РОЗВ’ЯЗКИ ДО ЗАВДАНЬ З ПРОГРАМУВАННЯ

Ось розв’язки до завдань з програмування, що розміщені укінці розділів. Часто завдання можна розв’язати кількома способами, тож той, що вдасться знайти тобі, може не збігатися з тим, який поданий тут, але приклади дадуть тобі уявлення про можливі підходи.

РОЗДІЛ 3

№1: УЛЮБЛЕНЕ

Ось розв'язок з трьома улюбленими хобі й трьома улюбленими стравами:

```
>>> ігри = ['Pokemon', 'LEGO Mindstorms', 'Волейбол']
>>> їжа = ['Млинці', 'Шоколад', 'Яблука']
>>> улюблене = ігри + їжа
>>> print(улюблене)
['Pokemon', 'LEGO Mindstorms', 'Волейбол', 'Млинці', 'Шоколад',
'Яблука']
```

№2: ПІДРАХУНОК ВОЇНІВ

Цей обрахунок можна виконати кількома різними способами. Ми маємо три будівлі, на дахах кожної з них ховається по 25 ніндзь, і два тунелі, у кожному з яких зачаїлося по 40 самураїв. Ми можемо вирахувати загальну кількість ніндзь, а тоді загальну кількість самураїв, і додати ці два числа:

```
>>> 3*25
75
>>> 2*40
80
>>> 75+80
155
```

Значно коротше (і краще) скомбінувати ці три вирази, застосовуючи дужки (дужки не обов'язкові, завдяки порядку виконання математичних операцій, але з ними рівняння легше читати):

```
>>> (3*25)+(2*40)
155
```

Але, можливо, гарніша програма мовою Python буде схожа на код унизу, який розповідає, що саме ми рахуємо:

```
>>> дахи = 3
>>> ніндзі_на_даху = 25
>>> тунелі = 2
>>> самураїв_на_тунель = 40
>>> print((дахи * ніндзі_на_даху) + (тунелі * самураїв_на_тунель))
155
```

№3: ВІТАННЯ!

У цьому розв'язку ми даємо змінним змістовні назви, а тоді застосовуємо у стрічці тимчасові замінники (`%s %s`), щоб підставити значення цих змінних у текст:

```
>>> ймення = 'Брандо'
>>> прізвище = 'Ікетт'
>>> print('Привіт, %s %s!' % (ймення, прізвище))
Привіт, Брандо Ікетт!
```

РОЗДІЛ 4

№1: ПРЯМОКУТНИК

Намалювати прямокутник можна майже так само, як квадрат, крім того, що «черепаха» має намалювати дві сторони довшими, ніж інші дві:

```
>>> import turtle
>>> t = turtle.Pen()
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(50)
```

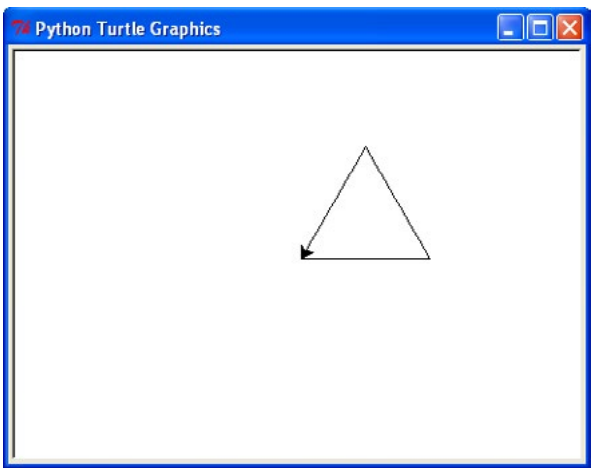
№2: ТРИКУТНИК

У завданні не сказано, який саме трикутник треба намалювати. Бувають три різновиди трикутників: *рівносторонні*, *рівнобедрені* та *різносторонні*. Складно оцінити, які знання ти маєш із геометрії, можливо тобі вже доводилося малювати різні трикутники, у цьому завданні спробуй побавитись із кутами, доки не вийде правильна фігура.

Для цього прикладу зосередьмося на перших двох типах, бо їх намалювати найлегше. Рівносторонній трикутник має три однакові сторони й три однакові кути:

```
>>> import turtle
>>> t = turtle.Pen()
❶ >>> t.forward(100)
❷ >>> t.left(120)
❸ >>> t.forward(100)
❹ >>> t.left(120)
❺ >>> t.forward(100)
```

Ми малюємо основу трикутника, пересуваючись на 100 пікселів уперед під ❶. Ми повертаємося ліворуч на 120 градусів (це створює внутрішній кут 60 градусів) під ❷, а тоді знову пересуваємося вперед на 100 пікселів під ❸. Наступний поворот — теж на 120 градусів під ❹, і «черепаха» повертається назад на початкову позицію, пересуваючись уперед ще на 100 пікселів під ❺. Ось результат запуску цього коду:



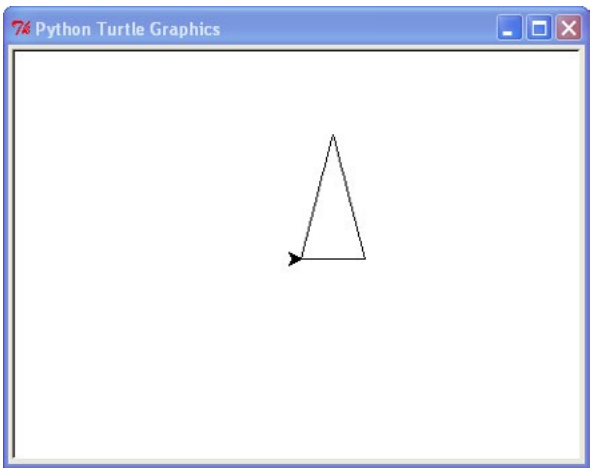
Рівнобедрений трикутник має дві однакові сторони й два однакові кути:

```
>>> import turtle
>>> t = turtle.Pen()
>>> t.forward(50)
>>> t.left(104.47751218592992)
>>> t.forward(100)
>>> t.left(151.04497562814015)
>>> t.forward(100)
```

У цьому розв'язку черепаха пересувається вперед на 50 пікселів, а тоді повертається на 104.47751218592992 градусів. Вона пересувається вперед на 100 пікселів, а після цього повертається на 151.04497562714015 градусів, а тоді знову на 100 пікселів уперед. Щоб повернути черепаху так, щоб вона лицем дивилася на свою початкову позицію, ми можемо знову викликати такий рядок:

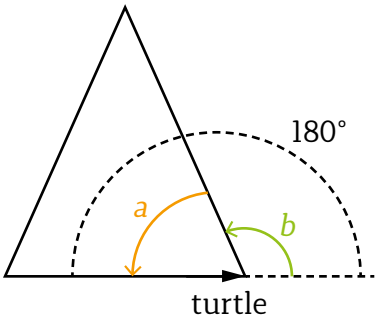
```
>>> t.left(104.47751218592992)
```

Ось результат запуску цього коду:

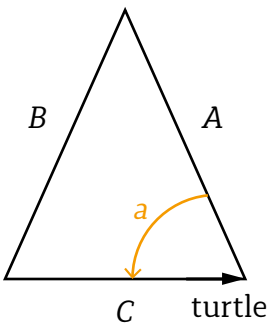


Звідки ми взяли кути в 104.47751218592992 градуси та 151.04497562814015 градусів? Бо ж ці числа досить дивні!

Коли ми визначилися з довжиною кожної сторони трикутника, ми можемо вирахувати внутрішні кути, застосовуючи Python і дрібку тригонометрії. На схемі нижче ти бачиш, що, знаючи градус кута *a*, ми можемо вирахувати градуси (зовнішнього) кута *b*, на який має повернути «черепаха». Два кути *a* та *b* мають в сумі давати 180 градусів.



Нескладно вирахувати внутрішній кут, якщо знаєш потрібну формулу. Наприклад, скажімо, ми хочемо намалювати трикутник із основою завдовжки 50 пікселів (назвімо цю сторону C) і двома сторонами A та B, обидві завдовжки 100 пікселів.



Формула для підрахунку внутрішнього кута *a*, застосовуючи сторони A, B та C, така:

$$a = \arccos\left(\frac{A^2 + C^2 - B^2}{2AC}\right)$$

Ми можемо створити маленьку програму мовою Python, щоб вирахувати значення, застосовуючи модуль Python під назвою `math`:

```
>>> import math
>>> A = 100
>>> B = 100
>>> C = 50
❶ >>> a = math.acos((math.pow(A,2) + math.pow(C,2) - \
        math.pow(B,2)) / (2*A*C))
>>> print(a)
1.31811607165
```

Спочатку ми імпортуємо (`import`) модуль `math`, а тоді створюємо змінні для кожної сторони (`A`, `B` та `C`). Під ❶ ми застосовуємо функцію `math` під назвою `acos` (арккосинус), щоб вирахувати кут. Цей обрахунок повертає значення радіана 1.31811607165. Радіани — це ще одна одиниця вимірювання кутів, як градуси.

ПРИМІТКА *Обернена коса риска (\) у рядку під ❶ не є частиною рівняння — косі риски, як пояснено в Розділі 16, застосовуються для розділення довгих рядків коду. Вони не є необхідними, але в цьому разі ми розділяємо довгий рядок, інакше він не поміститься на сторінку.*

Значення радіанів можна конвертувати в градуси за допомогою функції `degrees` (*градуси*), і ми можемо вирахувати зовнішній кут (значення, на яке ми маємо сказати «черепасі» повернутися), віднявши це значення від 180 градусів:

```
>>> print(180 - math.degrees(a))
104.477512186
```

Рівняння для наступного повороту «черепахи» ось таке:

$$b = \arccos\left(\frac{A^2 + B^2 - C^2}{2AB}\right)$$

Код для цього рівняння теж схожий:

```
>>> b = math.acos((math.pow(A,2) + math.pow(B,2) - \
        math.pow(C,2)) / (2*A*B))
>>> print(180 - math.degrees(b))
151.04497562814015
```

Звісно, ти не маєш застосовувати рівняння, щоб вирахувати кути. Також можна спробувати погратися з кутами, на які повертається «черепаха», доки не отримаєш більш-менш правильну фігуру.

№3: КОРОБКА БЕЗ КУТІВ

Розв'язок цього завдання (намалювати прямокутник без чотирьох кутів, або ж восьмикутник без чотирьох сторін) полягає в тому, щоб зробити одне й те ж саме чотири рази поспіль. Пересунутися вперед, повернути ліворуч на 45 градусів, підняти ручку, пересунутися вперед, опустити ручку й знову повернутися на 45 градусів:

```
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
```

Тож остаточний набір команд буде схожий на наступний код (найкращий спосіб його запустити — це створити новий файл в IDLE, а тоді зберегти його під назвою *nocorners.py*):

```
import turtle
t = turtle.Pen()
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
t.forward(50)
```

```
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
```

РОЗДІЛ 5

№1: ТИ БАГАТИЙ?

Власне, ти отримаєш *помилку відступу*, коли дійдеш до останнього рядка цієї інструкції `if`:

```
>>> гроші = 2000
>>> if гроші > 1000:
❶         print("Я багатий!!")
else:
          print("Я не багатий!!")
❷         print("А колись може й буду...")
SyntaxError: unexpected indent
```

Ця помилка стається, бо перший блок під ❶ починається з чотирьох пробілів, тож оболонка Python не очікує побачити два додаткові пробіли в останньому рядку під ❷. Вона виділяє кольором те місце, де бачить проблему, за допомогою вертикального прямокутного блока, щоб було видно, де сталася помилка.

№2: ПЕЧИВКА!

Код для перевірки кількості печивок має виглядати так:

```
>>> печивка = 600
>>> if печивка < 100 or печивка > 500:
        print('Забагато чи замало')
Забагато чи замало
```

№3: ПРАВИЛЬНЕ ЧИСЛО

Можна написати більше однієї інструкції `if` для перевірки того, чи сума грошей — між 100 та 500 чи 1000 та 5000, але, застосовуючи ключові слова `and` та `or`, ми можемо це зробити в одній інструкції:

```
if (гроші >= 100 and гроші <= 500) or \
            (гроші >= 1000 and гроші <= 5000):
    print('сума між 100 і 500 чи між 1000 і 5000')
```

(Не забудь поставити дужки навколо перших двох і останніх двох умов*, щоб Python перевірив, чи сума між 100 і 500 *чи* між 1000 та 5000).

Ми можемо протестувати цей код, присвоюючи змінній «гроші» різні значення:

```
>>> гроші = 800
>>> if (гроші >= 100 and гроші <= 500) or \
            (гроші >= 1000 and гроші <= 5000):
            print('сума між 100 і 500 чи між 1000 і 5000')
>>> гроші = 400
>>> if (гроші >= 100 and гроші <= 500) or \
            (гроші >= 1000 and гроші <= 5000):
            print('сума між 100 і 500 чи між 1000 і 5000')
сума між 100 і 500 чи між 1000 і 5000
```

* Оскільки програмістам нерідко доводиться перевіряти, чи перебуває значення в межах певного діапазону, існує також скорочений запис. Наприклад, якщо ми хочемо перевірити, чи значення змінної `гроші` між 100 і 500, можемо написати `100 <= гроші <= 500`.

```
>>> гроші = 3000
>>> if (гроші >= 100 and гроші <= 500) or \
        (гроші >= 1000 and гроші <= 5000):
        print('сума між 100 і 500 чи між 1000 і 5000')
сума між 100 і 500 чи між 1000 і 5000
```

№4: Я МОЖУ ПОДОЛАТИ ЦИХ НІНДЗЬ!

Це трохи хитра, підступна задача. Якщо створити інструкцію `if` у тому самому порядку, як це вказано у завданні, то очікуваних результатів не вийде. Ось приклад:

```
>>> ніндзя = 5
>>> if ніндзя < 50:
        print("Це забагато")
elif ніндзя < 30:
        print("Буде важко, але я впораюся")
elif ніндзя < 10:
        print("Я можу подолати цих ніндзь!")

Це забагато
```

Хоч кількість ніндзь й менша за 10, ти отримуєш повідомлення «Це забагато». Це тому, що перша умова (`< 50`) *оцінюється* першою (іншими словами, Python перевіряє її першою) і через те, що значення змінної і справді менше за 50, програма друкує повідомлення, яке ти не очікуєш побачити.

Щоб усе працювало правильно, зміни порядок перевірки числа, щоб спочатку перевірити, чи це число менше за 10:

```
>>> нінзя = 5
>>> if ніндзя < 10:
        print("Я можу подолати цих ніндзь!")
elif ніндзя < 30:
        print("Буде важко, але я впораюся")
elif ніндзя < 50:
        print("Це забагато")

Я можу подолати цих ніндзь!
```

РОЗДІЛ 6

№1: ЦИКЛ «ПРИВІТ»

Команда `print` у цьому `for`-циклі виконується лише раз. Це через те, що коли Python виконує інструкцію `if`, `x` менше за 9, тож він відразу ж розриває цикл.

```
>>> for x in range(0, 20):
        print('привіт %s' % x)
        if x < 9:
            break

привіт 0
```

№2: ПАРНІ ЧИСЛА

Ми можемо застосувати параметр «крок» із функцією `range`, щоб отримати список парних чисел. Якщо тобі 14 років, параметр «початок» буде 2, а «кінець» — 16 (бо `for`-цикл працюватиме до значення, яке передує кінцевому параметру).

```
>>> for x in range(2, 16, 2):
        print(x)
2
4
6
8
10
12
14
```

№3: МОЇ П'ЯТЬ УЛЮБЛЕНИХ ІНГРЕДІЄНТІВ

Є кілька різних способів надрукувати числа з елементами списку. Ось один із них:

```
>>> інгредієнти = ['слимаки', 'хробаки', 'бруд із пупа горили',
                   'брови гусені', 'пальці стоніжки']
```

```
❶ >>> x = 1
❷ >>> for i in інгредієнти:
❸         print('%s %s' % (x, i))
❹         x = x + 1

1 слимаки
2 хробаки
3 бруд із пупа горили
4 брови гусені
5 пальці стоніжки
```

Ми створюємо змінну `x` для зберігання числа, яке хочемо надрукувати, під ❶. Далі ми створюємо `for`-цикл, який пройде через елементи списку під ❷, присвоюючи кожен змінній `i`, і друкуємо значення змінних `x` та `i` під ❸, користуючись тимчасовим замінником `%s`. Ми додаємо 1 до змінної `x` під ❹, тож щоразу, коли виконується цикл, число, яке ми друкуємо, збільшується.

№4: ТВОЯ ВАГА НА МІСЯЦІ

Щоб вирахувати твою вагу в кілограмах на Місяці за 15 років, спочатку створи змінну, яка зберігатиме твою поточну вагу:

```
>>> вага = 30
```

За кожен рік ти можеш вирахувати нову вагу, додаючи кілограм, а тоді множачи на 16.5 відсотків (0.165), щоб отримати вагу на Місяці*:

* Можливо, тебе здивують числа в останніх восьми рядках результату. Адже ти можеш порахувати самостійно, що 45 × 0.165 = 7.425, а не 7.425000000000001. Додаткові 0.000000000000001 — це наслідок того, що комп'ютери не вміють зовсім точно працювати з дійсними числами. Причини цього досить складні і ти навряд чи матимеш багато проблем з такими дрібними похибками. Проте у серйозних програмах — тих, що контролюють космічні ракети чи військову техніку — ці неточності кілька разів призводили до великих збитків і, на жаль, навіть загибелі людей.

```
>>> вага = 30
>>> for рік in range(1, 16):
        вага = вага + 1
        вага_на_місяці = вага * 0.165
        print('Рік %s твоя вага %s' % (рік, вага_на_місяці))

Рік 1 твоя вага 5.115
Рік 2 твоя вага 5.28
Рік 3 твоя вага 5.445
Рік 4 твоя вага 5.61
Рік 5 твоя вага 5.775
Рік 6 твоя вага 5.94
Рік 7 твоя вага 6.105
Рік 8 твоя вага 6.2700000000000005
Рік 9 твоя вага 6.4350000000000005
Рік 10 твоя вага 6.6000000000000005
Рік 11 твоя вага 6.765000000000001
Рік 12 твоя вага 6.930000000000001
Рік 13 твоя вага 7.095000000000001
Рік 14 твоя вага 7.260000000000001
Рік 15 твоя вага 7.425000000000001
```

РОЗДІЛ 7

№1: ВАГА НА МІСЯЦІ

Ця функція має приймати два параметри: `вага` та `приріст` (значення, на яке вага збільшуватиметься щороку). Решта коду дуже схожа на розв'язання Завдання ❹ із Розділу 6.

```
>>> def вага_на_місяці(вага, приріст):
        for рік in range(1, 16):
            вага = вага + приріст
            вага_на_місяці = вага * 0.165
            print('Рік %s твоя вага %s' % (рік, вага_на_місяці))
>>> вага_на_місяці(40, 0.5)

Рік 1 твоя вага 6.6825
Рік 2 твоя вага 6.765
Рік 3 твоя вага 6.8475
Рік 4 твоя вага 6.93
```

```
Рік 5 твоя вага 7.0125
Рік 6 твоя вага 7.095
Рік 7 твоя вага 7.1775
Рік 8 твоя вага 7.26
Рік 9 твоя вага 7.3425
Рік 10 твоя вага 7.425
Рік 11 твоя вага 7.5075
Рік 12 твоя вага 7.59
Рік 13 твоя вага 7.6725
Рік 14 твоя вага 7.755
Рік 15 твоя вага 7.8375
```

№2: ВАГА НА МІСЯЦІ Й РОКИ

Нам потрібна лише невеличка зміна в функції, щоб кількість років можна було задати як параметр.

```
>>> def вага_на_місяці(вага, приріст, роки):
        роки = роки + 1
        for рік in range(1, роки):
            вага = вага + приріст
            вага_на_місяці = вага * 0.165
            print('Рік %s твоя вага %s' % (рік, вага_на_місяці))

>>> вага_на_місяці(35, 0.3, 5)
Рік 1 твоя вага 5.8245
Рік 2 твоя вага 5.874
Рік 3 твоя вага 5.9235
Рік 4 твоя вага 5.973
Рік 5 твоя вага 6.0225
```

Зауваж у другому рядку функції, що ми додаємо 1 до параметра `роки`, щоб `for`-цикл закінчився на правильному році (а не на попередньому).

№3: ПРОГРАМА ВИРАХУВАННЯ ВАГИ НА МІСЯЦІ

Ми можемо скористатися об'єктом `stdin` модуля `sys`, щоб дозволити комусь ввести значення (за допомогою функції `readline`).

Через те, що `sys.stdin.readline` повертає стрічку, ми маємо конвертувати ці стрічки у числа, щоб можна було виконати обрахунок.

```
import sys
def вага_на_місяці():
    print('Введи свою поточну вагу на Землі')
❶   вага = float(sys.stdin.readline())
    print('Введи, наскільки збільшуватиметься твоя вага щороку')
❷   приріст = float(sys.stdin.readline())
    print('Введи кількість років')
❸   роки = int(sys.stdin.readline())
    роки = роки + 1
    for рік in range(1, роки):
        вага = вага + приріст
        вага_на_місяці = вага * 0.165
        print('Рік %s твоя вага %s' % (рік, вага_на_місяці))
```

Під ❶ ми читаємо введені дані за допомогою `sys.stdin.readline`, а тоді конвертуємо стрічку у число з рухомою крапкою, застосовуючи функцію `float`. Це значення зберігається як змінна `вага`. Ми виконуємо той самий процес під ❷ для змінної `приріст`, але застосовуємо функцію `int` під ❸, через те, що ми вводимо лише цілі числа для кількості років (а не дробові числа). Решта коду після цього рядка точнісінько така сама, як у попередньому розв'язку.

Якщо ми викличемо функцію зараз, то побачимо щось на кшталт:

```
>>> вага_на_місяці()
Введи свою поточну вагу на Землі.
45
Введи, наскільки збільшуватиметься твоя вага щороку.
0.4
Введи кількість років.
12
Рік 1 твоя вага 7.491
Рік 2 твоя вага 7.557
Рік 3 твоя вага 7.623
Рік 4 твоя вага 7.689
```

```
Рік 5 твоя вага 7.755
Рік 6 твоя вага 7.821
Рік 7 твоя вага 7.887
Рік 8 твоя вага 7.953
Рік 9 твоя вага 8.019
Рік 10 твоя вага 8.085
Рік 11 твоя вага 8.151
Рік 12 твоя вага 8.217
```

РОЗДІЛ 8

№1: ТАНЕЦЬ ЖИРАФА

Перш ніж додавати функцію, яка примусить Реджинальда затанцювати, погляньмо ще раз на класи `Тварини`, `Ссавці` та `Жирафи`. Ось клас `Тварини` (він був підкласом класу `Живі`, який ми забрали, щоб зробити цей приклад трохи простішим):

```
class Тварини:
    def дихати(self):
        print('дихає')
    def рухатися(self):
        print('рухається')
    def харчуватися_їжею(self):
        print('харчується їжею')
```

Клас `Ссавці` — це підклас класу `Тварини`:

```
class Ссавці(Тварини):
    def годувати_дитинчат_молоком (self):
        print('годує дитинчат')
```

А клас `Жирафи` — це підклас класу `Ссавці`:

```
class Жирафи(Ссавці):
    def їсти_листя_з_дерев(self):
        print('їсть листя')
```

Функцію для пересування кожної ноги додати дуже легко:

```
class Жирафи(Ссавці):
    def їсти_листя_з_дерев(self):
        print('їсть листя')
    def ліву_ногу_вперед(self):
        print('ліва нога вперед')
    def праву_ногу_вперед(self):
        print('права нога вперед')
    def ліву_ногу_назад(self):
        print('ліва нога назад')
    def праву_ногу_назад(self):
        print('права нога назад')
```

Функція `танцювати` просто має викликати кожну з функцій пересування ніг у правильному порядку:

```
    def танцювати(self):
        self.ліву_ногу_вперед()
        self.ліву_ногу_назад()
        self.праву_ногу_вперед()
        self.праву_ногу_назад()
        self.ліву_ногу_назад()
        self.праву_ногу_назад()
        self.праву_ногу_вперед()
        self.ліву_ногу_вперед()
```

Щоб примусити Реджинальда танцювати, ми створюємо об'єкт і викликаємо функцію:

```
>>> реджинальд = Жирафи()
>>> реджинальд.танцювати()

ліву ногу вперед
ліву ногу назад
праву ногу вперед
праву ногу назад
ліву ногу назад
праву ногу назад
праву ногу вперед
ліву ногу вперед
```

№2: ЧЕРЕПАШАЧА ВИДЕЛКА

`Pen` — це клас, визначений у модулі `turtle`, тож ми можемо створити більше одного об'єкта класу `Pen` для кожної з чотирьох «черепах». Якщо ми присвоїмо кожен об'єкт як значення різним змінним, ми зможемо контролювати їх окремо, що спростить відтворення ліній, вказаних у завданні. Концепція того, що кожен *об'єкт* — це незалежна річ, у програмуванні дуже важлива, а надто коли йдеться про класи й об'єкти.

```
import turtle
t1 = turtle.Pen()
t2 = turtle.Pen()
t3 = turtle.Pen()
t4 = turtle.Pen()
t1.forward(100)
t1.left(90)
t1.forward(50)
t1.right(90)
t1.forward(50)
t2.forward(110)
t2.left(90)
t2.forward(25)
t2.right(90)
t2.forward(25)
t3.forward(110)
t3.right(90)
t3.forward(25)
t3.left(90)
t3.forward(25)
t4.forward(100)
t4.right(90)
t4.forward(50)
t4.left(90)
t4.forward(50)
```

Є багато шляхів намалювати те ж саме, тож твій код може дещо відрізнятися.

РОЗДІЛ 9

№1: ЗАГАДКОВИЙ КОД

Функція `abs` повертає абсолютне значення числа, а це означає, що від'ємне число стає додатним. Тож у цьому загадковому коді перша команда `print` друкує 20, а друга — 0.

```
❶ >>> a = abs(10) + abs(-10)
  >>> print(a)
  20

❷ >>> b = abs(-10) + -10
  >>> print(b)
  0
```

Обрахунок під ❶ виявляється `10+10`. Обрахунок під ❷ перетворюється на `10 + (-10)`.

№2: ПРИХОВАНЕ ПОВІДОМЛЕННЯ.

Штука тут у тому, щоб спочатку створити стрічку, яка міститиме повідомлення, а тоді застосувати функцію `dir`, щоб виявити, які функції доступні для цієї стрічки:

```
>>> s = 'Це якщо не ти дуже це хороший читаєш спосіб то заховати
щось повідомлення негаразд'
>>> print(dir(s))
['__add__', '__class__', '__contains__', '__delattr__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__
le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
```

```
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

Глянь на цей список — функція під назвою `split` здається корисною. Ми можемо скористатися функцією `help`, щоб дізнатися, що вона робить:

```
>>> help(s.split)
Help on built-in function split:

split(...) method of builtins.str instance
    S.split(sep=None, maxsplit=-1) -  list of strings

    Return a list of the words in S, using sep as the
    delimiter string. If maxsplit is given, at most maxsplit
    plits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

Згідно з цим описом*, `split` повертає стрічку, розділену на слова, залежно від того, які символи надані у параметрі `sep`. Якщо параметра `sep` немає, функція застосовує пробіл. Отож ця функція розіб'є нашу стрічку.

Тепер, коли ми знаємо, яку функцію застосувати, ми можемо пройтися в циклі словами зі стрічки. Є кілька різних способів надрукувати слова через одне (починаючи з першого). Ось один із них:

```
❶ >>> повідомлення = 'Це якщо не ти дуже це хороший читаєш спосіб
  то заховати щось повідомлення негаразд'
❷ >>> слова = повідомлення.split()
❸ >>> for x in range(0, len(слова), 2):
❹         print(слова[x])
```

* Дослівний переклад довідки: Повертає список слів у `S`, використовуючи `sep` як стрічку-роздільник. Якщо задане значення `maxsplit`, відбувається не більше `maxsplit` розділень. Якщо `sep` не вказаний або становить `None`, будь-який пробільний символ використовується як роздільник, а пусті стрічки вилучаються із результату.

Ми створюємо стрічку під ❶, а під ❷ застосовуємо функцію `split`, щоб розділити стрічку на список окремих слів. Далі ми створюємо `for`-цикл, користуючись функцією `range` під ❸. Перший параметр цієї функції — це 0 (початок списку); наступний параметр застосовує функцію `len`, щоб визначити довжину списку (це буде кінець діапазону); а останній параметр — це значення кроку 2 (тож діапазон чисел буде виглядати так: 0, 2, 4, 6, 8 і так далі). Ми застосовуємо змінну `x` з нашого `for`-циклу, щоб надрукувати значення зі списку під ❹.

№3: КОПІЮВАННЯ ФАЙЛУ

Щоб скопіювати файл, ми його відкриваємо, а тоді читаємо його вміст у змінну. Ми відкриваємо цільовий файл для писання (застосовуючи параметр `'w'` — writing, писання/письмо), а тоді записуємо вміст змінної. Кінцевий код виглядає так:

```
f = open('test.txt')
s = f.read()
f.close()
f = open('output.txt', 'w')
f.write(s)
f.close()
```

Цей приклад працює, але є кращий спосіб скопіювати файл — за допомогою модуля Python під назвою `shutil`:

```
import shutil
shutil.copy('test.txt', 'output.txt')
```

РОЗДІЛ 10

№1: СКОПІЙОВАНІ МАШИНИ

У цьому коді дві команди `print`, і ми маємо визначити, що саме друкується в кожному разі.

Ось перша:

```
>>> машина_1 = Машина()
❶ >>> машина_1.колеса = 4
❷ >>> машина_2 = машина_1
>>> машина_2.колеса = 3
>>> print(машина_1.колеса)

3
```

Чому результат команди `print` — 3, коли ми чітко задали 4 колеса для `машина_1` під ❶? Бо під ❷ обидві змінні `машина_1` і `машина_2` вказують на той самий об'єкт.

Далі, як щодо другої команди `print`?

```
>>> машина_3 = copy.copy(машина_1)
>>> машина_3.колеса = 6
>>> print(машина_1.колеса)

3
```

У цьому разі `машина_3` — це копія об'єкта; вона не позначає той самий об'єкт, що й `машина_1` і `машина_2`. Тож коли ми вказуємо кількість коліс як 6, це ніяк не впливає на колеса в `машина_1`.

№2: ЗБЕРЕЖЕНІ УЛЮБЛЕНІ РЕЧІ

Ми застосовуємо модуль `pickle`, щоб зберегти вміст змінної (чи змінних) у файл:

```
❶ >>> import pickle
❷ >>> улюблене = ['PlayStation', 'іриска', 'фільми',
        'Python для дітей']
❸ >>> f = open('favorites.dat', 'wb')
❹ >>> pickle.dump(улюблене, f)
>>> f.close()
```

Ми імпортуємо модуль `pickle` під ❶ і створюємо наш список улюблених речей під ❷. Далі ми відкриваємо (`open`) файл під назвою *favorites.dat*, задаючи стрічку `'wb'` як другий параметр під ❸ (він означає write-binary — бінарний (нетекстовий) запис).

Далі ми застосовуємо функцію `dump` модуля `pickle`, щоб зберегти вміст змінної `улюблене` у файл під ❹.

Друга частина цього розв'язку полягає в тому, щоб знову прочитати файл. Припускаючи, що оболонку було закрито й знову відкрито, ми маємо імпортувати модуль `pickle` знову.

```
>>> import pickle
>>> f = open('favorites.dat', 'rb')
>>> улюблене = pickle.load(f)
>>> print(улюблене)

['PlayStation', 'іриска', 'фільми', 'Python для дітей']
```

Цей код схожий на інший, крім того, що ми відкрили файл з параметром `'rb'` (що означає read-binary) і застосували функцію `load` модуля `pickle`.

РОЗДІЛ 11

№1: МАЛЮВАННЯ ВОСЬМИКУТНИКА

Восьмикутник має вісім сторін, тож для цього малюнка нам потрібен щонайменше `for`-цикл. Якщо ти на мить замислишся над напрямком «черепахи» і тим, що вона має зробити, малюючи восьмикутник, то збагнеш, що стрілочка «черепахи» обернеться повним колом, як стрілка годинника, на той час, коли вона закінчить малюнок. А це означає, що вона обернеться на всі 360 градусів. Якщо поділити 360 на кількість сторін восьмикутника, ми отримаємо кількість градусів кута, на який «черепаха» має повернути після кожного етапу циклу (45 градусів, як сказано у підказці).

```
>>> import turtle
>>> t = turtle.Pen()
>>> def восьмикутник(сторона):
        for x in range(1, 9):
            t.forward(сторона)
            t.right(45)
```

Ми можемо викликати цю функцію, щоб випробувати її, взявши 100 за розмір однієї зі сторін:

```
>>> восьмикутник(100)
```

№2: МАЛЮВАННЯ ЗАФАРБОВАНОГО ВОСЬМИКУТНИКА

Якщо ми змінимо функцію так, щоб вона малювала зафарбований восьмикутник, нам буде складніше намалювати контури. Кращий підхід полягає в тому, щоб задати параметр для контролю того, чи має бути заповнений восьмикутник.

```
>>> import turtle
>>> t = turtle.Pen()
>>> def восьмикутник(сторона, заповнити):
❶       if заповнити == True:
❷           t.begin_fill()
        for x in range(1, 9):
            t.forward(сторона)
            t.right(45)
❸       if заповнити == True:
❹           t.end_fill()
```

Спочатку ми перевіряємо, чи параметру `заповнити` присвоєне значення `True` під ❶. Якщо так, то ми кажемо «черепасі» почати замальовувати за допомогою функції `begin_fill` під ❷. Далі в наступних двох рядках ми малюємо восьмикутник, так само, як у Завданні ❶, а тоді перевіряємо, чи значення параметра `заповнити` — `True` під номером ❸. Якщо так, ми викликаємо функцію `end_fill` під ❹, яка саме й зафарбовує нашу фігуру.

Ми можемо випробувати цю функцію, задавши перу жовтий колір і викликаючи функцію зі значенням `True` для другого параметра (щоб вона зафарбувала). Далі ми можемо повернути колір знову на чорний і викликати функцію знову, задавши параметру значення `False` для створення контурів.

```
>>> t.color(1, 0.85, 0)
>>> восьмикутник(40, True)
>>> t.color(0, 0, 0)
>>> восьмикутник(40, False)
```

№3: ЩЕ ОДНА ФУНКЦІЯ МАЛЮВАННЯ ЗІРКИ

Штука з цією функцією для малювання зірки полягає в тому, щоб розділити 360 градусів на кількість вершин, що дасть нам внутрішній кут для кожної вершини (дивися рядок ❶ у коді внизу). Щоб визначити зовнішній кут, ми віднімаємо це число від 180, щоб отримати кількість градусів, на які «черепаха» має повернути під ❹.

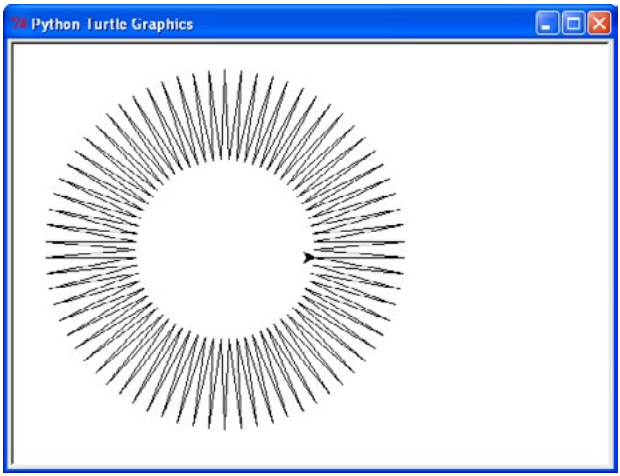
```
import turtle
t = turtle.Pen()
def намалювати_зірку(промінь, вершин):
❶    кут = 360 / вершин
❷    for x in range(0, вершин):
❸        t.forward(промінь)
❹        t.left(180 - кут)
❺        t.forward(промінь)
❻        t.right(180-(кут * 2))
```

Ми пропускаємо цикл від 0 до кількості вершин під ❷, а тоді переміщаємо «черепаху» вперед на кількість пікселів, вказану в параметрі `промінь` під ❸. Ми повертаємо черепаху на кількість градусів, яку попередньо вирахували під ❹, а тоді знову пересуваємо вперед під ❺, таким чином малюючи перший «промінь» зірки. Щоб переміщатися по колу, малюючи промені, ми маємо збільшувати кут, тож ми множимо вирахуваний кут на два й повертаємо «черепаху» праворуч під ❻.

Для прикладу, викличи цю функцію з 80 пікселями й 70 вершинами.

```
>>> намалювати_зірку(80, 70)
```

Це дасть такий результат:



РОЗДІЛ 12

№1: ЗАПОВНИ ЕКРАН ТРИКУТНИКАМИ

Щоб заповнити екран трикутниками, найперше треба облаштувати полотно. Перший крок: задаймо йому ширину й висоту 400 пікселів.

```
>>> from tkinter import *
>>> import random
>>> w = 400
>>> h = 400
>>> tk = Tk()
>>> canvas = Canvas(tk, width=w, height=h)
>>> canvas.pack()
```

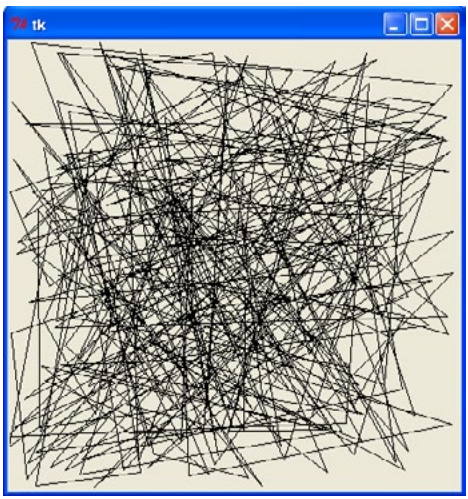
Будь-який трикутник має три вершини, а значить — три набори координат x та y. Ми можемо скористатися функцією `randrange` з модуля `random` (як у разі з випадковим прямокутником з Розділу 12), щоб згенерувати випадкові координати для трьох вершин (всього шість чисел). Далі ми можемо застосувати функцію `random_triangle` (*випадковий трикутник*), щоб намалювати трикутник.

```
>>> def random_triangle():
        p1 = random.randrange(w)
        p2 = random.randrange(h)
        p3 = random.randrange(w)
        p4 = random.randrange(h)
        p5 = random.randrange(w)
        p6 = random.randrange(h)
        canvas.create_polygon(p1, p2, p3, p4, p5, p6, \
                fill="", outline="black")
```

Врешті ми створюємо цикл, щоб намалювати цілу низку випадкових трикутників.

```
>>> for x in range(0, 100):
        random_triangle()
```

З цього виходить щось на кшталт:



Щоб заповнити вікно випадковими зафарбованими трикутниками, спочатку треба створити список кольорів. Ми можемо додати цей код до коду налаштування на початку програми.

```
>>> from tkinter import *
>>> import random
>>> w = 400
```

```
>>> h = 400
>>> tk = Tk()
>>> canvas = Canvas(tk, width=w, height=h)
>>> canvas.pack()
>>> colors = ['red', 'green', 'blue', 'yellow', 'orange',
              'white', 'purple']
```

Далі ми можемо застосувати функцію `choice` модуля `random`, щоб випадково вибрати колір зі списку кольорів і застосувати його, викликаючи `create_polygon`:

```
def random_triangle():
    p1 = random.randrange(w)
    p2 = random.randrange(h)
    p3 = random.randrange(w)
    p4 = random.randrange(h)
    p5 = random.randrange(w)
    p6 = random.randrange(h)
    color = random.choice(colors)
    canvas.create_polygon(p1, p2, p3, p4, p5, p6, \
            fill=color, outline="")
```

Припустімо, що ми знову виконали цикл 100 разів:

```
>>> for x in range(0, 100):
random_triangle()
```

У результаті має вийти щось на кшталт цих трикутників:

№2: ТРИКУТНИК, ЯКИЙ РУХАЄТЬСЯ

Для трикутника, який рухається, ми спочатку знову облаштовуємо полотно, а тоді малюємо трикутник за допомогою функції `create_polygon`:

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=200)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
```

Щоб перемістити трикутник горизонтально по екрану, значення `x` має бути додатним числом, а значення `y` — 0. Ми можемо створити для цього `for`-цикл, застосовуючи 1 як ідентифікатор трикутника, 10 як параметр `x` та 0 як параметр `y`:

```
for x in range(0, 35):
    canvas.move(1, 10, 0)
    tk.update()
    time.sleep(0.05)
```

Пересування екраном униз дуже подібне, зі значенням 0 для параметра `x` і будь-яким додатним значенням параметра `y`:

```
for x in range(0, 14):
    canvas.move(1, 0, 10)
    tk.update()
    time.sleep(0.05)
```

Щоб пересувати фігуру у протилежний бік, нам потрібне від'ємне значення параметра `x` (і, знову, 0 для параметра `y`). Щоб рухатися вгору, нам потрібне від'ємне значення параметра `y`:

```
for x in range(0, 35):
    canvas.move(1, -10, 0)
    tk.update()
    time.sleep(0.05)
```

```
for x in range(0, 14):
    canvas.move(1, 0, -10)
    tk.update()
    time.sleep(0.05)
```

№3: ФОТО, ЯКЕ РУХАЄТЬСЯ

Код для розв'язання задачі з рухливим фото залежить від розміру твого зображення. Зроблю припущення, що твоє зображення називається *face.gif* і воно збережене на диску C:, отже, ти можеш вивести його на екран і переміщати так само, як і інші намальовані фігури.

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
myimage = PhotoImage(file='c:\\face.gif')
canvas.create_image(0, 0, anchor=NW, image=myimage)
for x in range(0, 35):
    canvas.move(1, 10, 10)
    tk.update()
    time.sleep(0.05)
```

Цей код переміщатиме зображення униз по діагоналі.

Якщо ти користуєшся Ubuntu або Mac OS X, назва файлу із зображенням може бути інакша. Якщо файл міститься у твоєму домашньому каталозі, в Ubuntu, завантаження зображення може мати приблизно такий вигляд:

```
myimage = PhotoImage(file='/home/malcolm/face.gif')
```

На Mac завантаження зображення може виглядати десь отак:

```
myimage = PhotoImage(file='/Users/samantha/face.gif')
```

РОЗДІЛ 14

№1: ВІДТЕРМІНУЙ ПОЧАТОК ГРИ

Щоб гра починалася, коли гравець клацає по полотну, ми маємо внести у програму кілька невеличких змін. Перша — додати нову функцію до класу `Paddle`:

```
    def turn_left(self, evt):
        self.x = -2

    def turn_right(self, evt):
        self.x = 2

    def start_game(self, evt):
        self.started = True
```

Коли її викликати, ця функція присвоїть змінній `started` значення `True`. Ми також маємо включити цю змінну об'єкта до функції `__init__` класу `Paddle` (і присвоїти їй значення `False`), а тоді додати прив'язку до події для функції `start_game` (прив'язуючи її до клацання кнопки мишки).

```
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, \
                fill=color)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.canvas_width = self.canvas.winfo_width()
❶       self.started = False
        self.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        self.canvas.bind_all('<KeyPress-Right>', self.turn_right)
❷       self.canvas.bind_all('<Button-1>', self.start_game)
```

Під ❶ ти можеш побачити, що додалася нова змінна `started`, а під ❷ — прив'язку до клацання кнопки мишки.

Кінцеву зміну ми внесемо до останнього циклу в коді. Нам треба перевірити, чи значення змінної `started` `True`, перш ніж малювати м'яч і ракетку, що можна побачити в цій інструкції `if`.

```
while 1:
    if ball.hit_bottom == False and paddle.started == True:
        ball.draw()
        paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

№2: НАПИС «КІНЕЦЬ ГРИ»

Для створення напису «Кінець гри» ми можемо застосувати функцію `create_text`. Ми додамо цей код просто після коду для створення м'яча і ракетки.

```
paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, paddle, 'red')
game_over_text = canvas.create_text(250, 200, \
        text='КІНЕЦЬ ГРИ', state='hidden')
```

Функція `create_text` має іменований параметр під назвою `state`, якому ми присвоюємо значення `'hidden'` (*прихований*). Це означає, що Python створює текст, але робить його невидимим. Щоб вивести текст на екран під кінець гри, ми додаємо інструкцію `if` до циклу внизу коду:

```
while 1:
    if ball.hit_bottom == False and paddle.started == True:
        ball.draw()
        paddle.draw()
❶   if ball.hit_bottom == True:
❷       time.sleep(1)
❸       canvas.itemconfig(game_over_text, state='normal')
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Ми перевіряємо, чи змінній `hit_bottom` присвоєно значення `True` під ❶. Якщо так, програма засинає на 1 секунду під ❷ (щоб зробити коротеньку затримку перед виведенням тексту

на екран), а тоді змінюємо параметр тексту `state` на `'normal'` замість `'hidden'` під ❸, за допомогою функції полотна під назвою `itemconfig`. Ми передаємо цій функції два параметри: ідентифікатор тексту, написаного на полотні (який зберігається у змінній `game_over_text`), та іменований параметр `state`.

№3: ДОДАЙ М'ЯЧЕВІ ПРИСКОРЕННЯ

Ця зміна проста, але може бути складно визначити, де саме в коді її внести. Ми хочемо, щоб м'яч прискорився, якщо він рухається у тому самому горизонтальному напрямку, коли ударяється об ракетку, й уповільнився, якщо рухається у протилежному горизонтальному напрямку.

Найпростіше буде внести цю зміну у функції `hit_paddle` класу `Ball`:

```
    def hit_paddle(self, pos):
        paddle_pos = self.canvas.coords(self.paddle.id)
        if pos[2] >= paddle_pos[0] and pos[0] <= paddle_pos[2]:
❶           if pos[3] >= paddle_pos[1] and pos[3] <= paddle_pos[3]:
❷               self.x += self.paddle.x
                return True
        return False
```

Визначивши, що м'ячик ударився об ракетку під ❶, ми додаємо значення змінної `x` об'єкта-ракетки до змінної `x` м'яча під ❷. Якщо ракетка рухається екраном праворуч (її змінна `x` може мати значення 2, наприклад), і м'яч об неї ударяється, рухаючись праворуч зі значенням `x` 3, м'ячик відстрибне від ракетки з новою (горизонтальною) швидкістю 5. Додавання обох змінних `x` разом означає, що м'ячик отримує нову швидкість, коли ударяється об ракетку.

№4: ЗАПИС РАХУНКУ ГРАВЦЯ

Щоб додати рахунок до нашої гри, ми можемо створити новий клас під назвою `Score` (рахунок):

```
class Score:
    def __init__(self, canvas, color):
```

```
❶         self.score = 0
❷         self.canvas = canvas
❸         self.id = canvas.create_text(450, 10, text=self.score, \
                  fill=color)
```

Функція `__init__` класу `Score` приймає три параметри: `self`, `canvas` і `color`. У першому рядку функції ми створюємо змінну `score` зі значенням 0, під ❶. Ми також зберігаємо параметр `canvas`, щоб скористатися ним пізніше, як змінну об'єкта `canvas` під ❷.

Ми застосовуємо параметр `canvas`, щоб створити наш текст для рахунку, відображаючи його у позиції (450, 10) й зафарбовуючи відповідно до значення параметра `color` під ❸. Текст, який буде зображено на екрані, це поточне значення змінної score (іншими словами, 0).

Клас `Score` потребує ще однієї функції, яку буде використано для збільшення рахунку й зображення нового значення:

```
class Score:
    def __init__(self, canvas, color):
        self.score = 0
        self.canvas = canvas
        self.id = canvas.create_text(450, 10, text=self.score, \
                fill=color)

❶   def hit(self):
❷       self.score += 1
❸       self.canvas.itemconfig(self.id, text=self.score)
```

Функція `hit` не приймає ніяких параметрів під ❶ і просто збільшує рахунок на 1 під ❷, перш ніж застосовувати функцію `itemconfig` об'єкту `canvas` для зміни зображуваного тексту на значення нового рахунку під ❸.

Ми можемо створити об'єкти класу `Score` просто перед тим, як створимо об'єкти `paddle` та `ball`:

```
score = Score(canvas, 'green')
paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, paddle, score, 'red')
game_over_text = canvas.create_text(250, 200, text='GAME OVER', \
        state='hidden')
```

Остання зміна до цього коду — в класі `Ball`. Нам треба зберегти об'єкт `Score` (який ми застосовуємо для створення об'єкту `Ball`), а тоді запустити функцію `hit` у функції м'яча `hit_paddle`.

Початок функції `__init__` класу `Ball` тепер має параметр `score`, який ми застосовуємо для створення змінної об'єкта, яка також називається `score`.

```
    def __init__(self, canvas, paddle, score, color):
        self.canvas = canvas
        self.paddle = paddle
        self.score = score
```

Тепер функція `hit_paddle` має мати такий вигляд:

```
    def hit_paddle(self, pos):
        paddle_pos = self.canvas.coords(self.paddle.id)
        if pos[2] >= paddle_pos[0] and pos[0] <= paddle_pos[2]:
            if pos[3] >= paddle_pos[1] and pos[3] <= paddle_pos[3]:
                self.x += self.paddle.x
                self.score.hit()
                return True
        return False
```

Коли всі чотири завдання виконано, повний ігровий код тепер виглядає так:

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, paddle, score, color):
        self.canvas = canvas
        self.paddle = paddle
        self.score = score
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        random.shuffle(starts)
        self.x = starts[0]
        self.y = -3
```

```
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.hit_bottom = False

    def hit_paddle(self, pos):
        paddle_pos = self.canvas.coords(self.paddle.id)
        if pos[2] >= paddle_pos[0] \
                and pos[0] <= paddle_pos[2]:
            if pos[3] = paddle_pos[1] \
                    and pos[3] <= paddle_pos[3]:
                self.x += self.paddle.x
                self.score.hit()
                return True
        return False

    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if pos[3] >= self.canvas_height:
            self.hit_bottom = True
        if self.hit_paddle(pos) == True:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3

class Paddle:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, \
                fill=color)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.canvas_width = self.canvas.winfo_width()
        self.started = False
        self.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        self.canvas.bind_all('<KeyPress-Right>', \
                self.turn_right)
        self.canvas.bind_all('<Button-1>', self.start_game)
```

```
        def draw(self):
            self.canvas.move(self.id, self.x, 0)
            pos = self.canvas.coords(self.id)
            if pos[0] <= 0:
                self.x = 0
            elif pos[2] >= self.canvas_width:
                self.x = 0

        def turn_left(self, evt):
            self.x = -2

        def turn_right(self, evt):
            self.x = 2

        def start_game(self, evt):
            self.started = True

    class Score:
        def __init__(self, canvas, color):
            self.score = 0
            self.canvas = canvas
            self.id = canvas.create_text(450, 10, text=self.score, \
                    fill=color)

        def hit(self):
            self.score += 1
            self.canvas.itemconfig(self.id, text=self.score)

    tk = Tk()
    tk.title("Гра")
    tk.resizable(0, 0)
    tk.wm_attributes("-topmost", 1)
    canvas = Canvas(tk, width=500, height=400, bd=0,
    highlightthickness=0)
    canvas.pack()
    tk.update()

    score = Score(canvas, 'green')
    paddle = Paddle(canvas, 'blue')
    ball = Ball(canvas, paddle, score, 'red')
    game_over_text = canvas.create_text(250, 200, \
            text='КІНЕЦЬ ГРИ', state='hidden')
```

```
while 1:
    if ball.hit_bottom == False and paddle.started == True:
        ball.draw()
        paddle.draw()
    if ball.hit_bottom == True:
        time.sleep(1)
        canvas.itemconfig(game_over_text, state='normal')
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

РОЗДІЛ 16

№1: ШАХІВНИЦЯ

Щоб намалювати тло у вигляді шахівниці, нам треба змінити цикли у функції гри `__init__`, отак:

```
        self.bg = PhotoImage(file="background.gif")
        w = self.bg.width()
        h = self.bg.height()
❶       draw_background = 0
        for x in range(0, 5):
            for y in range(0, 5):
❷               if draw_background == 1:
❸                   self.canvas.create_image(x * w, y * h, \
                            image=self.bg, anchor='nw')
❹                   draw_background = 0
❺               else:
❻                   draw_background = 1
```

Під ❶ ми створюємо змінну під назвою `draw_background` і задаємо їй значення 0. Під ❷ ми перевіряємо, чи значення змінної становить 1, і якщо так, малюємо фонове зображення під ❸, а тоді знову присвоюємо змінній значення 0 під ❹. Якщо її значення не 1 (а це `else` під ❺), ми задаємо їй значення 1 під ❻.

Що робить ця зміна коду? Коли ми вперше дійдемо до інструкції `if`, ми не намалюємо зображення, а змінній `draw_background` присвоїмо значення 1. Коли ми дійдемо до цієї

інструкції наступного разу, то вже намалюємо зображення, а змінній повернемо значення 0. Після проходження кожного циклу ми поперемінно змінюємо значення змінної. Одного разу ми малюємо зображення; наступного разу — ні.

№2: ШАХІВНИЦЯ З ДВОМА ЗОБРАЖЕННЯМИ

Коли ти уже знаєш, як намалювати шахівницю, виконати два змінних зображення замість одного зображення й одного порожнього квадратика досить просто. Нам треба завантажити нове фонове зображення, а також оригінал. У наступному прикладі ми завантажуємо наше нове зображення *background2.gif* (тобі треба буде спочатку намалювати його в GIMP) і зберігаємо його як змінну об'єкта `bg2`.

```
        self.bg = PhotoImage(file="background.gif")
        self.bg2 = PhotoImage(file="background2.gif")
        w = self.bg.width()
        h = self.bg.height()
        draw_background = 0
        for x in range(0, 5):
            for y in range(0, 5):
                if draw_background == 1:
                    self.canvas.create_image(x * w, y * h, \
                            image=self.bg, anchor='nw')
                    draw_background = 0
                else:
                    self.canvas.create_image(x * w, y * h, \
                            image=self.bg2, anchor='nw')
                    draw_background = 1
```

У другій частині інструкції `if`, яку ми створили у розв'язку Завдання ❶, ми застосовуємо функцію `create_image`, щоб намалювати на екрані нове зображення.

№3: КНИЖКОВА ПОЛИЦЯ Й ЛАМПА

Щоб намалювати різні варіанти тла, ми можемо почати з коду для нашої шахівниці зі змінними зображеннями, але знову його змінити, щоб завантажити кілька нових зображень, а тоді

розмістити їх на полотні. Для цього прикладу я спочатку скопіював зображення *background2.gif* і намалював на ньому книжкову полицю, зберіг новий малюнок як *shelf.gif*. Далі я зробив іще одну копію *background2.gif*, намалював лампу й назвав нове зображення *lamp.gif*.

```
        self.bg = PhotoImage(file="background.gif")
        self.bg2 = PhotoImage(file="background2.gif")
❶       self.bg_shelf = PhotoImage(file="shelf.gif")
❷       self.bg_lamp = PhotoImage(file="lamp.gif")
        w = self.bg.width()
        h = self.bg.height()
❸       count = 0
        draw_background = 0
        for x in range(0, 5):
            for y in range(0, 5):
                if draw_background == 1:
                    self.canvas.create_image(x * w, y * h, \
                            image=self.bg, anchor='nw')
                    draw_background = 0
                else:
❹                   count = count + 1
❺                   if count == 5:
❻                       self.canvas.create_image(x * w, y * h, \
                            image=self.bg_shelf, anchor='nw')
❼                   elif count == 9:
❽                       self.canvas.create_image(x * w, y * h, \
                            image=self.bg_lamp, anchor='nw')
                    else:
                        self.canvas.create_image(x * w, y * h, \
                             image=self.bg2, anchor='nw')
                    draw_background = 1
```

Ми завантажуємо нові зображення під ❶ та ❷, зберігаючи їх як змінні `bg_shelf` та `bg_lamp` відповідно, а тоді створюємо нову змінну під назвою `count` під ❸. У попередньому розв'язку ми мали інструкцію `if`, де малювали те чи те фонове зображення залежно від значення змінної `draw_background`. Тут ми робимо те ж саме, тільки замість того, щоб просто виводити на екран позмінне зображення, ми збільшуємо значення у змінній

`count`, додаючи 1 (за допомогою `count = count + 1`) під ❹. Відштовхуючись від значення у змінній `count`, ми далі вирішуємо, яке зображення малювати. Під ❺, якщо значення досягнуло 5, ми зображаємо малюнок із полицею під ❻. Під ❼, якщо значення досягнуло 9, ми зображаємо малюнок із лампою під ❽. У протилежному разі ми просто зображаємо позмінне тло, як робили раніше.

РОЗДІЛ 18

№1: «ПЕРЕМОГА!»

Ми можемо додати текст «Перемога!» як змінну класу `Game` у його функції `__init__`:

```
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * w, y * h, \
                        image=self.bg, anchor='nw')
        self.sprites = []
        self.running = True
        self.game_over_text = self.canvas.create_text(250, 250, \
                text='ПЕРЕМОГА!', state='hidden')
```

Щоб зобразити текст під кінець гри, нам просто треба додати блок `else` до функції `mainloop`:

```
    def mainloop(self):
        while 1:
            if self.running == True:
                for sprite in self.sprites:
                    sprite.move()
❶           else:
❷               time.sleep(1)
❸               self.canvas.itemconfig(self.game_over_text, \
                        state='normal')
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)
```

Ми бачимо зміни в рядках від 1 до 3. Ми додаємо умову `else` до інструкції `if` під ❶, і Python виконує цей блок коду, якщо значення змінної `running` (*працює, триває*) більше не є `True` (*Так*). Під ❷ ми даємо кодові команду спати впродовж секунди, щоб текст «Перемога!» не з'явився відразу, а тоді змінюємо стан тексту на `'normal'` під ❸, щоб він з'явився на полотні.

№2: АНІМУВАННЯ ДВЕРЕЙ

Щоб анімувати двері так, щоб вони відчинялися й зачинялися, коли чоловічок до них добігає, ми маємо спочатку змінити клас `DoorSprite`. Замість того, щоб задавати зображення як параметр, тепер сам спрайт завантажить два зображення у функції `__init__`:

```
class DoorSprite(Sprite):
    def __init__(self, game, x, y, width, height):
        Sprite.__init__(self, game)
❶       self.closed_door = PhotoImage(file="door1.gif")
❷       self.open_door = PhotoImage(file="door2.gif")
        self.image = game.canvas.create_image(x, y, \
                image=self.closed_door, anchor='nw')
        self.coordinates = Coords(x, y, x + (width / 2), \
        y + height)
        self.endgame = True
```

Як бачиш, два зображення завантажено у змінні об'єкта під ❶ та ❷. Тепер нам треба змінити код внизу гри, де ми створюємо об'єкт `door`, щоб він більше не намагався взяти зображення за параметр:

```
door = DoorSprite(g, 45, 30, 40, 35)
```

`DoorSprite` потребує дві нові функції: одну для зображення на екрані малюнка з відчиненими дверима, а другу — для зображення малюнка з зачиненими дверима.

```
    def opendoor(self):
❶       self.game.canvas.itemconfig(self.image, image=self.open_door)
```

```
❷         self.game.tk.update_idletasks()

      def closedoor(self):
❸         self.game.canvas.itemconfig(self.image,
                  image=self.closed_door)
          self.game.tk.update_idletasks()
```

Застосовуючи функцію полотна `itemconfig`, ми змінюємо зображений малюнок на той, що зберігається у змінній об'єкта `open_door` під ❶. Ми викликаємо функцію `update_idletasks` об'єкта `tk`, щоб примусово зобразити новий малюнок під ❷. (Якщо ми цього не зробимо, зображення відразу не зміниться). Функція `closedoor` схожа, але зображає малюнок, який зберігається у змінній `closed_door` під ❸.

Наступну нову функцію додано до класу `StickFigureSprite`:

```
      def end(self, sprite):
❶         self.game.running = False
❷         sprite.opendoor()
❸         time.sleep(1)
❹         self.game.canvas.itemconfig(self.image, state='hidden')
❺         sprite.closedoor()
```

Ми присвоюємо змінній гри `running` значення `False` під ❶, а тоді викликаємо функцію `opendoor` параметра `sprite` під ❷. Це насправді об'єкт `DoorSprite`, як ми побачимо у наступній секції коду. Під ❸ ми «спимо» впродовж 1 секунди, а тоді приховуємо чоловічка під ❹ і викликаємо функцію `closedoor` під ❺. Таким чином буде здаватися, що чоловічок вийшов у двері й зачинив їх за собою.

Остання зміна — до функції `move`, яка належить `StickFigureSprite`. У ранішій версії цього коду, коли чоловічок стикався з дверима, ми присвоювали змінній `running` значення `False`, але позаяк це перейшло до функції `end`, ми маємо натомість викликати цю функцію:

```
            if left and self.x < 0 \
                    and collided_left(co, sprite_co):
                self.x = 0
                left = False
```

```
❶                 if sprite.endgame:
❷                     self.end(sprite)
            if right and self.x > 0 \
                    and collided_right(co, sprite_co):
                self.x = 0
                right = False
❸               if sprite.endgame:
❹                   self.end(sprite)
```

У тій секції коду, де ми перевіряємо, чи чоловічок рухається ліворуч і чи він зіткнувся зі спрайтом ліворуч, ми перевіряємо, чи значення змінної `endgame` — `True` під ❶. Якщо так, то ми знаємо, що це об'єкт `DoorSprite`, і під ❷ ми викликаємо функцію `end`, передавши змінну `sprite` параметром. Ми вносимо такі самі зміни у тій секції коду, де перевіряємо, чи чоловічок рухається праворуч і чи він зіткнувся зі спрайтом праворуч (під ❸ і ❹).

№3: РУХЛИВІ ПЛАТФОРМИ

Клас рухливої платформи буде схожий на клас чоловічка. Замість фіксованого набору координат ми маємо наново вираховувати позицію платформи. Ми можемо створити підклас класу `PlatformSprite`, щоб функція `__init__` стала така:

```
class MovingPlatformSprite(PlatformSprite):
❶    def __init__(self, game, photo_image, x, y, width, height):
❷        PlatformSprite.__init__(self, game, photo_image, x, y, \
                width, height)
❸        self.x = 2
❹        self.counter = 0
❺        self.last_time = time.time()
❻        self.width = width
❼        self.height = height
```

Ми задаємо ті ж самі параметри, що й у класі `PlatformSprite` під ❶, а тоді викликаємо функцію `__init__` батьківського класу з цими самими параметрами під ❷. Це означає, що будь-який об'єкт класу `MovingPlatformSprite` буде облаштований точнісінько так само, як об'єкт класу `PlatformSprite`. Далі ми створюємо змінну `x` зі значенням 2 (платформа почне рухатися

праворуч) під ❸, а після неї — змінну `counter` (*лічильник*) під ❹. Ми застосуємо цей лічильник, щоб просигналізувати, коли платформа має змінити напрям руху. Через те, що ми не хочемо, щоб платформа рухалася туди-сюди якомога швидше, так само, як наш `StickFigureSprite` не має рухатися туди-сюди якомога швидше, ми запишемо час у змінній `last_time` під ❺ (цією змінною `last_time` ми скористаємося для уповільнення руху платформи). Останнє доповнення цієї функції — зберегти ширину та висоту під ❻ та ❼.

Наступне доповнення до нашого нового класу — це функція `coords`:

```
        self.last_time = time.time()
        self.width = width
        self.height = height

    def coords(self):
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
❶       self.coordinates.x2 = xy[0] + self.width
❷       self.coordinates.y2 = xy[1] + self.height
        return self.coordinates
```

Функція `coords` — майже така сама, як та, якою ми скористалися для чоловічка, от тільки замість застосовувати фіксовану ширину й довжину, ми застосовуємо значення, які зберігаються у функції `__init__`. (Різницю можна побачити у рядках ❶ і ❷).

Позаяк це рухливий спрайт, ми також маємо додати функцію `move` (*рухатися*):

```
        self.coordinates.x2 = xy[0] + self.width
        self.coordinates.y2 = xy[1] + self.height
        return self.coordinates

    def move(self):
❶       if time.time() - self.last_time > 0.03:
❷           self.last_time = time.time()
❸           self.game.canvas.move(self.image, self.x, 0)
❹           self.counter = self.counter + 1
```

```
❺            if self.counter > 20:
❻                self.x = self.x * -1
❼                self.counter = 0
```

Функція `move` перевіряє, чи час перевищує одну десяту секунди під ❶. Якщо так, ми присвоюємо змінній `last_time` значення поточного часу під ❷. Під ❸ ми переміщаємо малюнок із платформою, а тоді збільшуємо значення змінної `counter` під ❹. Якщо лічильник перевищує 20 (інструкція `if` під ❺), ми змінюємо напрямок руху на протилежний, множачи змінну `x` на `-1` (тож якщо вона від'ємна, стає додатною, а якщо додатна, то стає від'ємною) під ❻, і обнуляємо лічильник під ❼. Тепер платформа рухатиметься в одному напрямку до рахунку 20, а тоді — в протилежному до рахунку 20.

Щоб випробувати рухливі платформи, ми можемо замінити кілька існуючих платформ з `PlatformSprite` на `MovingPlatformSprite`:

```
platform5 = MovingPlatformSprite(g, \
        PhotoImage(file="platform2.gif"), 175, 350, 66, 10)
platform6 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
        50, 300, 66, 10)
platform7 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
        170, 120, 66, 10)
platform8 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
        45, 60, 66, 10)
platform9 = MovingPlatformSprite(g, \
        PhotoImage(file="platform3.gif"), 170, 250, 32, 10)
platform10 = PlatformSprite(g, \
        PhotoImage(file="platform3.gif"), 230, 200, 32, 10)
```

Далі викладено код із усіма змінами:

```
from tkinter import *
import random
import time

class Game:
    def __init__(self):
        self.tk = Tk()
```

```
            self.tk.title("Містер Руки-палички біжить до виходу")
            self.tk.resizable(0, 0)
            self.tk.wm_attributes("-topmost", 1)
            self.canvas = Canvas(self.tk, width=500, height=500, \
                    highlightthickness=0)
            self.canvas.pack()
            self.tk.update()
            self.canvas_height = 500
            self.canvas_width = 500
            self.bg = PhotoImage(file="background.gif")
            w = self.bg.width()
            h = self.bg.height()
            for x in range(0, 5):
                for y in range(0, 5):
                    self.canvas.create_image(x * w, y * h, \
                            image=self.bg, anchor='nw')
            self.sprites = []
            self.running = True
            self.game_over_text = self.canvas.create_text(250, 250, \
                    text='ПЕРЕМОГА!', state='hidden')

        def mainloop(self):
            while 1:
                if self.running:
                    for sprite in self.sprites:
                        sprite.move()
                else:
                    time.sleep(1)
                    self.canvas.itemconfig(self.game_over_text, \
                            state='normal')
                self.tk.update_idletasks()
                self.tk.update()
                time.sleep(0.01)

    class Coords:
        def __init__(self, x1=0, y1=0, x2=0, y2=0):
            self.x1 = x1
            self.y1 = y1
            self.x2 = x2
            self.y2 = y2

    def within_x(co1, co2):
        if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
```

```
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x1):
        return True
    else:
        return False

def within_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y1):
        return True
    else:
        return False

def collided_left(co1, co2):
    if within_y(co1, co2):
        if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
            return True
    return False

def collided_right(co1, co2):
    if within_y(co1, co2):
        if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
            return True
    return False

def collided_top(co1, co2):
    if within_x(co1, co2):
        if co1.y1 <= co2.y2 and co1.y1 >= co2.y1:
            return True
    return False

def collided_bottom(y, co1, co2):
    if within_x(co1, co2):
        y_calc = co1.y2 + y
        if y_calc >= co2.y1 and y_calc <= co2.y2:
            return True
    return False

class Sprite:
    def __init__(self, game):
```

```
        self.game = game
        self.endgame = False
        self.coordinates = None
    def move(self):
        pass
    def coords(self):
        return self.coordinates

class PlatformSprite(Sprite):
    def __init__(self, game, photo_image, x, y, width, height):
        Sprite.__init__(self, game)
        self.photo_image = photo_image
        self.image = game.canvas.create_image(x, y, \
                image=self.photo_image, anchor='nw')
        self.coordinates = Coords(x, y, x + width, y + height)

class MovingPlatformSprite(PlatformSprite):
    def __init__(self, game, photo_image, x, y, width, height):
        PlatformSprite.__init__(self, game, photo_image, x, y, \
                width, height)
        self.x = 2
        self.counter = 0
        self.last_time = time.time()
        self.width = width
        self.height = height

    def coords(self):
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
        self.coordinates.x2 = xy[0] + self.width
        self.coordinates.y2 = xy[1] + self.height
        return self.coordinates

    def move(self):
        if time.time() - self.last_time > 0.03:
            self.last_time = time.time()
            self.game.canvas.move(self.image, self.x, 0)
            self.counter += 1
            if self.counter > 20:
                self.x *= -1
                self.counter = 0
```

```
class DoorSprite(Sprite):
    def __init__(self, game, x, y, width, height):
        Sprite.__init__(self, game)
        self.closed_door = PhotoImage(file="door1.gif")
        self.open_door = PhotoImage(file="door2.gif")
        self.image = game.canvas.create_image(x, y, \
                image=self.closed_door, anchor='nw')
        self.coordinates = Coords(x, y, x + (width / 2), \
                y + height)
        self.endgame = True

    def opendoor(self):
        self.game.canvas.itemconfig(self.image, \
                image=self.open_door)
        self.game.tk.update_idletasks()

    def closedoor(self):
        self.game.canvas.itemconfig(self.image, \
                image=self.closed_door)
        self.game.tk.update_idletasks()

class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
        self.images_left = [
            PhotoImage(file="figure-L1.gif"),
            PhotoImage(file="figure-L2.gif"),
            PhotoImage(file="figure-L3.gif")
        ]
        self.images_right = [
            PhotoImage(file="figure-R1.gif"),
            PhotoImage(file="figure-R2.gif"),
            PhotoImage(file="figure-R3.gif")
        ]
        self.image = game.canvas.create_image(200, 470, \
                image=self.images_left[0], anchor='nw')
        self.x = -2
        self.y = 0
        self.current_image = 0
        self.current_image_add = 1
        self.jump_count = 0
        self.last_time = time.time()
        self.coordinates = Coords()
```

```
        game.canvas.bind_all('<KeyPress-Left>', \
                self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', \
                self.turn_right)
        game.canvas.bind_all('<space>', self.jump)

    def turn_left(self, evt):
        if self.y == 0:
            self.x = -2

    def turn_right(self, evt):
        if self.y == 0:
            self.x = 2

    def jump(self, evt):
        if self.y == 0:
            self.y = -4
            self.jump_count = 0

    def animate(self):
        if self.x != 0 and self.y == 0:
            if time.time() - self.last_time > 0.1:
                self.last_time = time.time()
                self.current_image += self.current_image_add
                if self.current_image >= 2:
                    self.current_image_add = -1
                if self.current_image <= 0:
                    self.current_image_add = 1
        if self.x < 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[self.current_
                        image])
        elif self.x > 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[self.current_image])
```

```
    def coords(self):
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
        self.coordinates.x2 = xy[0] + 27
        self.coordinates.y2 = xy[1] + 30
        return self.coordinates

    def move(self):
        self.animate()
        if self.y < 0:
            self.jump_count += 1
            if self.jump_count > 20:
                self.y = 4
        if self.y > 0:
            self.jump_count -= 1
        co = self.coords()
        left = True
        right = True
        top = True
        bottom = True
        falling = True
        if self.y > 0 and co.y2 >= self.game.canvas_height:
            self.y = 0
            bottom = False
        elif self.y < 0 and co.y1 <= 0:
            self.y = 0
            top = False
        if self.x > 0 and co.x2 >= self.game.canvas_width:
            self.x = 0
            right = False
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            left = False
        for sprite in self.game.sprites:
            if sprite == self:
                continue
            sprite_co = sprite.coords()
            if top and self.y < 0 and collided_top(co, sprite_co):
                self.y = -self.y
                top = False
            if bottom and self.y > 0 and collided_bottom(self.y, \
                    co, sprite_co):
```

```
                    self.y = sprite_co.y1 - co.y2
                    if self.y < 0:
                        self.y = 0
                    bottom = False
                    top = False
                if bottom and falling and self.y == 0 \
                        and co.y2 < self.game.canvas_height \
                        and collided_bottom(1, co, sprite_co):
                    falling = False
                if left and self.x < 0 and collided_left(co, sprite_co):
                    self.x = 0
                    left = False
                    if sprite.endgame:
                        self.end(sprite)
                if right and self.x > 0 \
                        and collided_right(co, sprite_co):
                    self.x = 0
                    right = False
                    if sprite.endgame:
                        self.end(sprite)
            if falling and bottom and self.y == 0 \
                    and co.y2 < self.game.canvas_height:
                self.y = 4
            self.game.canvas.move(self.image, self.x, self.y)

        def end(self, sprite):
            self.game.running = False
            sprite.opendoor()
            time.sleep(1)
            self.game.canvas.itemconfig(self.image, state='hidden')
            sprite.closedoor()

    g = Game()
    platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
            0, 480, 100, 10)
    platform2 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
            150, 440, 100, 10)
    platform3 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
            300, 400, 100, 10)
    platform4 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
            300, 160, 100, 10)
    platform5 = MovingPlatformSprite(g, PhotoImage(file="platform2.gif"),\
            175, 350, 66, 10)
```

```
platform6 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
        50, 300, 66, 10)
platform7 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
        170, 120, 66, 10)
platform8 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
        45, 60, 66, 10)
platform9 = MovingPlatformSprite(g, PhotoImage(file="platform3.gif"),\
        170, 250, 32, 10)
platform10 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
        230, 200, 32, 10)
g.sprites.append(platform1)
g.sprites.append(platform2)
g.sprites.append(platform3)
g.sprites.append(platform4)
g.sprites.append(platform5)
g.sprites.append(platform6)
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
door = DoorSprite(g, 45, 30, 40, 35)
g.sprites.append(door)
sf = StickFigureSprite(g)
g.sprites.append(sf)
g.mainloop()
```