy kolejnymi wywołaniami ich ruchu należałoby na chwilę zatrzymywać program, na przykład na 10 milisekund, co możemy zrobić poleceniem **pygame.time.wait(10)**.

```
if gramy == True:
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
    pygame.display.update()
    pygame.time.wait(10)
```

Gracz

Dla gracza nie będziemy definiować oddzielnej klasy. Wszystko, co go dotyczy, zapiszemy bezpośrednio.

1 Zaczynamy od zapisania lokalizacji, w której narysujemy gracza. Tworzymy zatem dwie zmienne **x_gracz** i **y_gracz**

```
x_gracz = 300
y_gracz = 300
```

– możemy im nadać wartości początkowe po **300**. Wyznaczymy w ten sposób miejsce, w którym zacznie poruszać się gracz. Zmienne te powinniśmy zadeklarować po utworzeniu klasy **Biedronka**, przed pętlą **while** (pętlą główną gry).

2 Dodajemy też zmienną **v**. Posłużymy się nią, programując ruch gracza. Będziemy dodawać lub odejmować jej wartość od współrzędnych gracza z każdym wykonywanym przez niego krokiem. W ten sposób, gdy w fazie testów stwierdzimy, że gracz poru-

```
x_gracz = 300
y_gracz = 300
v = 20
```

sza się zbyt wolno lub zbyt szybko, bardzo łatwo zmienimy jego prędkość, a w zasadzie odległość pokonywaną w jednym kroku.

3 Dla gracza powinniśmy też utworzyć obiekt typu **Rect**, co robimy poleceniem **gracz = pygame.Rect(x_gracz, y_gracz, 20, 20)**, gdzie dwie liczby **20** oznaczają szerokość i wysokość prostokąta reprezentującego gracza.

```
x_gracz = 300
y_gracz = 300
v = 20

gracz = pygame.Rect(x_gracz, y_gracz, 20, 20)
```

4 Mając obiekt typu **Rect**, możemy już dopisać wywołanie sprawdzania kolizji biedronek z graczem.

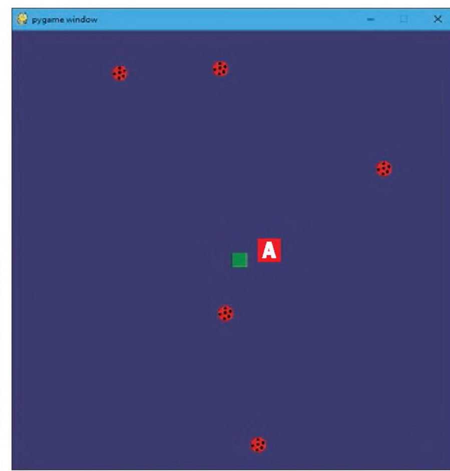
```
if gramy == True:
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
        biedroneczka.kolizja(gracz)
```

5 Możemy też użyć polecenia **pygame.draw.rect(screen, (0,130,0), gracz, 0)**, aby umieścić w oknie reprezentację graficzną stworzonego wcześniej prostokąta.

```
if gramy == True:
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
        biedroneczka.kolizja(gracz)
    pygame.draw.rect(screen, (0,130,0), gracz, 0)
    pygame.display.update()
    pygame.time.wait(10)
```

6 Już niewiele zostało nam do zrobienia. Gracz pojawia się w oknie gry **A**, ale nie może się jeszcze poruszać. Do pętli przechodzącej po zdarzeniach musimy dodać jeszcze mechanizm poszukiwania zdarzenia **pygame.KEYDOWN**, co robimy poleceniem **if event.type == pygame.KEYDOWN:**.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
        if event.type == pygame.KEYDOWN:
```



7 Gdy warunek opisany w ten sposób będzie spełniony, trzeba wykonywać kolejne sprawdzanie – który z klawiszy został wciśnięty. Poleceniem **if event.key == pygame.K_UP:** sprawdzamy, czy klawiszem tym była strzałka w górę.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        pygame.quit()
        quit()
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_UP:
```

8 Jeśli opisany warunek był prawdą, powinniśmy sprawdzić, czy możemy przesunąć gracza w górę, to znaczy, czy jego górna krawędź mieści się w oknie gry. Robimy to za pomocą polecenia **if y_gracz - v > 0:**.

```
if event.type == pygame.QUIT:
    pygame.quit()
    quit()
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_UP:
        if y_gracz - v > 0:
```

9 Jeśli opisany wcześniej warunek był prawdą, zmieniamy wartość zmiennej **y_gracz**, pomniejszając ją o wartość zmiennej **v**.

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_UP:
        if y_gracz - v > 0:
            y_gracz = y_gracz - v
```

10 Podobnie postępujemy w przypadku strzałki w dół. Jednak jeśli została ona naciśnięta, musimy sprawdzić, czy dolna krawędź prostokąta reprezentującego gracza znajduje się w oknie gry. Jeżeli tak, można powiększyć współrzędną **y** gracza o wartość zmiennej **v**. Dzięki temu gracz przesunie się w dół.

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_UP:
        if y_gracz - v > 0:
            y_gracz = y_gracz - v
    if event.key == pygame.K_DOWN:
        if y_gracz + v < wys_okna - 20:
            y_gracz = y_gracz + v
```

11 Następnie poleceniem **if event.key == pygame.K_RIGHT:** sprawdzamy, czy naciśnięto strzałkę w prawą stronę.

```
if event.key == pygame.K_UP:
    if y_gracz - v > 0:
        y_gracz = y_gracz - v
if event.key == pygame.K_DOWN:
    if y_gracz + v < wys_okna - 20:
        y_gracz = y_gracz + v
if event.key == pygame.K_RIGHT:
```

12 Jeśli warunek był spełniony, sprawdzamy, czy prawa krawędź prostokąta reprezentującego gracza mieści się w oknie gry. Jeśli tak, możemy zwiększyć

```
if event.key == pygame.K_RIGHT:
    if x_gracz + v < szer_okna - 20:
```

wartość zmiennej **x_gracz** o wartość zmiennej **v**, co przesunie gracza w stronę prawą.

```
if event.key == pygame.K_RIGHT:
    if x_gracz + v < szer_okna - 20:
        x_gracz = x_gracz + v
```

13 Te same mechanizmy musimy zastosować, sprawdzając, czy naciśnięto strzałkę w stronę lewą, następnie, czy lewa krawędź prostokąta mieści się w oknie gry, a potem dopiero przesuwamy prostokąt w stronę lewą poprzez pomniejszenie wartości zmiennej **x_gracz** o wartość zmiennej **v**, co zmniejsza odległość prostokąta od lewej krawędzi okna, a tym samym przesuwa prostokąt w stronę lewą.

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_UP:
        if y_gracz - v > 0:
            y_gracz = y_gracz - v
    if event.key == pygame.K_DOWN:
        if y_gracz + v < wys_okna - 20:
            y_gracz = y_gracz + v
    if event.key == pygame.K_RIGHT:
        if x_gracz + v < szer_okna - 20:
            x_gracz = x_gracz + v
    if event.key == pygame.K_LEFT:
        if x_gracz - v > 0:
            x_gracz =  x_gracz - v
```

14 Jeśli naciśnięto jakiś klawisz, niezależnie od tego, jaki był to klawisz, mogła nastąpić zmiana współrzędnych prostokąta. Aby przesunęła się też jego reprezentacja graficzna i obszar kolizji, powinniśmy na nowo utworzyć obiekt typu **Rect** o tej samej nazwie co wcześniej (**gracz**) ze zaktualizowanymi współrzędnymi.

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_UP:
        if y_gracz - v > 0:
            y_gracz = y_gracz - v
    if event.key == pygame.K_DOWN:
        if y_gracz + v < wys_okna - 20:
            y_gracz = y_gracz + v
    if event.key == pygame.K_RIGHT:
        if x_gracz + v < szer_okna - 20:
            x_gracz = x_gracz + v
    if event.key == pygame.K_LEFT:
        if x_gracz - v > 0:
            x_gracz =  x_gracz - v
    gracz = pygame.Rect(x_gracz, y_gracz, 20, 20)
```

Punkty

Gracz może się już poruszać, jednak aby gra miała większy sens, powinna być punktowana.

1 Przed pętlą główną gry należy utworzyć zmienną **punkty** o wartości początkowej 0.

```
punkty = 0

gramy = True

while True:
```

2 Z każdym przejściem pętli głównej w naszym ifie, który sprawdza wartość zmiennej **gramy**, powinniśmy zwiększać o jeden (lub o większą liczbę, jeśli chcemy, aby punk-

ty można było zdobywać szybciej) wartość zmiennej **punkty**.

```
if gramy == True:
    punkty = punkty + 1
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
        biedroneczka.kolizja(gracz)
    pygame.draw.rect(screen, (0,130,0), gracz, 0)
    pygame.display.update()
    pygame.time.wait(10)
```

3 Gracz powinien dowiedzieć się, ile punktów zdobył. Skorzystamy z tego samego zestawu poleceń, co w przypadku tworzenia napisu mówiącego o końcu gry. Zaczynamy od utworzenia czcionki.

```
if gramy == True:
    punkty = punkty + 1
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
        biedroneczka.kolizja(gracz)
    czcionka = pygame.font.SysFont("Georgia", 20)
    pygame.draw.rect(screen, (0,130,0), gracz, 0)
    pygame.display.update()
    pygame.time.wait(10)
```

4 Dla czcionki używamy polecenia **render**, które wytwarza graficzną reprezentację tekstu. Tym razem piszemy: **napis = czcionka.render (str(punkty), 1, (123, 213, 231))**, gdzie **str(punkty)** przerabia wartość zmiennej **punkty** na ciąg znaków – robiliśmy to też w grze o helikopterze.

```
if gramy == True:
    punkty = punkty + 1
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
        biedroneczka.kolizja(gracz)
    czcionka = pygame.font.SysFont("Georgia", 20)
    napis = czcionka.render(str(punkty), 1, (123, 213, 231))
```

5 Ostatnim poleceniem jest umieszczenie tej reprezentacji graficznej tekstu z liczbą punktów w oknie gry poprzez polecenie **blit** użyte w formie **screen.blit(napis, (30,30))**.

Teraz nasza gra działa i spełnia wszystkie wymogi zadania. Przeprowadźmy testy. Możemy też spróbować ją rozbudować, na przykład dodając do niej menu startowe na wzór gry o helikopterze.

```
if gramy == True:
    punkty = punkty + 1
    screen.fill((50,50,100))
    for biedroneczka in przeciwnicy:
        biedroneczka.ruch()
        biedroneczka.rysuj()
        biedroneczka.kolizja(gracz)
    czcionka = pygame.font.SysFont("Georgia", 20)
    napis = czcionka.render(str(punkty), 1, (123, 213, 231))
    screen.blit(napis, (30,30))
    pygame.draw.rect(screen, (0,130,0), gracz, 0)
    pygame.display.update()
    pygame.time.wait(10)
```

Zadanie 2: Graficzne kodowanie liczb

Podstawą realizacji tego zadania jest rozbicie go na etapy. Do realizacji każdego z etapów stworzymy oddzielną procedurę. To ułatwi zrozumienie zagadnienia i uporządkuje tworzony kod.
Zastanówmy się teraz, jak podzielić zadanie na etapy. Najlepszy sposób to zwykle wydzielenie powtarzających się elementów. Cała kratka zbudowana jest ze słupków. Elementem, który się powtarza, jest więc słupek. Słupek zbudowany jest z kwadratów, a to jest kolejny powtarzalny element. Wiemy już zatem, że potrzebujemy przynajmniej dwóch oddzielnych procedur – do rysowania kwadratów i do rysowania całych słupków.

Rysowanie kwadratów

1 Realizując zadanie polegające na napisaniu gry Kółko i krzyżyk w rozdziale **4**, również należało stworzyć procedurę o nazwie **kwadrat()**. Przypomnijmy sobie ten etap zadania i napiszmy teraz taką samą procedurę.

```
def kwadrat():
    for i in range(4):
        fd(bok)
        left(90)
```

2 Jednak nie możemy zostawić jej w takiej formie. Przede wszystkim jeśli w skrypcie zostawimy tylko taką procedurę, to Python zwróci nam błędy. Dlaczego? Dlatego, że nie wie jeszcze, co znaczą polecenia **fd()** i **left()**. Aby się dowiedział, należy na początku skryptu zaimportować moduł **turtle**.

```
*zadanie.py - C:\Users\clientadmin\AppData\Local\Programs\Py
File Edit Format Run Options Window Help
from turtle import *

def kwadrat():
    for i in range(4):
        fd(bok)
        left(90)
```

3 Po tych zmianach jest już lepiej, ale Python nadal czegoś nie rozumie. Używamy zmiennej **bok**, mówiącej o rozmiarze kwadratu – natomiast nie utworzyliśmy jej. Należy ją stworzyć i nadać jej wartość.

```
*zadanie.py - C:\Users\clientadmin\AppData\Local\Programs\Py
File Edit Format Run Options Window Help
from turtle import *

bok = 80

def kwadrat():
    for i in range(4):
        fd(bok)
        left(90)
```

4 Teraz Python nie powinien mieć już zastrzeżeń co do poprawności kodu. Jednak nie wszystko w procedurze **kwadrat()** jest zgodne z założeniami zadania. My potrzebujemy kwadratów wypełnionych kolorem. W podpowiedzi do zadania opisane zostały trzy polecenia, które są niezbędne do realizacji tego zagadnienia. Aby wypełnić kwadrat kolorem, należy przed rozpoczęciem wytyczania kształtu przez żółwia użyć polecenia **begin_fill()**.

```
def kwadrat():
    begin_fill()
    for i in range(4):
        fd(bok)
        left(90)
```

5 Aby kształt został wypełniony po jego wytyczeniu, używamy polecenia **end_fill()**.

```
def kwadrat():
    begin_fill()
    for i in range(4):
        fd(bok)
        left(90)
    end_fill()
```

6 Teraz wywołanie procedury **kwadrat** faktycznie skutkuje stworzeniem kwadratu wypełnionego kolorem, jednak jest to kolor czarny. Należy więc jeszcze zmienić kolor wypełnienia. Zgodnie z treścią zadania kwadraty mogą mieć dwa kolory – niebieski i zielony. Najlepiej by było, gdyby jedna pro-

cedura **kwadrat()** mogła tworzyć figury wypełnione tymi dwoma kolorami. W takich sytuacjach wprowadza się do procedury parametr – w nawiasie przy definicji nazwy procedury wpisujemy jego nazwę: **kolor**.

```
def kwadrat(kolor):
    begin_fill()
    for i in range(4):
        fd(bok)
        left(90)
    end_fill()
```

7 Teraz w treści procedury możemy korzystać z wyrażenia **kolor** jak ze zmiennej przechowującej w sobie nazwę koloru. Aby zmienić kolor wypełnienia, należy użyć polecenia **fillcolor()**, w nawiasie wpisując przekazany przez parametr kolor: **fillcolor(kolor)**.

```
def kwadrat(kolor):
    begin_fill()
    fillcolor(kolor)
    for i in range(4):
        fd(bok)
        left(90)
    end_fill()
```

W ten sposób rysowanie kwadratów jest już zakończone. Jak teraz je przetestować? Uruchom stworzony skrypt, a następnie w konsoli wpisz **kwadrat("green")** lub **kwadrat("blue")**. Program powinien stworzyć wtedy rysunek kwadratu wypełnionego odpowiednio kolorem zielonym **A** lub niebieskim **B**.

Rysujemy słupek

Drugim wydzielonym z zadania elementem jest słupek. Składa się on z trzech narysowanych razem kwadratów. Procedurę rysującą słupek powinniśmy zdefiniować w taki sposób, aby móc tworzyć słupki o dowolnej kombinacji kolorystycznej. Inaczej mówiąc, cel jest taki, żeby jedną procedurą zrealizować rysowanie każdego możliwego w zadaniu słupka.

1 Osiągniemy to, kiedy nasza procedura **slupek()** będzie w parametrze przyjmo-

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Users\clientadmin\AppData\Local\Programs\Python\Python37-32\zadanie
.py
27.0
>>> kwadrat("green")
>>>
```

A

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (In
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Users\clientadmin\AppData\Local\Programs\Python\Python37-32\zadan
.py
27.0
>>> kwadrat("blue")
>>>
```

B

wała trzy nazwy kolorów odpowiadających trzem kolejnym kolorom kwadratów. Nazwijmy je **k1**, **k2** i **k3**. Te kolory trafią potem do parametru procedury **kwadrat()** podczas jej wywołań.

```
def slupek(k1,k2,k3):
```

2 Tworząc rysunek słupka, warto mieć na uwadze to, żeby nie przemieszczać żółwia z opuszczonym pisakiem. Dlatego przyjmiemy zasadę, że przed użyciem polecenia **kwadrat()** przykładamy pisak, a zaraz po jego użyciu podnosimy go. Zrobimy to odpowiednio poleceniami **pd()** i **pu()**. Można

```
def slupek(k1,k2,k3):
    pd()
    kwadrat()
    pu()
```

to od razu zapisać w treści procedury. Pamiętajmy też o tym, aby wywołując procedurę **kwadrat()**, podawać w jej parametrze kolor. Można to zrobić, używając koloru zapisanego jako **k1**, czyli pierwszego podanego w parametrze podczas wywoływania rysowania słupka.

```
def slupek(k1,k2,k3):
    pd()
    kwadrat(k1)
    pu()
```

3 Na razie rysowanie słupka ogranicza się do narysowania jednego kwadratu. Kolejnym krokiem powinno być przemieszczenie żółwia i ponowne wywołanie rysowania kwadratu, tym razem już w innym kolorze (**k2**). Obecnie żółw kończy rysowanie w lewym dolnym rogu kwadratu. Aby kolejne wywołanie procedury **kwadrat()** dało nam rysunek drugiego kwadratu ponad tym kwadratem, który już mamy, żółw musiałby powędrować w lewy górny róg figury. Żółw zwrócony jest w prawą stronę. Aby przemieścić go w miejsce docelowe, musimy obrócić go o 90 stopni w lewą stronę (polecenie **left(90)**),

```
def slupek(k1,k2,k3):
    pd()
    kwadrat(k1)
    pu()
    left(90)
    fd(bok)
    right(90)
```

następnie przesunąć o bok kroków przed siebie (polecenie **fd(bok)**), po czym obrócić go o 90 stopni w stronę prawą, aby kolejne wywołanie procedury **kwadrat()** dało nam rysunek figury w odpowiednim miejscu.

4 Następnie możemy użyć polecenia **kwadrat(k2)**, co da nam rysunek

```
def slupek(k1,k2,k3):
    pd()
    kwadrat(k1)
    pu()
    left(90)
    fd(bok)
    right(90)
    pd()
    kwadrat(k2)
    pu()
```

kwadratu w kolorze zapisanym przy wywoływaniu słupka jako środkowy parametr. Pamiętajmy też o przyłożeniu pisaka przed narysowaniem kwadratu i podniesieniu go po narysowaniu.

5 To, co musimy zrobić pomiędzy narysowaniem pierwszego i drugiego kwadratu w słupku, niczym nie różni się od tego, co należy wykonać pomiędzy narysowaniem drugiego i trzeciego kwadratu w słupku. Musimy użyć dokładnie tego samego ciągu

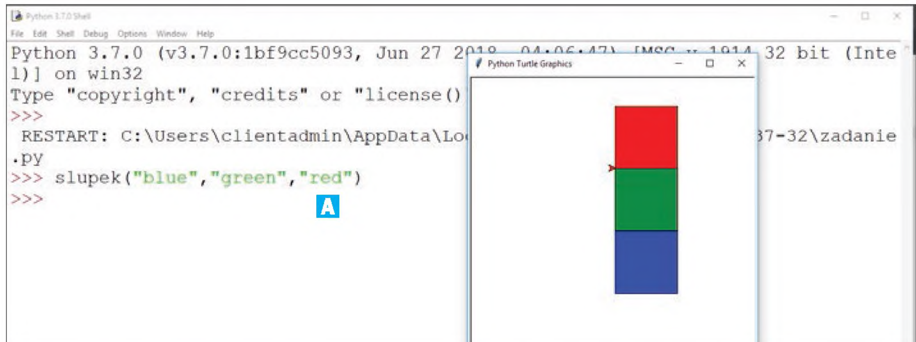
```
def slupek(k1,k2,k3):
    pd()
    kwadrat(k1)
    pu()
    left(90)
    fd(bok)
    right(90)
    pd()
    kwadrat(k2)
    pu()
    left(90)
    fd(bok)
    right(90)
```

poleceń co wcześniej, aby przemieścić się w lewy górny róg tego z dwóch kwadratów, który znajduje się wyżej.

```
def slupek(k1,k2,k3):
    pd()
    kwadrat(k1)
    pu()
    left(90)
    fd(bok)
    right(90)
    pd()
    kwadrat(k2)
    pu()
    left(90)
    fd(bok)
    right(90)
    pd()
    kwadrat(k3)
    pu()
```

6 Dalej również jest znajomo, bo pozostaje nam do zrobienia przyłożenie pisaka, wywołanie rysowania kwadratu kolorem **k3** i podniesienie pisaka.

7 Rozwiązanie możemy przetestować – po uruchomieniu skryptu, wpisując w konsoli wywołanie rysowania słupka składającego się z niebieskiego kwadratu na dole, zielonego pośrodku i czerwonego u samej góry poprzez polecenie **slupek("blue", "green", "red")** A (patrz kolejna strona).
Dlaczego czerwony kwadrat, mimo że nie mamy takiego w treści zadania? Procedura napisana przez nas w ten sposób powinna pozwalać na rysowanie kwadratów w dowolnym kolorze – więc i taką możliwość powinniśmy przetestować.



Sumowanie cyfr liczby

Kolejne zagadnienie nie jest ściśle związane z modułem **turtle**. To typowe zadanie matematyczno-logiczne, polegające na obliczeniu sumy cyfr, z których składa się liczba. Tym razem zamiast procedury stworzymy funkcję. W Pythonie składanie tych dwóch elementów nie różni się. O ile przez procedurę możemy rozumieć zdefiniowany fragment kodu (jej wywołanie wykonuje po prostu ten fragment kodu), o tyle funkcja działa bardziej jak zmienna. Oprócz wykonania fragmentu kodu niesie w sobie określoną wartość (liczbową albo tekstową). Wywołanie naszej funkcji będzie niosło w sobie wartość liczbową, czyli wartość sumy cyfr liczby przekazanej do funkcji przez parametr.

1 Tworzymy więc definicję o nazwie **sumuj**, która w parametrze pobiera liczbę.

```
def sumuj(liczba):
```

2 W treści zadania wyraźnie podkreślono, że liczby mają być trzycyfrowe. W związku z tym definicja funkcji mogłaby działać tylko na takich liczbach. I takie rozwiązanie przyjmijmy: działamy tylko dla liczb trzycyfrowych. A co by było, gdyby ktoś spróbował podać inną liczbę? Funkcja mogłaby nie reagować. Poprzez **if** sprawdźmy, czy liczba jest trzycyfrowa (czyli większa niż 99 i mniejsza niż 1000).

```
def sumuj(liczba):
    if liczba > 99 and liczba < 1000:
```

3 Kolejny krok będzie dość matematyczny (matematyka idzie w parze z programowaniem). W jaki sposób „dobrać się" do poszczególnych cyfr z liczby? Rozwiązanie jest prostsze, niż mogłoby się wydawać. Z pomocą przychodzi nam dzielenie liczby przez 10. W każdym przypadku, w którym ostatnia cyfra jest różna od zera, podzielenie liczby przez 10 da nam w wyniku ułamek. Dzieje się tak, jeśli korzystamy ze „zwykłego" dzielenia. W matematyce jest jednak jeszcze dzielenie z resztą. I właśnie reszta z dzielenia liczby przez 10 daje nam dostęp do ostatniej cyfry. W Pythonie mamy bardzo proste polecenie wykorzystujące resztę z dzielenia – to znak **%**. Aby zdobyć cyfrę jedności z liczby i zapisać ją do zmiennej jedności, tworzymy polecenie **jedności = liczba%10**.

```
def sumuj(liczba):
    if liczba > 99 and liczba < 1000:
        jedności = liczba%10
```

4 Dalej powinniśmy uzyskać z liczby cyfrę dziesiątek i cyfrę setek. Czy można zrobić to podobnie do cyfry jedności? Oczywiście, można to zrobić, także korzystając z reszty z dzielenia przez 10. Jednak najpierw trzeba pozbyć się ostatniej cyfry z liczby. Jak to zrobić? Od naszej liczby odejmujemy najpierw to, co mamy w zmiennej **jedności**, a wynik dzielimy przez 10.

```
def sumuj(liczba):
    if liczba > 99 and liczba < 1000:
        jedności = liczba%10
        liczba = (liczba - liczba%10)/10
```

5 Dostaniemy wtedy liczbę dwucyfrową, z której możemy „odczytać” ostatnią cyfrę. Robimy to, tworząc zmienną **dziesiątki** i zapisując w niej resztę z dzielenia liczby przez 10.

```
def sumuj(liczba):
    if liczba > 99 and liczba < 1000:
        jedności = liczba%10
        liczba = (liczba - liczba%10)/10
        dziesiątki = liczba%10
```

6 Co teraz z ostatnią z potrzebnych nam cyfr? Znowu zastosujmy zabieg „pozbywania się” obliczonej cyfry z liczby. Tym razem od liczby odejmujemy wartość zmiennej **dziesiątki** i uzyskany wynik dzielimy przez 10.

```
def sumuj(liczba):
    if liczba > 99 and liczba < 1000:
        jedności = liczba%10
        liczba = (liczba - liczba%10)/10
        dziesiątki = liczba%10
        liczba = (liczba - liczba%10)/10
```

7 To, co zostało w zmiennej liczba, to cyfra setek. Dla przejrzystości kodu możemy zapisać ją wprost do zmiennej **setki**.

```
def sumuj(liczba):
    if liczba > 99 and liczba < 1000:
        jedności = liczba%10
        liczba = (liczba - liczba%10)/10
        dziesiątki = liczba%10
        liczba = (liczba - liczba%10)/10
        setki = liczba
```

8 Wiemy już, co odróżnia funkcję od procedury – nasza definicja jest jeszcze procedurą. Aby stała się funkcją, musi znaleźć się w niej słowo **return**. Po nim należy

```
def sumuj(liczba):
    if liczba > 99 and liczba < 1000:
        jedności = liczba%10
        liczba = (liczba - liczba%10)/10
        dziesiątki = liczba%10
        liczba = (liczba - liczba%10)/10
        setki = liczba
    return jedności + dziesiątki + setki
```

napisać, jaką wartość ma zwracać funkcja. A ma zwracać sumę cyfry jedności, dziesiątek i setek.

9 Aby przetestować działanie napisanej funkcji, uruchamiamy skrypt i wpisujemy w konsoli polecenie **print(sumuj(753))**. W parametrze możemy użyć oczywiście innej trzycyfrowej liczby. Program w odpowiedzi powinien w konsoli wpisać wtedy sumę cyfr tej liczby.

Kodowanie jednego słupka

Mamy już stworzone funkcje i procedury niezbędne, aby móc zakodować pojedynczy słupek na podstawie jednej trzycyfrowej liczby.

1 Tworzymy procedurę **koduj_slupek()**, która w parametrze przyjmuje liczbę trzycyfrową.

```
def koduj_slupek(liczba):
```

2 Zależnie od tego, w jakim zakresie znajduje się suma cyfr liczby z parametru, rysujemy słupek oparty na kwadratach w innej kombinacji kolorów. Jeżeli suma cyfr jest większa bądź równa 1 i jednocześnie mniej-

```
Python 3.6.4 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
================ RESTART: C:\Users\konra\Downloads\zadanie.py ================
>>> print(sumuj(753))
15.0
>>> |
```

```
def koduj_slupek(liczba):
    if sumuj(liczba) >= 1 and sumuj(liczba) <= 5:
        slupek("blue","blue","green")
```

sza bądź równa 5, rysujemy słupek zawierający dwa niebieskie kwadraty u dołu i jeden zielony na samej górze.

3 Na tej samej zasadzie, budując kolejne **if** po jednym dla każdego zakresu liczb, sprawdzamy, w którym zakresie znajduje się suma cyfr z liczby.

```
def koduj_slupek(liczba):
    if sumuj(liczba) >= 1 and sumuj(liczba) <= 5:
        slupek("blue","blue","green")
    if sumuj(liczba) >= 6 and sumuj(liczba) <= 10:
        slupek("blue","green","green")
    if sumuj(liczba) >= 11 and sumuj(liczba) <= 15:
        slupek("green","green","green")
    if sumuj(liczba) >= 16 and sumuj(liczba) <= 20:
        slupek("green","green","blue")
    if sumuj(liczba) >= 21 and sumuj(liczba) <= 25:
        slupek("green","blue","blue")
    if sumuj(liczba) >= 26 and sumuj(liczba) <= 26:
        slupek("blue","blue","blue")
```

4 Podobnie jak w przypadku realizacji poprzednich etapów zadania, i ten etap należy przetestować. Uruchamiamy skrypt, wpisujemy w konsoli **koduj_slupek(838)** i sprawdzamy, czy efekt jest taki sam, jak na rysunku. Możemy sprawdzić też kodowanie innych liczb i upewnić się, że program poprawnie sumuje i zależnie od sumy inaczej koloruje słupek.

Tworzenie całego rysunku

Możemy teraz przejść do stworzenia procedury **zakoduj()**, która jest głównym elementem zadania.

```
def zakoduj(a,b,c):
```

1 Zgodnie z treścią zadania procedura ta powinna przyjmować w parametrze trzy liczby. Nazwijmy je **a**, **b** i **c**.

2 Od razu możemy przejść do zakodowania pierwszej z liczb – poprzez polecenie **koduj_slupek(a)**.

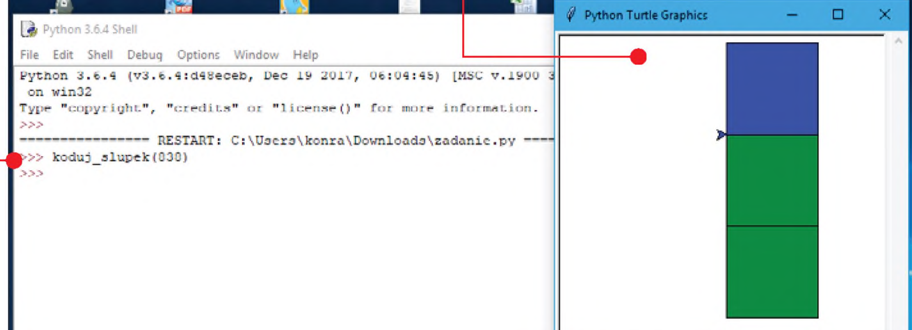
```
def zakoduj(a,b,c):
    koduj_slupek(a)
```

3 Po narysowaniu jednego słupka żółw znajduje się w lewym dolnym rogu górnego kwadratu słupka. Aby móc narysować kolejny słupek w odpowiednim miejscu, żółw powinien znajdować się w prawym dolnym rogu dolnego kwadratu słupka. Musimy go tam zaprowadzić. W związku z tym przesuwamy go do przodu o bok kroków.

```
def zakoduj(a,b,c):
    koduj_slupek(a)
    fd(bok)
```

4 Teraz żółw jest już w prawym dolnym rogu górnego kwadratu. A powinien być o dwukrotność boku niżej. Ponieważ jest on zwrócony w prawą stronę, obracamy go o 90 stopni w prawo. Kiedy jest on już zwróco-

```
def zakoduj(a,b,c):
    koduj_slupek(a)
    fd(bok)
    right(90)
    fd(bok*2)
```



ny w dół, może przejść dwukrotność boku przed siebie. W ten sposób dociera do docelowego miejsca.

5 Aby słupek został narysowany w odpowiednim miejscu, żółw powinien „patrzeć" w prawą stronę. Teraz „patrzy" w dół. Należy więc go obrócić o 90 stopni w lewą stronę.

```
def zakoduj(a,b,c):
    koduj_slupek(a)
    fd(bok)
    right(90)
    fd(bok*2)
    left(90)
```

6 Kiedy żółw jest już obrócony w odpowiednią stronę, można wywołać kodowanie drugiej z liczb przekazanych przez parametr, czyli liczby **b**.

```
def zakoduj(a,b,c):
    koduj_slupek(a)
    fd(bok)
    right(90)
    fd(bok*2)
    left(90)
    koduj_slupek(b)
```

7 W ten sposób mamy już figurę z dwóch słupków, a żółw kończy rysowanie, będąc w lewym dolnym rogu górnego kwadratu z drugiego słupka. Droga, jaką musi przejść do miejsca, w którym powinien się znajdować, gdy będziemy kodować trzecią z liczb, jest taka sama, jaką musiał pokonać przed narysowaniem drugiego słupka. Używamy więc tych samych poleceń.

```
def zakoduj(a,b,c):
    koduj_slupek(a)
    fd(bok)
    right(90)
    fd(bok*2)
    left(90)
    koduj_slupek(b)
    fd(bok)
    right(90)
    fd(bok*2)
    left(90)
```

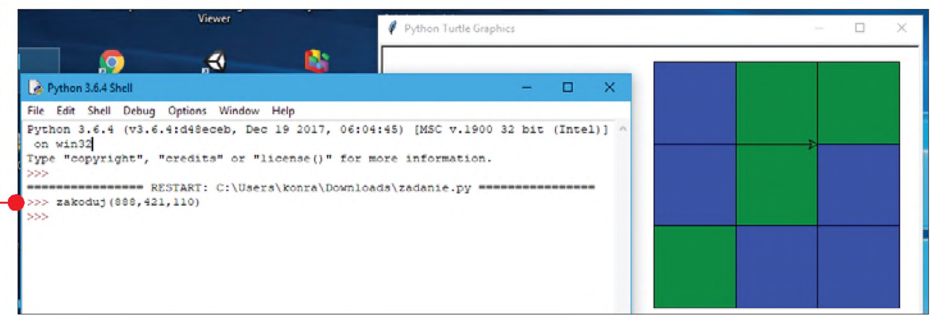
8 Na koniec możemy uruchomić procedurę **koduj_slupek()** dla liczby **c**.

```
def zakoduj(a,b,c):
    koduj_slupek(a)
    fd(bok)
    right(90)
    fd(bok*2)
    left(90)
    koduj_slupek(b)
    fd(bok)
    right(90)
    fd(bok*2)
    left(90)
    koduj_slupek(c)
```

Testowanie

Jeśli wszystko zostało napisane zgodnie z instrukcją, zadanie można uznać za rozwiązane. Oczywiście nie jest to jedyny sposób na jego realizację, a tylko jedna z propozycji. Możemy spróbować rozwiązać to samo zadanie w inny sposób.

Niezależnie od sposobu, w jaki zadanie zostało rozwiązane, przed nami ważny etap, czyli testowanie. Uruchamiamy napisany skrypt i w konsoli wywołujemy procedurę **zakoduj** dla wybranego zestawu trzech liczb. Sprawdźmy, czy uda się stworzyć kratkę zgodną z wymaganiami zadania. Przetestujmy rozwiązanie na kilku zestawach liczb.



Zadanie 3: Program sam wstawia znaki

Zgodnie z podpowiedzią, powinniśmy zacząć od stworzenia zmiennej **czyj_ruch**, w której zapiszemy, czy postawiony ma być teraz krzyżyk, czy kółko. Jeśli przyjmiemy, że grę rozpoczyna gracz stawiający krzyżyk, w nowej zmiennej zapiszmy wartość **"x"**. Gdyby ktoś chciał, aby w jego wersji gry rozgrywkę rozpoczynał gracz stawiający kółko, do zmiennej mógłby wpisać wartość **o**. (Tak naprawdę wartości te są umowne; moglibyśmy wpisać tam jakiekolwiek inne znaki, ponieważ najważniejsze jest to, co potem zawrzemy w procedurze **postaw()**).

```
czyj_ruch = "x"
```

1 Deklarujemy procedurę **postaw()**, która do poprawnego działania musi w parametrze otrzymać dwie liczby – **a** i **b**. W końcu musi ona wiedzieć, gdzie należy postawić znak.

```
czyj_ruch = "x"

def postaw(a,b):
```

Jak można przedstawić logikę działania procedury **postaw()**? Sprawdza ona, czy znak, który należy postawić, to krzyżyk. Jeśli tak, to stawia krzyżyk w określonym przez gracza miejscu i mówi, że następny znak do postawienia to kółko. Jeżeli znakiem, który mieliśmy postawić, nie jest krzyżyk, tylko kółko, procedura stawia kółko w określonym miejscu i mówi, że kolejnym znakiem do postawienia jest krzyżyk. Nie jest to skomplikowany mechanizm. Nie wymaga też wiele pisania.

2 Trzeba zacząć od tego, że zgodnie z logiką procedury ma ona zmieniać znak, który będziemy musieli postawić w kolejnym jej wywołaniu. Znak ten przechowujemy w zmiennej **czyj_ruch**, którą musieliśmy zadeklarować poza definicją procedury. Kiedy korzystamy ze zmiennych spoza procedury, możemy dowolnie odczytywać ich wartości. Gorzej jest ze zmianą wartości takiej zmiennej. Jeśli chcemy w procedurze zmieniać wartość zmiennej **czyj_ruch**, to powinniśmy zaznaczyć zmienną **czyj_ruch** w procedurze jako globalną.

```
def postaw(a,b):
    global czyj_ruch
```

3 Niezbędna okaże się także instrukcja warunkowa – to w niej sprawdzamy, czy zmienna **czyj_ruch** ma wartość **"x"**.

```
def postaw(a,b):
    global czyj_ruch
    if czyj_ruch == "x":
```

4 Jeśli warunek jest spełniony, należy postawić krzyżyk w polu określonym przez podane przez gracza **a** i **b**, wywołując polecenie **krzyzyk (a,b)**.

```
def postaw(a,b):
    global czyj_ruch
    if czyj_ruch == "x":
        krzyzyk(a,b)
```

5 Drugą instrukcją, którą należy wykonać w przypadku spełnienia tego warunku, jest wskazanie, że następny ruch to będzie kółko. W związku z tym trzeba zmienić wartość zmiennej **czyj_ruch** na **"o"**.

```
def postaw(a,b):
    global czyj_ruch
    if czyj_ruch == "x":
        krzyzyk(a,b)
        czyj_ruch = "o"
```

6 Dalej należałoby napisać, co ma się stać, gdy warunek opisany w **if** nie zostanie spełniony. Możemy użyć do tego instrukcji **elif** rozbudowującej **if** o drugi warunek,

```
def postaw(a,b):
    global czyj_ruch
    if czyj_ruch == "x":
        krzyzyk(a,b)
        czyj_ruch = "o"
    elif czyj_ruch == "o":
        kolko(a,b)
```

który jest sprawdzany, gdy pierwszy nie został spełniony. W tym drugim warunku możemy sprawdzać, czy zmienna **czyj_ruch** miała wartość **"o"**.

7 Jeśli drugi z warunków był spełniony, to należy postawić kółko poprzez wywołanie polecenia **kolko (a,b)**.

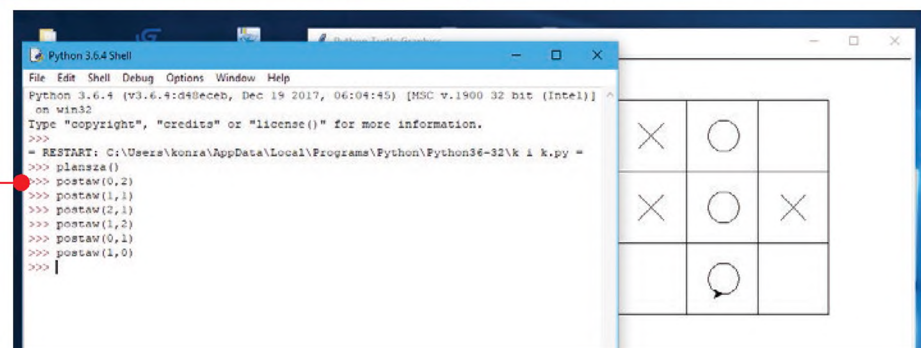
```
def postaw(a,b):
    global czyj_ruch
    if czyj_ruch == "x":
        krzyzyk(a,b)
        czyj_ruch = "o"
    elif czyj_ruch == "o":
        kolko(a,b)
```

8 Analogicznie do czynności wykonywanych po spełnieniu pierwszego warunku również po spełnieniu drugiego warunku powinniśmy wskazać, kto ma wykonać kolejny ruch, zmieniając wartość zmiennej **czyj_ruch** na **"x"**.

```
def postaw(a,b):
    global czyj_ruch
    if czyj_ruch == "x":
        krzyzyk(a,b)
        czyj_ruch = "o"
    elif czyj_ruch == "o":
        kolko(a,b)
        czyj_ruch = "x"
```

Testowanie

Aby przetestować rozwiązanie, uruchamiamy skrypt z wprowadzonymi zmianami i rysujemy planszę. Następnie spróbujmy przeprowadzić rozgrywkę. To najlepszy sposób – łączy testowanie i rozrywkę. W konsoli wpisujemy kolejne polecenia **postaw()** odnoszące się do różnych pól planszy.



Zadanie 4: Blokada zajętego pola

Z pozoru rozwiązanie zadania polegającego na takiej zmianie stworzonego skryptu gry w kółko i krzyżyk, aby uniemożliwić postawienie znaku na zajętym już polu, może wydawać się skomplikowane. Jednak jest rozwiązanie, które można napisać w zaledwie czterech linijkach kodu! Zgodnie z podpowiedzią możemy wykorzystać zagadnienie dwuwymiarowych tablic, które znamy już z poprzednich rozdziałów.

1 Właśnie od dwuwymiarowej tablicy powinniśmy zacząć. To będzie pierwsza z tych czterech linijek kodu. Tworzymy tablicę o nazwie **pole** o wymiarze 3 na 3, wypełniając ją całą wartościami **False**, co

```
pole = [[False,False,False],[False,False,False],[False,False,False]]
```

w logice naszego rozwiązania będzie oznaczać, że wszystkie pola są puste.

2 Kolejne z czterech linijek powinny znaleźć się w treści procedury **postaw()**. Jej edytowanie jest podstawą rozwiązania zadania. W treści procedury będziemy odnosić się do tablicy **pole**, odczytując i zmieniając jej wartości. Dlatego w procedurze powinniśmy użyć polecenia **global pole** ●, które wytworzy połączenie pomiędzy tablicą zadeklarowaną poza procedurą a jej wykorzystywaniem wewnątrz procedury.

```
def postaw(a,b):
    global czyj_ruch
    global pole
    if czyj_ruch == "x":
        krzyzyk(a,b)
        czyj_ruch = "o"
    elif czyj_ruch == "o":
        kolko(a,b)
        czyj_ruch = "x"
```

3 Trzecia z czterech linijek powinna sprawdzać, czy w tablicy **pole** w miejscu ukrytym pod indeksami odpowiadającymi wybranym przez gracza liczbom **a** i **b** znajduje się wartość **False**. Jeśli tak, powinniśmy móc postawić znak w tym polu. Aby zrealizować ten mechanizm, przed **if** sprawdzającymi wartość zmiennej **czyj_ruch** dokładamy kolejny **if** ● sprawdzający wartość w tablicy pole w miejscu ukrytym pod indeksem **[a][b]**. Wymaga to również dodania kolejnego wcięcia przed wszystkimi linijkami, które znajdują się poniżej w procedurze.

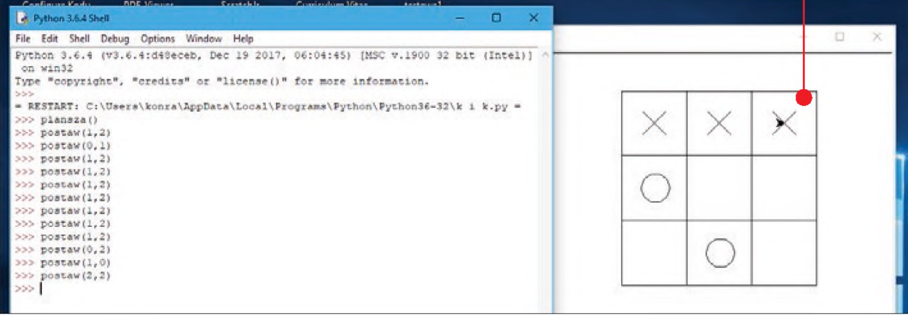
```
def postaw(a,b):
    global czyj_ruch
    global pole
    if pole[a][b] == False:
        if czyj_ruch == "x":
            krzyzyk(a,b)
            czyj_ruch = "o"
        elif czyj_ruch == "o":
            kolko(a,b)
            czyj_ruch = "x"
```

4 Ostatnia linijka jest kluczowa. Sprawdzaliśmy wartość w tablicy **pole**, ale wypełniliśmy ją samymi wartościami **False**. Do tej pory warunek musiał być zawsze spełniony. Teraz w tablicy **pole** w miejscu odpowiadającemu polu na planszy wybranemu przez gracza powinniśmy umieścić wartość **True** ●, która będzie oznaczała, że ta lokalizacja została już zajęta i nie można postawić w niej kolejnego znaku.

```
def postaw(a,b):
    global czyj_ruch
    global pole
    if pole[a][b] == False:
        pole[a][b] = True
        if czyj_ruch == "x":
            krzyzyk(a,b)
            czyj_ruch = "o"
        elif czyj_ruch == "o":
            kolko(a,b)
            czyj_ruch = "x"
```

Testowanie

Aby przetestować rozwiązanie, uruchamiamy skrypt z wprowadzanymi zmianami i rysujemy planszę. Następnie spróbujmy przeprowadzić kolejną rozgrywkę. Tym razem nie trzymamy się ściśle zasad i próbujemy postawić znak w zajętym już miejscu ●. Program powinien to uniemożliwiać.



7 Podsumowanie: co warto wiedzieć o Pythonie

Książka zbliża się do końca. Jednak nasza przygoda z programowaniem w Pythonie nie powinna się skończyć. Teraz czas na dużo samodzielnej pracy – szczególnie nad projektami opartymi na własnych pomysłach

Aby lepiej zapamiętać polecenia i instrukcje, których używaliśmy podczas wykonywania wskazówek, zapoznajmy się z przykładowymi fragmentami kodu i ich opisami. Przedstawione zagadnienia mogą nam się później przydać w naszych projektach.

Program wczytujący trzy liczby i wyznaczający najmniejszą z nich

```
A czy_dalej = "t"
B while czy_dalej == "t":
C     a = int(input("Podaj pierwszą liczbę: "))
D     b = int(input("Podaj drugą liczbę: "))
E     c = int(input("Podaj trzecią liczbę: "))

F     if a < b:
G         if a < c:
H             najmniejsza = a
I         else:
J             najmniejsza = c
K     elif b < c:
L         najmniejsza = b
M     else:
N         najmniejsza = c

O     print(najmniejsza)

P     czy_dalej = input("Czy chcesz podać liczby jeszcze raz (t/n)? ")
```

A – tworzy zmienną o nazwie **czy_dalej**, z wartością początkową **"t"**.

B – pętla **while**, która będzie powtarzać zawarte w niej instrukcje, aż do momentu, gdy zmienna **czy_dalej** przestanie mieć wartość **"t"**.

C – tworzy zmienną o nazwie **a**, której wartość będzie nadana podczas wykonywania programu. Podany w nawiasie i cudzysłowie tekst zostanie wyświetlony użytkownikowi, gdy ten będzie musiał podać liczbę. A podana wtedy liczba zostanie zrzutowana na typ **Integer** i zapisana w zmiennej. Typ **Integer** służy do przechowywania liczb całkowitych.

D – tworzy zmienną o nazwie **b** – analogicznie jak poprzednie polecenie.

E – tworzy zmienną o nazwie **c** – analogicznie do dwóch wcześniejszych linijek kodu.

F – instrukcja warunkowa sprawdzająca, czy zmienna **b** ma wartość większą niż zmienna **a**.

G – instrukcja warunkowa sprawdzająca, czy zmienna **c** ma wartość większą niż zmienna **a**; sprawdzenie to będzie wykonywać się tylko wtedy, gdy zmienna **a** ma wartość mniejszą niż zmienna **b**.

H – tworzy zmienną o nazwie **najmniejsza** i nadaje jej wartość zmiennej **a**. Ma to wskazać, że ta z trzech zmiennych ma najmniejszą wartość. Można tak zrobić, ponieważ ta linijka kodu wykona się tylko wtedy, gdy dwa wcześniejsze warunki zostały spełnione, to znaczy, gdy zmienna **a** ma wartość mniejszą od zmiennej **b** i mniejszą od zmiennej **c**.

I – dodaje sekcję **else** do poprzedzającej tę linijkę instrukcji warunkowej.

J – tworzy zmienną **najmniejsza** i nadaje jej wartość zmiennej **c**. Linijka ta wykona się tylko wtedy, gdy warunek z pierwszego **if** jest spełniony, ale z drugiego już nie.

K – dodaje sekcję **elif** do pierwszej instrukcji warunkowej. Sekcja ta ma swój wa-

runek sprawdzający, czy zmienna **b** ma wartość mniejszą niż **c**.

L – tworzy zmienną **najmniejsza** i nadaje jej wartość zmiennej **b**. Można tak zrobić, ponieważ ta linia kodu wykona się tylko wtedy, gdy warunek **a<b** nie jest spełniony, a warunek **b<c** jest spełniony.

M – dodaje sekcję **else** do pierwszej instrukcji warunkowej.

N – tworzy zmienną **najmniejsza** i nadaje jej wartość zmiennej **c**. Możemy tak zrobić, ponieważ ta linijka kodu wykona się, tylko gdy warunki **a<b** i **b<c** nie zostały spełnione.

O – wypisuje wartość zmiennej **najmniejsza**.

P – prosi użytkownika o podanie nowej wartości zmiennej **czy_dalej**.

CO IMPORTOWAĆ, GDY CHCĘ...

korzystać z funkcji matematycznych	import math	`import math`
tworzyć rysunki, przemieszczając po oknie grafikę żółwia	import turtle	`import turtle`
używać liczb losowych (pseudolosowych)	import random	`import random`
tworzyć programy użytkowe	import tkinter	`import tkinter`
tworzyć gry z grafiką 2D	import pygame	`import pygame`

Program wczytujący 10 liczb i określający, które z nich są parzyste, a które nie

```
A  liczby = []
B  for i in range(10):
C      liczby.append(int(input("Podaj liczbę: ")))

D  for i in liczby:
E      if i%2 == 0:
F          print(str(i) + " - Parzysta")
G      else:
H          print(str(i) + " - Nieparzysta")
```

A – tworzy listę o nazwie **liczby**.

B – tworzy pętlę **for**, której treść wykona się 10 razy. Zmienna w tej pętli, z każdym jej przejściem, będzie przyjmowała kolejne wartości od 0 do 9.

C – prosi użytkownika o podanie liczby, której wartość jest rzutowana na typ **Integer**, a następnie dodawana do listy o nazwie **liczby**.

D – tworzy pętlę **for**, która wykona się tyle

razy, ile elementów jest na liście o nazwie **liczby**. Zmienna **i** w tej pętli będzie przyjmować wartości kolejnych elementów listy.

E – dla elementu z listy **liczby** sprawdza, czy reszta z dzielenia liczby przez 2 jest równa 0 (jeśli tak, liczba jest parzysta).

F – gdy warunek określony w instrukcji warunkowej jest spełniony, wykonuje się linijka kodu, która wypisuje wartość zmiennej **i**, a potem słowo **"Parzysta"** jako informację o liczbie.

G – dodaje do instrukcji warunkowej sekcję **else**.

H – ta linijka kodu wykonuje się, tylko gdy podany w instrukcji warunkowej warunek nie jest spełniony. Wtedy powinna zostać wypisana wartość zmiennej **i**, po czym w tej samej linijce wyświetlane jest słowo **"Nieparzysta"** jako informacja o liczbie.

NAJWAŻNIEJSZE POLECENIA Z MODUŁU MATH

Nie wszystkie z nich zostały wykorzystane w zrealizowanych w książce zadaniach, jednak warto zapoznać się z nimi, ponieważ mogą okazać się przydatne w przyszłości.

cos(x)	Zwraca cosinus argumentu x
sin(x)	Zwraca sinus argumentu x
tan(x)	Zwraca tangens argumentu x
acos(x)	Zwraca arcus cosinus argumentu x
asin(x)	Zwraca arcus sinus argumentu x
cosh(x)	Zwraca cosinus hiperboliczny argumentu x
sinh(x)	Zwraca sinus hiperboliczny argumentu x
tanh(x)	Zwraca tangens hiperboliczny argumentu x
sqrt(x)	Zwraca pierwiastek kwadratowy argumentu x
pow(x,y)	Zwraca wartość argumentu x podniesioną do potęgi y
fabs(x)	Zwraca wartość bezwzględną argumentu x
degrees(x)	Zwraca wartość kąta x podaną w radianach – w stopniach
radians(x)	Zwraca wartość kąta x podaną w stopniach – w radianach
log10(x)	Zwraca logarytm z argumentu x o podstawie 10
log(x,y)	Zwraca logarytm o podstawie y z argumentu x. Jeśli podstawa logarytmu nie jest podana, zwraca logarytm naturalny z argumentu x
pi	Zwraca wartość liczby Pi
e	Zwraca wartość liczby E

Jak zatrzymać na chwilę wykonywanie programu?

Spośród modułów Pythona, które nie zostały omówione w książce, warto wymienić moduł **time**. Jest on wbudowany w Pythona i nie musimy go dodatkowo instalować, wystarczy sam import. Moduł ten ma ciekawe polecenie **time.sleep(x)**, gdzie **x** oznacza czas zatrzymania programu wyznaczony w sekundach.

Do czego można wykorzystać takie polecenie? Na przykład do budowy minutnika, który odliczy zadany czas. Oto przykładowy skrypt budujący minutnik.

```
import time
czas = 0
czas_do_odliczenia = int(input("Podaj ile sekund odliczyć: "))
for i in range (czas_do_odliczenia):
    czas = czas + 1
    time.sleep(1)
    print(czas)
print("Czas minął")
```

IMPORT MODUŁÓW – NAJWAŻNIEJSZE INFORMACJE

W Pythonie mamy dwa sposoby importowania modułów, które były pokazane podczas realizacji zadań. Obydwa są przydatne, dlatego też powinniśmy umieć je wykorzystywać.

Pierwszym ze sposobów jest użycie polecenia **import nazwamodułu**. Korzystając z poleceń zawartych w module importowanym w ten sposób, musimy przed nazwą polecenia podawać też nazwę modułu, z którego ono pochodzi. Na przykładzie modułu **turtle** wyglądałoby to następująco.

Drugi sposób to użycie polecenia **from nazwamodułu import ***. Atrybuty i metody modułu są importowane bezpośrednio do lokalnej przestrzeni nazw, a więc będą dostępne bezpośrednio i nie musimy określać, z którego modułu pochodzą. Możemy importować określone polecenia albo skorzystać ze znaku *, aby zaimportować wszystko. Ostatnia z opcji najlepiej sprawdza się w programach, gdzie importujemy jeden bądź niewiele więcej modułów. Gdy musimy importować dużo modułów, lepiej korzystać z pierwszej opcji, aby nie pomylić poleceń.

```
import turtle

turtle.fd(100)
turtle.right(90)
turtle.fd(120)
turtle.left(90)
```

```
import turtle
from turtle import fd

fd(100)
turtle.right(90)
fd(120)
turtle.left(90)
```

```
from turtle import *

fd(100)
right(90)
fd(120)
left(90)
```

NAJWAŻNIEJSZE POLECENIA Z MODUŁU TURTLE

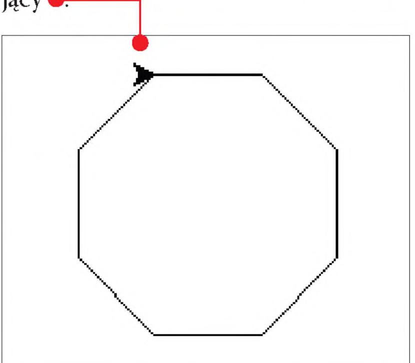
forward(x), fd(x)	Przesuwają żółwia do przodu, w kierunku, w którym jest on zwrócony, o liczbę kroków podaną jako argument x
backward(x), bk(x), back(x)	Przesuwają żółwia do tyłu, względem strony, w którą jest on zwrócony, o liczbę kroków podaną jako argument x
right(x), rt(x)	Obracają żółwia w prawą stronę o kąt x wyrażony w stopniach
left(x), lt(x)	Obracają żółwia w lewą stronę o kąt x wyrażony w stopniach
goto(x,y), setpos(x,y), setposition(x,y)	Przesuwają żółwia do punktu w oknie programu określonego współrzędnymi x i y
setx(x)	Przesuwa żółwia po osi X do współrzędnej określonej argumentem x
sety(x)	Przesuwa żółwia po osi Y do współrzędnej określonej argumentem x
setheading(x), seth(x)	Ustawia kierunek żółwia na podany w argumencie x
home()	Przesuwa żółwia do punktu początkowego (0,0)
circle(x)	Przesuwa żółwia po okręgu o określonym promieniu x
stamp()	Pozostawia w oknie programu stempel w postaci grafiki żółwia
clearstamp(x)	Usuwa stempel żółwia o identyfikatorze x, który jest nadawany zgodnie z kolejnością tworzenia stempli
clearstamps(x)	Usuwa ostatnie lub początkowe stemple grafiki żółwia. Jeśli x nie jest podany, usuwa wszystkie stemple. Jeżeli jest on większy od zera, usuwa x początkowych stempli. Jeśli x jest poprzedzony znakiem minus, usuwa x ostatnich stempli
undo()	Przesuwa żółwia do pozycji, w której znajdował się przed użyciem ostatniego polecenia. Operacja jest możliwa do wykorzystania w pętli
speed(x)	Ustawia prędkość animacji poruszania się żółwia w skali od 1 do 10. Możliwe jest podanie wartości 0, co oznacza brak animacji, czyli największą możliwą prędkość powstawania rysunku
pendown(), pd(), down()	Opuszczenie pisaka żółwia
penup(), pu(), up()	Podniesienie pisaka żółwia
pensize(x), width(x)	Ustawia rozmiar pisaka, czyli grubość linii rysowanej przez żółwia
isdown()	Zwraca wartość True lub False, zależnie od tego, czy pisak żółwia jest opuszczony
pencolor(x)	Ustawia kolor linii rysowanych przez żółwia na kolor podany w argumencie x
fillcolor(x)	Ustawia kolor wypełnienia kształtów wytyczonych przez żółwia
begin_fill()	Rozpoczyna śledzenie ścieżki pokonywanej przez żółwia
end_fill()	Kończy śledzenie ścieżki pokonywanej przez żółwia i zamyka kształt

Procedura do wytyczania żółwiem dowolnych wielokątów foremnych

```
A from turtle import *

B def wielobok(boki,dl_boku):
C     for i in range(boki):
D         fd(dl_boku)
E         right(360/boki)

F wielobok(8, 50)
```

- **A** – importuje do przestrzeni nazw projektu wszystkie polecenia z modułu **turtle**.
- **B** – tworzy procedurę o nazwie **wielobok**, której wywołanie wymaga podania dwóch argumentów – **boki** i **dl_boku**. Mają one oznaczać odpowiednio liczbę boków w wielokącie oraz długość każdego z boków w figurze.
- **C** – pętla **for**, która wykona się tyle razy, ile boków ma mieć figura, jaką chcemy stworzyć.
- **D** – przesuwa żółwia w przód w kierunku, w którym był ustawiony, o tyle kroków do przodu, ile równa jest wartość parametru **dl_boku**.
- **E** – zgodnie z matematyką suma kątów wewnątrz wielokąta foremnego ma być równa 360. Ponieważ wszystkie boki i kąty w tej figurze są takie same, liczba kątów to wartość parametru **boki** – każdy kąt ma **360/boki** stopni i o taki kąt należy obrócić żółwia. Tym razem w prawą stronę.
- **F** – wywołuje zdefiniowaną wcześniej procedurę **wielobok**, która ma narysować ośmiokąt foremny o bokach wielkości **50**.

Efekt uruchomienia programu jest następujący.



Program do przeliterowania dowolnego słowa

```
A slowo = input("Podaj słowo do przeliterowania: ")

B for litera in slowo:
C     print(litera)
```

Wykorzystamy tu bardzo ważną właściwość zmiennych typu **String**, czyli ciągów znaków. Takie zmienne możemy traktować jako listy.

A – tworzymy zmienną o nazwie **slowo**, której wartość ma być podana przez użytkownika programu.

B – pętla **for**, która wykona się tyle razy, ile liter ma słowo zapisane w zmiennej **slowo**.

C – wypisuje wartość zmiennej **litera**, która w każdym przejściu pętli **for** przyjmuje wartości kolejnych liter z ciągu znaków **slowo**.

Efekt powinien być następujący .

```
>>>
Podaj słowo do przeliterowania: kamizelka
k
a
m
i
z
e
l
k
a
>>> |
```

ZASOBY, KTÓRE MOGĄ CI SIĘ JESZCZE PRZYDAĆ

- **NumPy – Numerical Python** – podstawowy zestaw narzędzi, które umożliwiają zaawansowane obliczenia matematyczne na macierzach oraz szeregach i wektorach.
- **SciPy** – podobnie jak NumPy jest to zestaw narzędzi, który umożliwia wiele operacji matematycznych, a co najważniejsze – wiele metod numerycznych, takich jak całkowanie, różniczkowanie numeryczne, algorytmy rozwiązywania równań różniczkowych, algorytmy z algebry liniowej, transformaty Fouriera czy przetwarzanie sygnałów.
- **Pandas** – zestaw narzędzi do analizy danych relacyjnych (podobnych do SQL, lecz na obiektach).
- **Matplotlib** – moduł do tworzenia wykresów, który wraz z NumPy, SciPy i Pandas jest konkurentem dla popularnych narzędzi – MatLab i Mathematica.
- **Django** – wolny i otwarty framework przeznaczony do tworzenia aplikacji.
- **Pyglet** to biblioteka graficzna i multimedialna przeznaczona do tworzenia gier komputerowych oraz aplikacji multimedialnych w języku Python.
- **Scrapy** – otwartoźródłowy szkielet do tworzenia aplikacji napisany w języku programowania Python i służący do pisania robotów internetowych, które mogą przeszukiwać strony internetowe i wydobywać z nich określone dane.
- **Pylons** – framework stworzony w Pythonie, który służy do szybkiego tworzenia skalowalnych aplikacji internetowych.

Nie tylko IDLE

IDLE to IDE, które standardowo dołączone jest do Pythona. Nie oznacza to jednak, że chcąc pisać w tym języku, jesteśmy skazani tylko na ten program. Na płycie dołączonej do książki znajdziemy między innymi aplikację **Thonny** (DVD-KOD: 020) – to IDE Pythona przeznaczone szczególnie dla początkujących użytkowników.

Aplikacja ta w jednym oknie łączy pole, w którym wpisujemy skrypty ●, i **Shell** ●, czyli naszą konsolę do komunikacji użytkownika z programem.

NARZĘDZIA

Pisząc skrypty w Pythonie, możemy korzystać także z uniwersalnych edytorów kodu, jak dołączone na płycie **Notepad++** (DVD-KOD: 006/007 32-/64-BIT), **Sublime Text** (DVD-KOD: 018/019 32-/64-BIT) czy **Visual Studio Code** (DVD-KOD: 021/022 32-/64-BIT).

Thonny oferuje kilka przydatnych funkcji, jak możliwość sprawdzenia aktualnych wartości zmiennych, które pojawiają się w programie.

Thonny - C:\Users\programowanie\trsad.py @ 14 : 1

File Edit View Run Device Tools Help

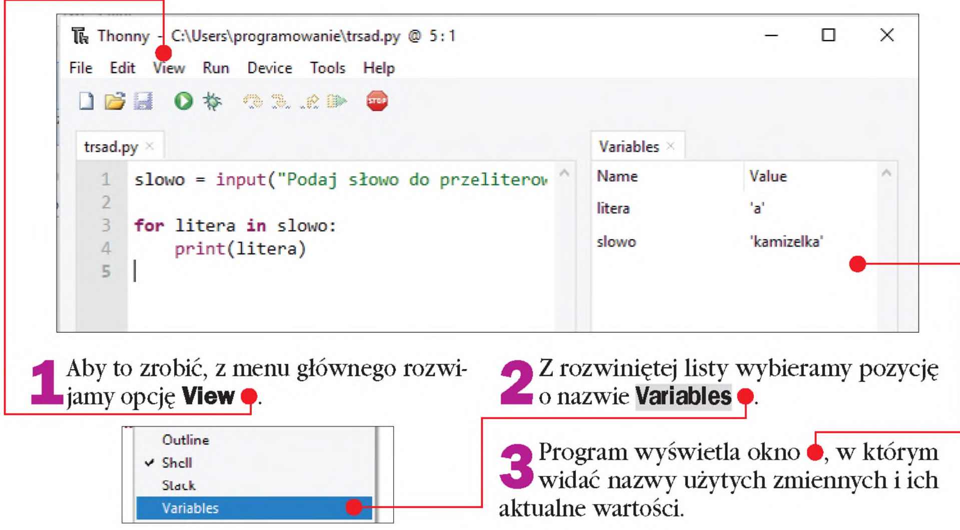
Program arguments:

trsad.py

```
import os
import time
czas = [0,0,0,0]
input("kliknięcie entera uruchamia stoper")
while True:
    if czas[3] ==10:
        czas[2]=czas[2]+1
        czas[3]=0
    elif czas[2] ==60:
        czas[1]=czas[1]+1
        czas[2]=0
    elif czas[1] ==60:
        czas[0]=czas[0]+1
        czas[1]=0
    os.system('cls')
    czas[3]=czas[3]+1
    print(czas[0],czas[1],czas[2],czas[3])
    time.sleep(0.1)
```

Shell

```
□0 0 0 1
□0 0 0 2
□0 0 0 3
□0 0 0 4
□0 0 0 5
□0 0 0 6
□0 0 0 7
□0 0 0 8
□0 0 0 9
□0 0 0 10
```



1 Aby to zrobić, z menu głównego rozwijamy opcję **View**.

2 Z rozwiniętej listy wybieramy pozycję o nazwie **Variables**.

3 Program wyświetla okno, w którym widać nazwy użytych zmiennych i ich aktualne wartości.

Python nie tylko dla programistów

Ten język programowania jest na tyle prosty, że często wykorzystuje się go w rozmaitych programach do tworzenia tak zwanych wtyczek. Są to programy przydatne w różnych dziedzinach nauki, często specjalistyczne, których wykorzystanie wymaga dużej wiedzy. Tak jest na przykład z programami typu **GIS**, czyli systemami informacji przestrzennej. Często pozwalają na tworzenie wtyczek właśnie w Pythonie. Popularność tego typu programów i ich specyfika, czyli wymaganie wiedzy geograficznej i informatycznej, doprowadziły do powstania dziedziny zwanej geoinformatyką, w której specjaliści od geografii, wykorzystując programistyczne umiejętności, mogą rozwijać programy przydatne właśnie geografom. To tylko jedna z gałęzi nauki bazująca na programowaniu. Tego typu połączeń jest więcej. Wśród programów wykorzystujących wtyczki w Pythonie należy wymienić **QGIS**.

DZIAŁANIA NA OGROMNYCH LICZBACH

Zaletą Pythona jest to, że doskonale radzi sobie z działaniami na ogromnych liczbach, które znacznie przekraczają zakres zmiennych w wielu innych językach. Dla Pythona wykonanie przykładowego działania **12345621412942149122139 + 423523534124141351341241 2** – nie stanowi problemu.

```
>>> 12345621412942149122139 + 4235235341241413513412412
4247580962654355662534551
```

Warto wiedzieć

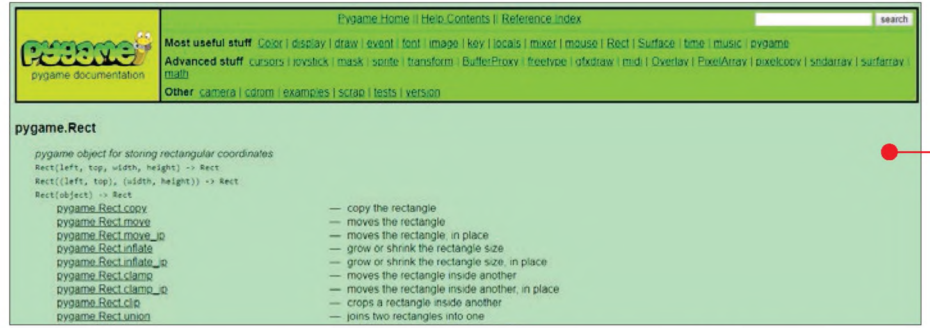
Filary programowania obiektowego – programowanie obiektowe opiera się na czterech filarach. Należą do nich – abstrakcja, hermetyzacja, polimorfizm i dziedziczenie.

- **Abstrakcja** – zgodnie z jej założeniami obiekt jest tylko wykonawcą metod, który może opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie bez konieczności ujawniania, w jaki sposób zaimplementowano dane cechy obiektu. Klasa stanowi zamkniętą całość, na przykład, jeśli znajduje się w niej metoda zwracająca dla obiektu informację o tym, czy obiekt na ekranie koliduje z innym, z punktu widzenia systemu istotne jest to, co zwraca dana metoda, a nie to, w jaki sposób została ona zaimplementowana.
- **Hermetyzacja** – tylko własne metody obiektu są uprawnione do zmiany jego stanu, obiekt nie może zmieniać stanu wewnętrznego innych obiektów w sposób niezaplanowany. Na przykład jeden „samochód" nie może zmienić długości innego „samochodu".
- **Dziedziczenie** – możemy tworzyć pewną hierarchię klas, która pozwala na wydzielenie ogólnych klas będących bazą dla większej liczby klas, które precyzują szczegóły. Dla obiektów klas dziedziczących nie trzeba ponownie definiować całej funkcjonalności, lecz tylko tę, której nie ma obiekt ogólniejszy. Na przykład możemy stworzyć klasę bazową „samochód" i opisać w niej właściwości i mechanizmy wspólne dla wszystkich rodzajów samochodów. Dalej możemy stworzyć klasy dziedziczące po niej – „ciężarówka" i „osobówka" – w których możemy implementować tylko rzeczy, które są różne dla tych dwóch rodzajów samochodów i nie zostały opisane w klasie bazowej.
- **Polimorfizm** – w kontekście programistycznym to mechanizmy pozwalające programiście używać wartości, zmiennych i podprogramów na kilka różnych sposobów. Możemy pisać ogólne struktury, bez precyzowania, jakiego typu danych one dotyczą. W pojęciu polimorfizmu mieści się też rzutowanie typów danych, które stosowaliśmy w kilku zadaniach.

Interpreter – IDLE to tak zwany interpreter Pythona. Interpreter to program komputerowy, który analizuje kod źródłowy programu, a przeanalizowane fragmenty wykonuje. Realizowane jest to w inny sposób niż w procesie kompilacji, gdzie tłumaczy się skrypty do wykonywalnego kodu maszynowego lub kodu pośredniego, który jest następnie zapisywany do pliku w celu późniejszego wykonania. W interpreterze każda kolejna linijka kodu brana przez interpreter jest od razu wykonywana.

Klasa – to całościowa lub częściowa definicja, na podstawie której powstają obiekty. Obejmuje ona stan obiektu (jego pola, właściwości), a także zachowanie, czyli metody.

Konkursy Informatyczne LOGIA – organizowane są przez Ośrodek Edukacji Informatycznej i Zastosowań Komputerów. To jedne z najpopularniejszych konkursów informatycznych, w których zadania polegają



na sprawdzeniu logicznego myślenia w połączeniu z umiejętnością wykorzystania języka Logo, także tego obecnego jako moduł **turtle** w Pythonie.

■ **Logo** to język programowania stworzony jako środek do nauczania informatyki i matematyki. Składa się z gotowych elementarnych procedur, które wykorzystuje się do definiowania procedur użytkownika. Język ten został zaprojektowany przez pracujących pod koniec lat sześćdziesiątych na MIT Seymoura Paperta i Jeana Piageta. Do dziś czerpią z niego inspirację edukacyjne narzędzia programistyczne. Podobne edukacyjne działanie ma obecny w języku Python moduł **turtle**.

■ **Moduł** – w Pythonie moduły są zwykłymi plikami z rozszerzeniem ***.py**, w których zawarto pewien zestaw poleceń. Moduły importujemy do swojego programu za pomocą komendy **import**. Działa to tak, jakbyśmy polecenia z tego pliku (z modułem) dodawali do swojego. To oznacza, że moduły może tworzyć każdy programista, nawet do użytku własnego.

■ **Obiekt** jest instancją klasy. Każdy obiekt ma trzy główne cechy. Pierwsza z nich to tożsamość umożliwiająca jego identyfikację i odróżnienie od innych obiektów. Druga to stan, a konkretniej aktualny stan jego danych składowych (czyli pól, właściwości). Trzecia to jego zachowanie, czyli zestaw metod wykonujących operacje na danych obiektu (czyli jego aktualnym stanie).

■ **Programowanie obiektowe** (object-oriented programming, OOP) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów.

■ **Pygame** – to moduł języka Python służący do tworzenia gier. Zawiera zestaw klas i funkcji, które znacząco ułatwiają tworzenie gier. Aby zapoznać się z jego możliwościami, najlepiej sprawdzić dokumentację modułu dostępną pod adresem **pygame.org/docs**. Każde polecenie z modułu jest opisane w dokumentacji wraz z przykładami.

■ **True i False** – to wartości, jakie może przyjmować zmienna typu boolowskiego. Jest to zmienna o najmniejszym możliwym rozmiarze. Mieści się bowiem na jednym bicie. Bit to najmniejsza jednostka informacji (to stąd pochodzą znane i kojarzące się z informatyką ciągi zer i jedynek).

■ **Zmienna** to reprezentacja jakiegoś obszaru w pamięci komputera. Zamiast pisać bezpośrednio, wskazując interesujący nas obszar pamięci, podajemy jedynie nazwę, która się do niego odnosi. Dobrą praktyką związaną z używaniem zmiennych jest nadawanie im nazw wzkazujących na to, jakie dane są w obszarze pamięci przez nią reprezentowane. Jeżeli zmienna ma reprezentować na przykład liczbę punktów, powinna nazywać się **punkty**, a nie **ashgfgajhasd**. Teoretycznie nie ma to znaczenia dla interpretera, ale ma dla programisty, który może zapomnieć, co oznacza nazwa niewskazująca na zawartość zmiennej.

Na płycie DVD

Płyta dołączona do książki zawiera zestaw startowy niezbędnych narzędzi dla programistów rozpoczynających swoją przygodę z językiem Python: najlepsze środowiska programistyczne, edytory kodu źródłowego i pliki szkoleniowe do wskazówek przedstawionych w książce.

PYTHON

Python 2.7.16 (32-bit)	DVD-KOD:013
Python 2.7.16 (64-bit)	DVD-KOD:014
Python 3.7.3 (32-bit)	DVD-KOD:015
Python 3.7.3 (64-bit)	DVD-KOD:016

ŚRODOWISKA PROGRAMISTYCZNE

Eclipse IDE 2019-03	DVD-KOD:003
PyCharm Community 2019.1.3	DVD-KOD:012
Thonny 3.1.2	DVD-KOD:020
Visual Studio Community 2019	DVD-KOD:023

EDYTORY KODU

Atom 1.38.2 (32-bit)	DVD-KOD:001
Atom 1.38.2 (64-bit)	DVD-KOD:002
Notepad++ 7.7 (32-bit)	DVD-KOD:006
Notepad++ 7.7 (64-bit)	DVD-KOD:007
Sublime Text 3.2.1 (32-bit)	DVD-KOD:018
Sublime Text 3.2.1 (64-bit)	DVD-KOD:019
Visual Studio Code 1.35.1 (32-bit)	DVD-KOD:021
Visual Studio Code 1.35.1 (64-bit)	DVD-KOD:022

MATERIAŁY SZKOLENIOWE

Skrypty	DVD-KOD:017

INNE

Java Runtime Environment 8 (32-bit)	DVD-KOD:004
Java Runtime Environment 8 (64-bit)	DVD-KOD:005
Panda3D 1.10.3 (32-bit)	DVD-KOD:008
Panda3D 1.10.3 (64-bit)	DVD-KOD:009
Panda3D Samples	DVD-KOD:010
Piskel 0.14	DVD-KOD:011

DVD-KOD np. 001 wpisujemy w pole wyszukiwarki w menu płyty

JAK SKORZYSTAĆ Z E-WYDANIA KSIĄŻKI

W KŚ+ znajdziemy **e-wydanie tej Biblioteczki i obraz ISO dołączonej do niej płyty z narzędziami dla programistów i plikami szkoleniowymi do wskazówek opisanych w książce.**

1 Otwieramy stronę **www.ksplus.pl**. Logujemy się (używamy konta z serwisu **Komputerswiat.pl**). Jeżeli nie mamy konta, klikamy na , by się zarejestrować.

Załóż konto Logowanie
Zarejestruj kod

2 Po zalogowaniu się możemy zarejestrować kod nadrukowany na płycie dołączonej do książki. Wystarczy kliknąć na link i przepisać kod.

Moje konto
Zarejestruj kod

3 Uzyskamy w ten sposób dostęp do e-wydania i do bonusowego obrazu płyty . Do serwisu KŚ+ możemy logować się z dowolnego urządzenia z dostępem do internetu.

CZYTAJ E-WYDANIE
PROGRAMY
BONUSY

UWAGA! W KŚ+ ZA DARMO E-WYDANIE KSIĄŻKI ORAZ PLIK ISO PŁYTY

POLECAMY INNE NASZE KSIĄŻKI

50 SUPERFUNKCJI EXCELA

Kompletny kurs najważniejszych funkcji i formuł Excela w 50 praktycznych wskazówkach krok po kroku. Na DVD: pliki szkoleniowe oraz wideoporadniki.

NAJLEPSZE WSKAZÓWKI

Jubileuszowa 100. książka ze 100 najlepszymi sposobami na usprawnienie działania komputera, systemu i najważniejszych programów. Na DVD: narzędzia pokazane we wskazówkach.

Nasze książki kupisz na www.literia.pl/ksiazki
lub w dziale prenumeraty, tel. 22 336 79 01
Książki są również dostępne w wersji elektronicznej na www.ksplus.pl

POZNAJ PYTHONA!

Dlaczego warto poznać Pythona? Bo Python to język programowania obecnie najbardziej ceniony przez programistów i najszybciej zdobywający popularność. Jest wykorzystywany do tworzenia takich usług, jak Google, Facebook, Netflix, Spotify, Instagram czy Dropbox, a także między innymi do obróbki danych oraz programowania sztucznej inteligencji.

W dodatku wyróżnia się bardzo prostą składnią, świetnie nadaje się więc do tego, by ucząc się go, zacząć przygodę z programowaniem. I warto przy nim zostać, pogłębiając zdobytą dzięki tej książce wiedzę – to inwestycja w przyszłość.

W tej książce poznamy podstawowe polecenia i instrukcje Pythona oraz przećwiczymy je krok po kroku, tworząc własne gry z grafiką 2D. Dowiemy się też, jak korzystać z bibliotek, jak instalować dodatkowe moduły i jak tworzyć projekty z grafiką 3D. Książka zawiera również zadania do samodzielnego wykonania wraz z rozwiązaniami oraz mnóstwo przykładowych skryptów pokazujących działanie Pythona.

Na DVD znajdziemy zestaw startowy niezbędnych narzędzi dla programistów: najlepsze środowiska programistyczne, edytory kodu źródłowego i pliki szkoleniowe do wskazówek przedstawionych w książce.

CENA 16,90 zł
w tym 5% VAT

Nr 3/2019 (102)