Budget manager

Managing money can be a tedious task, but it can be made easier with a computer. There are various apps for tracking what you spend, based on setting budgets for different kinds of expenses. This project will create a simple budget manager using Python dictionaries and classes.

What the program does

This budget manager will allow users to keep track of their expenditures against an overall budget of **2500**. To start, a budget is allocated for different kinds of expenditures, such as groceries and household bills. The expenses can then be compared against their allocated budget. A summary is displayed to get a quick overview of the finances.

Budget planner
Rather than directly creating a program, this project will create a set of functions that can be called from the Python shell. These functions can also be imported and used in other programs.

Python 3.7.0 Shell

```
>>> add_budget("Groceries", 500)
2000.00
>>> add_budget("Rent", 900)
1100.00
>>> spend("Groceries", 35)
465.00
>>> spend("Groceries", 15)
450.00
>>> print_summary()
Budget            Budgeted      Spent  Remaining
--------------- ---------- ---------- ----------
Groceries           500.00      50.00     450.00
Rent                900.00       0.00     900.00
--------------- ---------- ---------- ----------
Total              1400.00      50.00    1350.00
```

Allocates spend against budget

Returns the budget left

The summary gives an overview of all of the expenses

YOU WILL LEARN

- How to use Python dictionaries
- How to raise exceptions for errors
- How to format strings for output
- How to create a Python class

Time: 1 hour

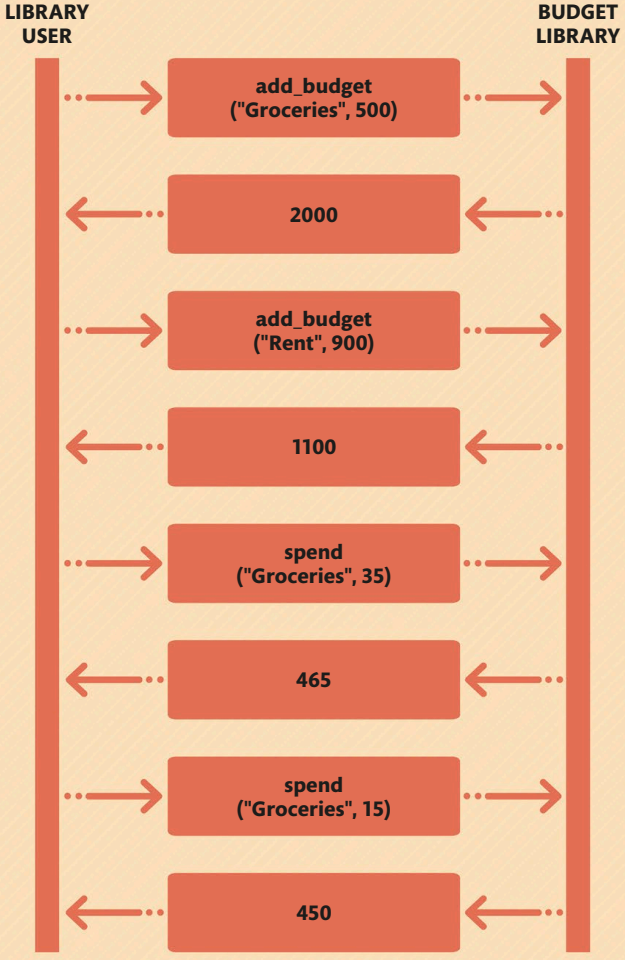
Lines of code: 43

Difficulty level

WHERE THIS IS USED

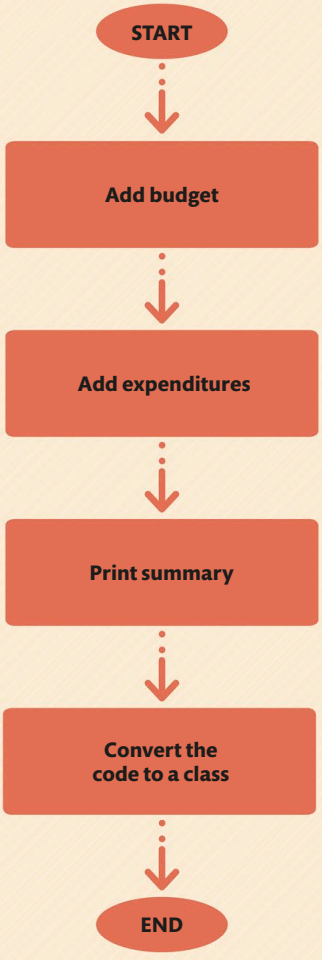
The library developed in this project, with the addition of a user interface, can be used in a simple financial-planning application. Splitting up a program into multiple modules and encapsulating code and data in classes are both techniques that are used extensively in programming.

Function calls
The functions written in this program will allow you to allocate an amount of income to different named budgets. The functions will then track spending against these budgets to see the total amount spent compared to the amount that was budgeted. The diagram below shows an example of making a series of function calls.



Program design

In this project, functions are added sequentially to create the budget library. Next, a summary of all of the expenses is printed. In the end, all the code is converted into a Python class to make it more useful.

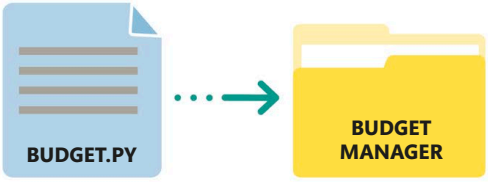


1 Setting up

To create this budget manager, you will need a new Python file. You can then add some basic code to the file and build on it later. The use of Python dictionaries will allow you to save the amount budgeted and spent.

1.1 CREATE A NEW FILE

The first step is to create a new file that will contain the code for this project. Open IDLE and select New File from the File menu. Create a new folder on your desktop called "BudgetManager", and save this empty file inside of it. Name the file "budget.py".



1.2 SET UP THE VARIABLES

Now create some global variables that will track the amount of money available, the amount you have budgeted, and the amount spent. You will use Python dictionaries (see box, below) for the budgets and expenditures that will map from a name, such as "Groceries", to an amount of money. Type this code into the new file.

```
available = 2500.00
budgets = {}
expenditure = {}
```

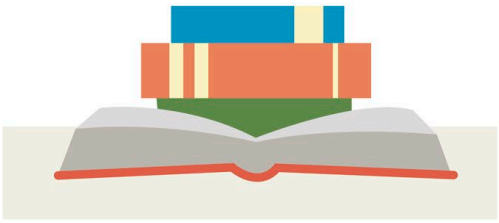
Sets the variable `available` to an example starting amount

Curly brackets are used to create a dictionary—an empty dictionary, in this instance

DICTIONARIES

When deciding what Python data structure to use, think about how you would write the information to be stored. Often, the obvious way to write something down is similar to the table shown below. If the information in the first column is unique (the items in it are not repeated), then using Python dictionaries might be the answer.
A dictionary is a data structure that consists of multiple key:value pairs. It maps one value, such as a name, to another, such as an amount of money. In the table below, the first column contains the keys of the dictionary and the second contains the values. If the table has multiple value columns, then these can be stored in separate dictionaries using the same keys. You can, therefore, have one dictionary for budgets and another for expenditures.

DICTIONARY FORMAT	
Budget name	**Budget amount**
Groceries	500
Bills	200
Entertainment	50

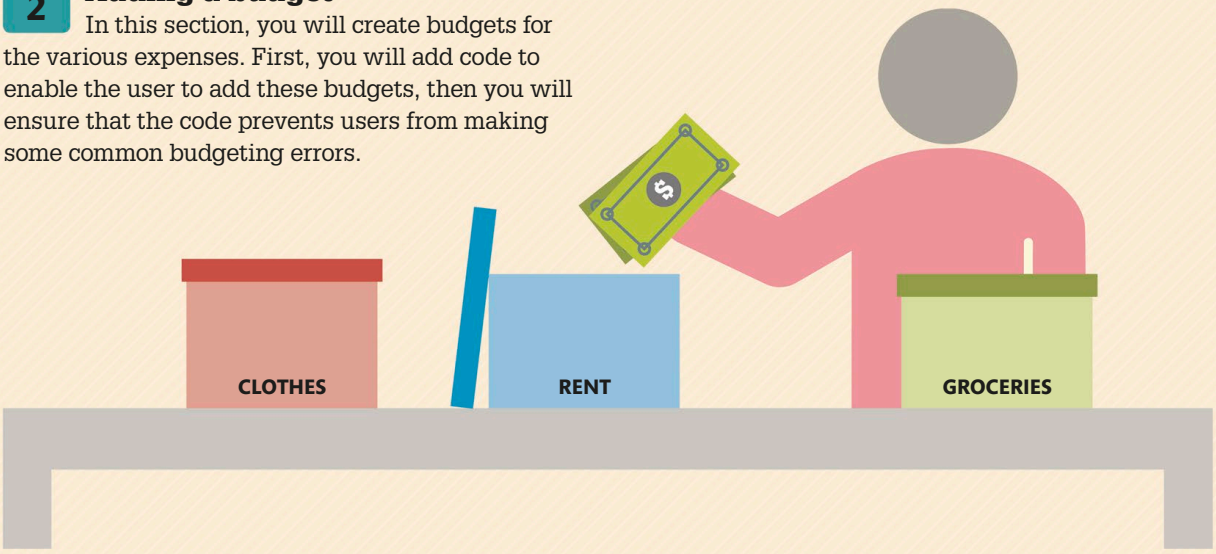


```
{"Groceries": 500, "Bills": 200, "Entertainment": 50}
```

The table information above as a Python dictionary

2 Adding a budget

In this section, you will create budgets for the various expenses. First, you will add code to enable the user to add these budgets, then you will ensure that the code prevents users from making some common budgeting errors.



2.1 ADD A BUDGET FUNCTION

Write a function to add a budget. The function will take the name of the budget and the amount of money to be budgeted. It will then store these in the budgets dictionary and deduct the amount from the amount available. The function then returns the new available amount to show how much is still left to budget. Add this code below the global variables.

```
def add_budget(name, amount):
    global available
    budgets[name] = amount
    available -= amount
    expenditure[name] = 0
    return available
```

`available` will be global when set in this function

Stores the budgeted amount in the `budgets` dictionary

Deducts the budgeted amount from the available amount

Returns the new available amount

Sets the spent amount for this budget to **0**

SAVE

2.2 RUN FILE

Save and then run the file by selecting Run Module from the Run menu. This will open the IDLE Python shell window. You can test the function by typing an example call in the shell. You will see >>> in the window, which is the Python shell prompt. You can type small pieces of Python code next to this and they will be executed when you press Enter or return.

```
>>> add_budget("Groceries", 500)
2000.0
>>> budgets
{'Groceries': 500}
>>> expenditure
{'Groceries': 0}
```

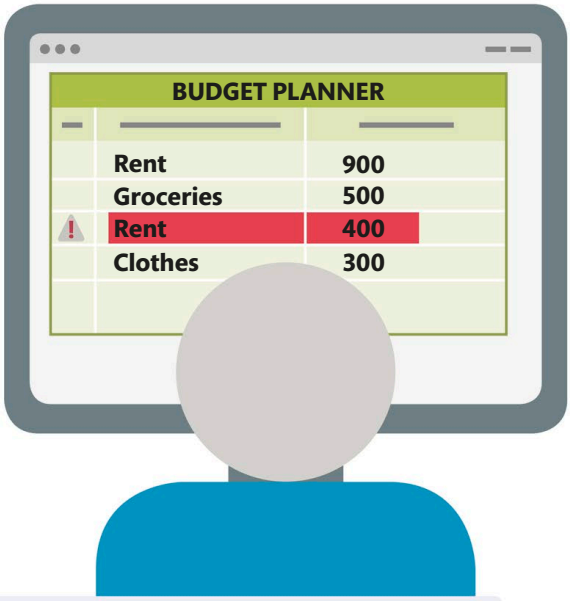
Type this line and press Enter

Returned value of the function call

Typing the name of variables at the prompt will show their values

2.3 ERROR CHECKING

To see what happens if you add a budget twice, type this code in the shell window. You will notice that the budgets dictionary will be updated with the new value, but the `available` amount is reduced by both. To avoid this, you need to add some code to check if the same budget name has been used twice. Add the code shown in the editor window below to make changes to the `add_budget()` function.




```
>>> add_budget("Rent", 900)
1100.0
>>> add_budget("Rent", 400)
700.0
>>> budgets
{'Groceries': 500, 'Rent': 400}
```

The available amount is deducted twice

```
def add_budget(name, amount):
    global available
    if name in budgets:
        raise ValueError("Budget exists")
    budgets[name] = amount
```

Checks if `name` already exists as a key in the budgets dictionary

Leaves the function immediately with an exception if a budget name appears more than once

EXCEPTIONS

In Python, errors are indicated by raising exceptions. These exceptions interrupt the normal execution of code. Unless the exception is caught, the program will immediately exit and display the exception that has been raised and the line of code it occurred at.

There are a number of standard exception types in Python. Each of these accept a string value giving an error message that can be displayed to the user to explain what has gone wrong. The table below lists a few standard exception types and when they should be used.

TYPES OF EXCEPTIONS

Name	Use when
TypeError	A value is not of the expected type—for example, using a string where a number was expected.
ValueError	A value is invalid in some way—for example, too large or too small.
RuntimeError	Some other unexpected error has occurred in the program.

2.4 RUN THE MODULE

Test the code again to check if the error has now been fixed. When you run the code, the three global variables will be set back to their initial values. Type this code in the shell window. You will now get an error message if you try adding the same budget twice. If you check the variables **budgets** and **available**, you will see that they have not been updated with the wrong values.

```
>>> add_budget("Groceries", 500)
2000.0
>>> add_budget("Rent", 900)
1100.0
>>> add_budget("Rent", 400)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    add_budget("Rent", 400)
  File "budget.py", line 7, in add_budget
    raise ValueError("Budget exists")
ValueError: Budget exists
```

Error message displayed onscreen

```
>>> budgets
{'Groceries': 500, 'Rent': 900}
>>> available
1100.0
```

The variables will not be updated with wrong values

2.5 MORE ERROR CHECKING

Continue in the shell window to see what happens if you budget an amount of money that is more than what is available. This is clearly an error, since you should not be able to overbudget. Add another check into the **add_budget()** function to fix this.

Update the code in the editor window. Save the file and run the code once again to test if the new error message is displayed and overbudgeting is prevented.

```
>>> add_budget("Clothes", 2000)
-900.0
```

A negative value indicates overbudgeting

```
    if name in budgets:
        raise ValueError("Budget exists")
    if amount > available:
        raise ValueError("Insufficient funds")
    budgets[name] = amount
```

Checks if the amount being budgeted is more than the amount available

Raises an exception and leaves the function immediately

SAVE

```
>>> add_budget("Groceries", 500)
2000.0
>>> add_budget("Rent", 900)
1100.0
>>> add_budget("Clothes", 2000)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    add_budget("Clothes", 2000)
  File "budget.py", line 9, in add_budget
    raise ValueError("Insufficient funds")
ValueError: Insufficient funds
```

Error message for overbudgeting is displayed

3 Tracking expenditures

Next, you need to add a way to track all of the expenditures. To do this, you will first add a function that allows you to enter the money that has been spent and then add another function to display the summary. This will indicate the total money spent and the amount remaining.

3.1 ADD SPEND FUNCTION

Add a function to note the amount you have spent and the name of the budget that you want to track it against. Add a new **`spend()`** function below the **`add_budget()`** function. The Python "+=" operator is used to add an amount to a variable. Save the file and then run the module to try using this new function.

```
    return available

def spend(name, amount):
    if name not in expenditure:
        raise ValueError("No such budget")
    expenditure[name] += amount
```

Raises an exception if the value of **`name`** is not a key in the **`expenditure`** dictionary

Adds **`amount`** to the corresponding key in the **`expenditure`** dictionary

3.2 RETURNING THE REMAINING AMOUNT

It will also be useful to track the amount left in the budget. Add this code to the end of the **spend()** function you just created, then save and run the file to test the code. You will notice that you can spend more than the budgeted amount. You do not need an exception for this, as you will want to track overspending.

```
    budgeted = budgets[name]
    spent = expenditure[name]
    return budgeted - spent
```

Gets the budgeted amount for **name**

Returns the amount left in the budget

Gets the total amount spent

```
>>> add_budget("Groceries", 500)
2000.0
>>> spend("Groceries", 35)
465
>>> spend("Groceries", 15)
450
>>> spend("Groceries", 500)
-50
```

Negative value indicates that spending exceeds the budget

3.3 PRINT A SUMMARY

In this step, you will add a function that will display an overview of each budget name, the amount originally budgeted, the amount spent, and the amount left to spend (if any). Add this code at the bottom of the file. Save the changes and run the file in the shell window. The summary will display the figures for every category.

```
def print_summary():
    for name in budgets:
        budgeted = budgets[name]
        spent = expenditure[name]
        remaining = budgeted - spent
        print(name, budgeted, spent, remaining)
```

Loops through all the keys in the **budgets** dictionary

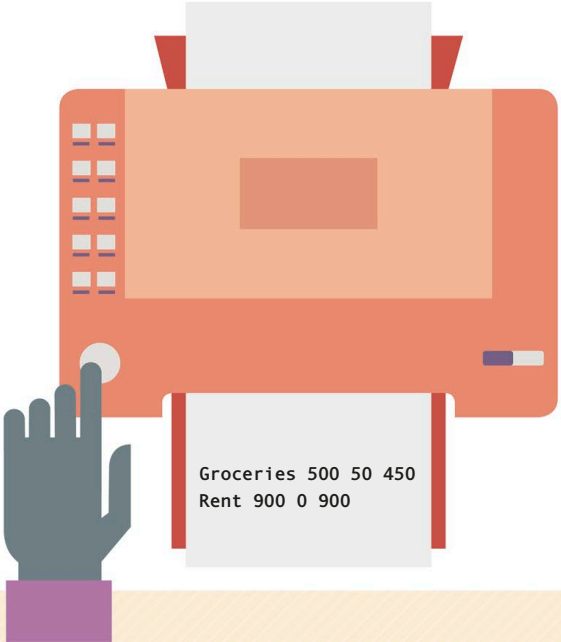
Gets the budgeted amount for the **name** key

Gets the amount spent for the **name** key

Calculates the remaining amount by deducting budgeted from spent

Prints a single line summary for this budget

```
>>> add_budget("Groceries", 500)
2000.0
>>> add_budget("Rent", 900)
1100.0
>>> spend("Groceries", 35)
465
>>> spend("Groceries", 15)
450
>>> print_summary()
Groceries 500 50 450
Rent 900 0 900
```



3.4 FORMAT THE SUMMARY

At this stage, the summary will be a bit hard to read with the numbers squeezed together. To fix this, you can line them up in a table by using "string formatting" (see box, below). Change the `print` line in the `print_summary()` function as shown below. This will create a string from the values, formatting each to a specific width and number of decimal places. It will then print that string.

```
        remaining = budgeted - spent
        print(f'{name:15s} {budgeted:10.2f} {spent:10.2f} '
              f'{remaining:10.2f}')
```

The amount will be displayed with two decimal places

```
>>> add_budget("Groceries", 500)
2000
>>> add_budget("Rent", 900)
1100
>>> spend("Groceries", 35)
465
>>> spend("Groceries", 15)
450
>>> print_summary()
Groceries             500.00      50.00     450.00
Rent                  900.00       0.00     900.00
```

The values will have two decimal places and will be lined up in columns, similar to a table

FORMAT STRINGS

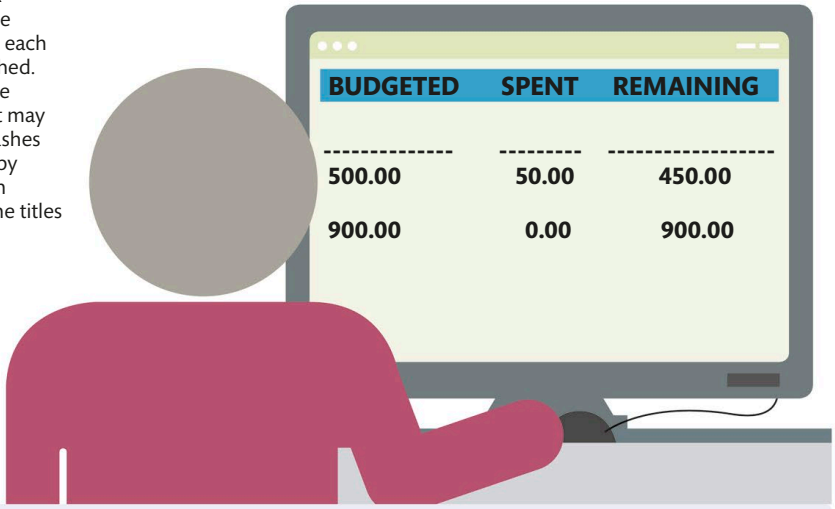
In Python, formatted strings can be created from values with special format strings. These are written like normal strings but have an "f" character before the opening quotation mark. Inside the string, you can place code expressions within curly brackets. These will be executed and replaced with their values. The most common expressions used are variable names, but arithmetic calculations can also be used. Any part of the string outside of the brackets is used without change. Detailed formatting instructions can be added after a colon. This includes a letter specifying how to format the value. Placing a number before this letter allows a width to be specified.

EXAMPLES OF FORMAT STRINGS

Example	Result
f'{greeting} World!'	'Hello World!'
f'{greeting:10s}'	'Hello '
f'{cost:5.2f}'	' 3.47'
f'{cost:5.1f}'	' 3.5'
f'The answer is {a * b}'	'The answer is 42'

3.5 ADD A TABLE HEADER

Now add a header to the table so that the numbers within each category can be easily distinguished. Add two `print` statements in the `print_summary()` function. It may be easier to type the line with dashes first—one of 15 dashes followed by three of 10 dashes, with spaces in between. You can then line up the titles against the dashes.




```
def print_summary():
    print("Budget              Budgeted      Spent  Remaining")
    print("--------------- ---------- ---------- ----------")
    for name in budgets:
        budgeted = budgets[name]
        spent = expenditure[name]
```

The titles have been aligned against the dashes

3.6 ADD A TABLE FOOTER

To complete the summary table, you can add a footer to it. This will add up the contents of the various columns and display their total value. Update the `print_summary()` function as shown below. Use the same format instructions for printing the totals that you used for the budget. However, remember to use "Total" instead of the budget name and `total_budgeted`, `total_spent`, and `total_remaining` for the other variables.

```
def print_summary():
    print("Budget              Budgeted      Spent  Remaining")
    print("--------------- ---------- ---------- ----------")
    total_budgeted = 0
    total_spent = 0
    total_remaining = 0
    for name in budgets:
```

Sets the total variables to **0**

```
        budgeted = budgets[name]
        spent = expenditure[name]
        remaining = budgeted - spent
        print(f'{name:15s} {budgeted:10.2f} {spent:10.2f} '
              f'{remaining:10.2f}')
        total_budgeted += budgeted
        total_spent += spent
        total_remaining += remaining
    print("--------------- ---------- ---------- ----------")
    print(f'{"Total":15s} {total_budgeted:10.2f} {total_spent:10.2f} '
          f'{total_budgeted - total_spent:10.2f}')
```

Adds the amount to the totals

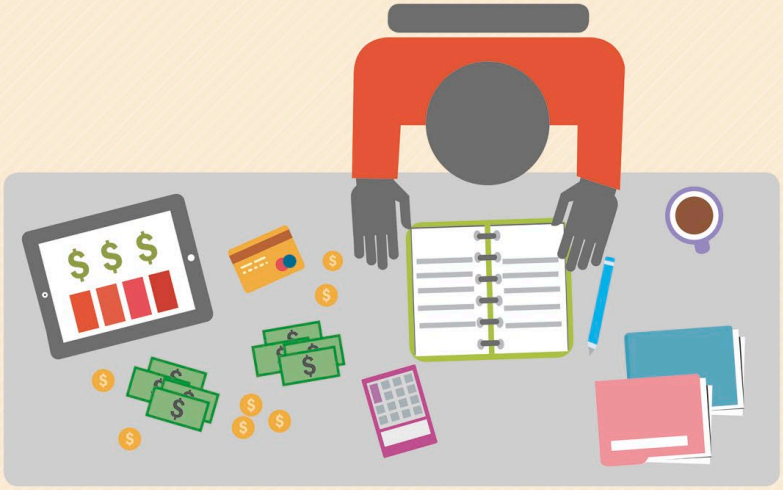
Prints another separator line and the summary with the totals below it

```
>>> add_budget("Groceries", 500)
2000.0
>>> add_budget("Rent", 900)
1100.0
>>> spend("Groceries", 35)
465
>>> spend("Groceries", 15)
450
>>> print_summary()
Budget             Budgeted      Spent  Remaining
--------------- ---------- ---------- ----------
Groceries            500.00      50.00     450.00
Rent                 900.00       0.00     900.00
--------------- ---------- ---------- ----------
Total               1400.00      50.00    1350.00
```

Final summary table printed with a header and footer

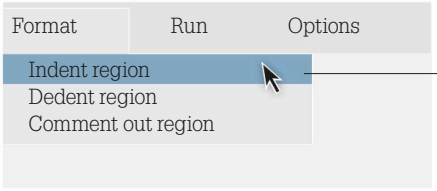
4 Converting the code into a class

In this section, you will take all the code written so far and turn it into a Python class (see pp.156–157). This will allow the user to track multiple budgets simultaneously.



4.1 INDENT THE CODE

Because Python is structured using indentation, you need to indent the entire code to convert it into a class. Select all of the code in the file and then choose "Indent region" from the Format menu. Next, add a new class header at the top of the file, before the variables.



Defines the new class

The variables will now appear indented

```
class BudgetManager:
    available = 2500
    budgets = {}
    expenditure = {}
```

4.2 ADD INITIALIZER

Indent the three variables again and add a function header to them. Functions inside a class are known as methods. The `__init__` method is called when a new instance of a class is created. This method is called the "initializer," as it sets the initial values for the instance variables. The first argument of the initializer is the new instance, called **`self`** by convention. You can also add additional arguments that will allow you to provide useful values, such as **`amount`** here.

```
class BudgetManager:
    def __init__(self, amount):
        available = 2500
        budgets = {}
        expenditure = {}
```

Arguments within the initializer

4.3 CREATE INSTANCE VARIABLES

Next, convert the three variables into instance variables. This is done by adding "**self.**" before each of the variable names. Use the argument **amount** instead of **2500** as the initial value for the available instance variable.

```
class BudgetManager:
    def __init__(self, amount):
        self.available = amount
        self.budgets = {}
        self.expenditure = {}
```

Converts the variables to instance variables

4.4 TURN THE FUNCTIONS INTO METHODS

Now you need to turn all of the other functions in the code into methods. Just like with the initializer, you can do this by adding **self** as the first argument of every function and then adding **self.** before each use of the instance variables. Modify the **add_budget()** function as shown below. Delete the **global available** line from the **add_budget** method, as **available** is now an instance variable.

Remove the line **global available** from between these two lines of code

```
    def add_budget(self, name, amount):
        if name in self.budgets:
            raise ValueError("Budget exists")
        if amount > self.available:
            raise ValueError("Insufficient funds")
        self.budgets[name] = amount
        self.available -= amount
        self.expenditure[name] = 0
        return self.available
    def spend(self, name, amount):
        if name not in self.expenditure:
            raise ValueError("No such budget")
        self.expenditure[name] += amount
        budgeted = self.budgets[name]
        spent = self.expenditure[name]
        return budgeted - spent
    def print_summary(self):
```

Adds an argument to the function

```
        print("Budget                Budgeted      Spent  Remaining")
        print("--------------- ---------- ---------- ----------")
        total_budgeted = 0
        total_spent = 0
        total_remaining = 0
        for name in self.budgets:
            budgeted = self.budgets[name]
            spent = self.expenditure[name]
```

Add **self.** before each use of the instance variable

SAVE

4.5 RUN THE MODULE

Save and run the module. Type these lines in the shell window to test the code. This will add a newly created instance of the BudgetManager class. The code inspects the instance variables by putting **outgoings.** before their name. You can call methods in a similar way, by putting the variable name before the function name with a period.

Sets the variable **outgoings** to an instance of the BudgetManager class

```
>>> outgoings = BudgetManager(2000)
>>> outgoings.available
2000
>>> outgoings.budgets
{}
>>> outgoings.expenditure
{}
>>> outgoings.add_budget("Rent", 700)
1300
>>> outgoings.add_budget("Groceries", 400)
900
```

```
>>> outgoings.add_budget("Bills", 300)
600
>>> outgoings.add_budget("Entertainment", 100)
500
>>> outgoings.budgets
{'Rent': 700, 'Groceries': 400, 'Bills': 300, 'Entertainment': 100}
>>> outgoings.spend("Groceries", 35)
365
>>> outgoings.print_summary()
Budget             Budgeted      Spent  Remaining
---------------- ---------- ---------- ----------
Rent                700.00       0.00     700.00
Groceries           400.00      35.00     365.00
Bills               300.00       0.00     300.00
Entertainment       100.00       0.00     100.00
---------------- ---------- ---------- ----------
Total              1500.00      35.00    1465.00
```

5 TRACKING MULTIPLE BUDGETS

It is possible to reset the budget by simply creating a new instance of the BudgetManager class, by typing this code in the shell window. You can even have multiple BudgetManager instances for tracking separate budgets. To test this, create a new budget called `vacation`. As the available, budgets, and expenditure variables are stored within each instance, they are distinct from each other and can have different values for the different instances.

Creates a new instance of the BudgetManager class

```
>>> outgoings = BudgetManager(2500)
>>> outgoings.add_budget("Groceries", 500)
2000
>>> outgoings.print_summary()
```

Prints the summary for the new instance

```
Budget             Budgeted      Spent  Remaining
-------------- ---------- ---------- ----------
Groceries            500.00       0.00     500.00
-------------- ---------- ---------- ----------
Total                500.00       0.00     500.00
>>> vacation = BudgetManager(1000)
>>> vacation.add_budget("Flights", 250)
750
>>> vacation.add_budget("Hotel", 300)
450
>>> vacation.spend("Flights", 240)
10
>>> vacation.print_summary()
Budget             Budgeted      Spent  Remaining
-------------- ---------- ---------- ----------
Flights              250.00     240.00      10.00
Hotel                300.00       0.00     300.00
-------------- ---------- ---------- ----------
Total                550.00     240.00     310.00
```

Adds another new instance of BudgetManager

5.1 USING THE CODE

The code written in this project is a module that can be used in other programs. This module can be imported and used like any other Python library (see pp.116–117). Try this out by creating a new module that will import this one. Open a new file and save it in the BudgetManager folder you created earlier. Name this new file "test.py". Now add this code to create an instance of the BudgetManager class that calls methods on it.

```
import budget
outgoings = budget.BudgetManager(2500)
outgoings.add_budget("Groceries", 500)
outgoings.print_summary()
```

Imports the module **budget** into this new one

The BudgetManager class is referenced by adding the **budget** module name before it with a period

Hacks and tweaks

Changing your mind

In the project, you modified the **add_budget** method to stop the same budget from being added twice. However, it would also be useful to have a way to change a budget later. You can add a new method to do this. Add the following new method below the existing **add_budget** method. You may need to look at this code carefully to follow the logic. Add a line in the test.py module to call this new method so that you can see it working.

```
    def change_budget(self, name, new_amount):
        if name not in self.budgets:
            raise ValueError("Budget does not exist")
        old_amount = self.budgets[name]
        if new_amount > old_amount + self.available:
            raise ValueError("Insufficient funds")
        self.budgets[name] = new_amount
        self.available -= new_amount - old_amount
        return self.available
```

Checks if the budget to be changed exists

Gets the old amount of the budget

Checks if the old amount added to the available amount covers the new amount

Updates the budget

Reduces **available** by the difference between the old and the new amounts

Record expenditure details

So far, the project tracks the total expenditures against each budget. However, in a more advanced program, you would want to keep track of each particular item of expenditure. You can do this by using lists of amounts spent inside the **expenditure** dictionary and then adding these together whenever you need the total.

1 **CREATE AN EXPENDITURE LIST**

Start by modifying the **expenditure** dictionary in the **add_budget** method. You need to store an empty list inside **expenditure**, instead of 0. This allows multiple values to be stored in it.

```
        self.budgets[name] = amount
        self.expenditure[name] = []
        self.available -= amount
        return self.available
```

Stores an empty list

2 ADD EXPENSES TO LIST

Now in the **spend** method, change the **expenditure** variable so that each new expense is added to the list. Because **expenditure** no longer sums up the amounts spent automatically, you will have to modify the **spent** variable to perform the calculations and get the total spent so far.

```
            raise ValueError("No such budget")
        self.expenditure[name].append(amount)
        budgeted = self.budgets[name]
        spent = sum(self.expenditure[name])
        return budgeted - spent
```

Appends the amount to the list

3 GET TOTAL EXPENDITURES

You now need to sum up the items in the **print_summary** method. Modify the **spent** variable as shown below. You will find that the code functions identically if you run the "test.py" module again, with a record for each item of expenditure.

```
        for name in self.budgets:
            budgeted = self.budgets[name]
            spent = sum(self.expenditure[name])
            remaining = budgeted - spent
```

Gets the amount spent for each budget

Pygame Zero

Pygame Zero is a tool that enables programmers to build games using Python. It provides a simplified way to create programs using the powerful functions and data types in the pygame library.

Installing Pygame Zero on Windows

The latest versions of **pygame** and **Pygame Zero** can be installed on a Windows computer by following the steps given below. An active internet connection is required for this.

START

1 Open the Command Prompt
On a Windows 10 operating system, click Start and open the Command Prompt by typing "Cmd" into the Search field. If you have an older version of Windows, find the Command Prompt in the Systems folder.

The Command Prompt thumbnail looks like this

2 Install a package manager
The easiest way to install or update Python libraries and modules on a system is to use a package manager called "pip". Type the following command in the Command Prompt and press Enter.

```
python -m pip install -U pip
```

Installing Pygame Zero on a Mac

The latest versions of **pygame** and **Pygame Zero** can be installed on a computer with macOS using the **Homebrew package manager**. Internet connectivity is essential for this.

START

1 Open the Terminal and install a package manager
Use the Terminal app to install the modules. It can be found in the "Utilities" folder under "Applications". Type the following command and then press Enter to install "Homebrew". The installation process will ask for a user login password to continue and may take some time to complete.

```
ruby -e "$(curl -fsSL https://raw.
githubusercontent.com/Homebrew/
install/master/install)"
```

2 Install Python 3
Next, use "Homebrew" to check if Python 3 is already installed on the system, and install it if not. Type the following command in the Terminal window and then press Enter.

```
brew install python3
```

UPDATES

Occasionally, programmers may experience problems while running **Pygame Zero** programs after updating to a new version of their operating system. To fix this, the tools added during **Pygame Zero** installation can be uninstalled, then reinstalled using the instructions here.

FINISH

3 Install Pygame
Once the "pip" package manager is installed, type the command shown below and then press Enter. This will use "pip" to install the **pygame** library.

```
pip install pygame
```

4 Install Pygame Zero
Finally, type the following command and then press Enter. This will install **Pygame Zero**.

```
pip install pgzero
```

3 Install extra tools
To install some extra tools that the system requires to run **Pygame Zero**, use "Homebrew" and type this command into the Terminal window and then press Enter.

```
brew install sdl sdl_mixer sdl_sound
sdl_ttf
```

4 Install pygame
Now type this command to install the **pygame** library and press Enter.

```
pip3 install pygame
```

FINISH

5 Install Pygame Zero
Finally, this last command will install **Pygame Zero**.

```
pip3 install pgzero
```

Knight's quest

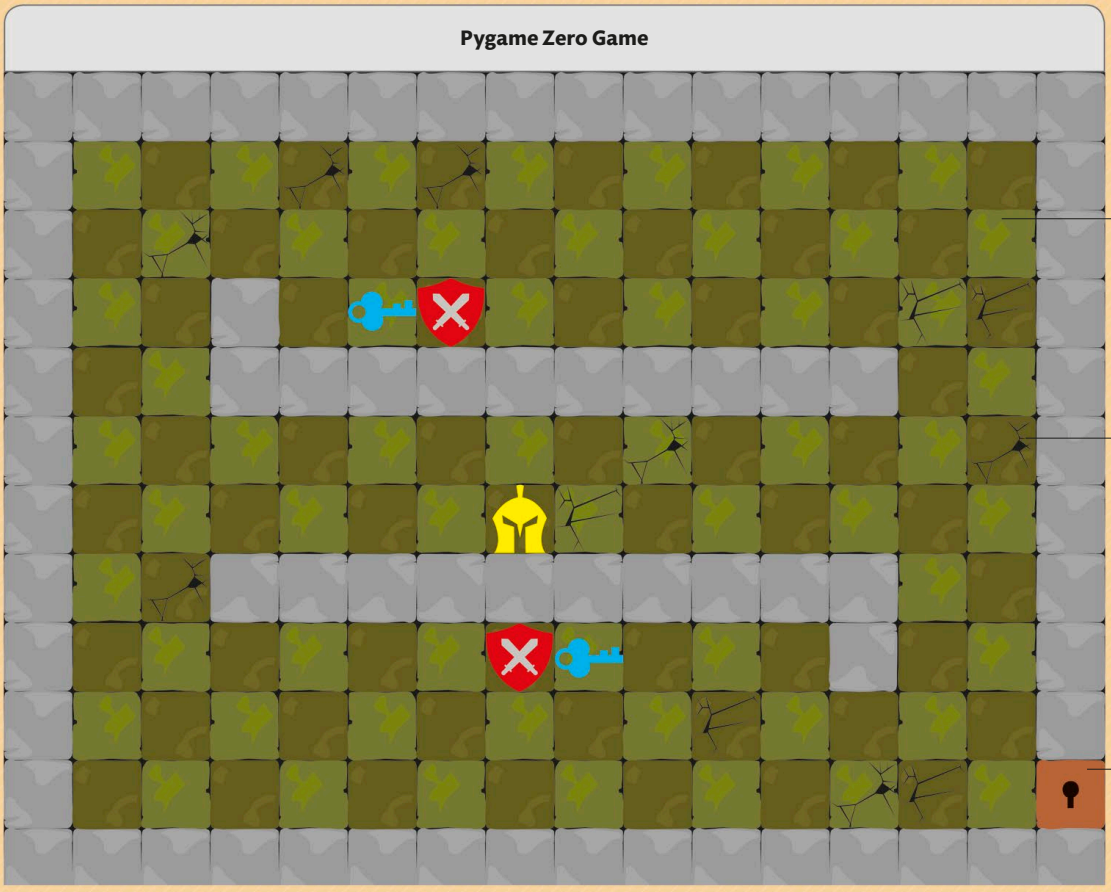
This fast-paced, two-dimensional game will put your reflexes to the test. It uses coordinates to create a two-dimensional playing area and Pygame Zero's Actor class to introduce the characters and collectable items in the game. An event loop program makes the game run smoothly.

How to play this game

The aim of this game is to navigate the knight around the dungeon—a two-dimensional playing area—with the arrow keys, but you cannot move through walls or the locked door. Collect the keys by moving over them. However, you need to avoid the guards as they try to move toward the knight. Any contact with the guards ends the game. You win if you can get to the door after picking up all of the keys.

Dungeon crawl

This project is an example of a style of game called dungeon crawl. In such games, the player usually navigates a labyrinthine environment, collecting items and battling or avoiding enemies. This game will use the classic top-down 2D view, where the player appears to be looking down at the play area from above.



YOU WILL LEARN

- How to use lists
- How to index strings
- How to use nested loops
- How to use **Pygame Zero** to make a simple game

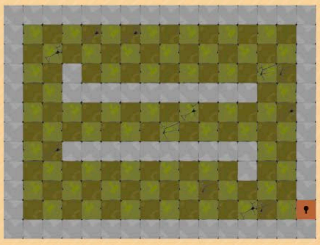
Time: 2 hours

Lines of code: 151

Difficulty level

WHERE THIS IS USED

The concepts in this project are applicable to all kinds of 2D computer games, especially ones that are played on cell phones. Apart from in dungeon-crawl games, image tile grids are also used in color- and shape-matching games. The logic applied in this game could also be adapted to simple robotics projects.



The scenery
The game is based on a simple grid on which square images called "tiles" are placed. The scenery of the game consists of a background of floor tiles, with additional tiles representing the walls and the door.

GUARD

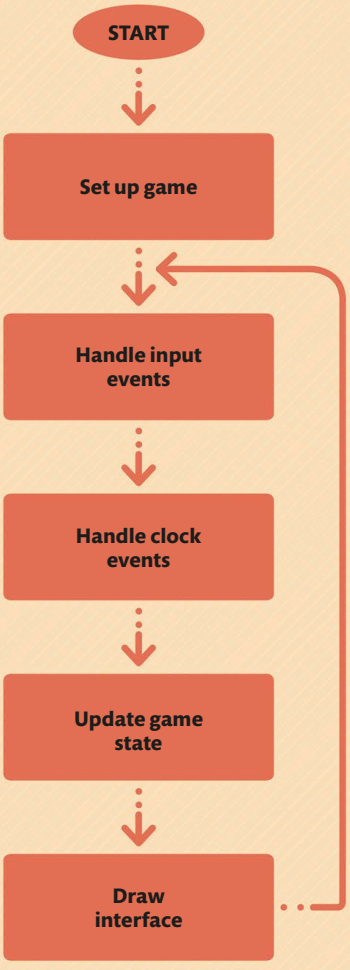
KNIGHT

KEY

The actors
The movable or collectable items in the game are called actors. In this game, the actors are the same size as the tiles so that each is contained within one grid square. They are drawn on top of the scenery so that the background can be seen behind and through them.

The Pygame Zero game loop

A **Pygame Zero** program is an example of an event loop program. An event loop runs continuously, calling other parts of the program when an event occurs so that actions can be taken. The code necessary to manage this loop is part of **Pygame Zero**, so you only need to write the handler functions that deal with these events.



Set up game
Top-level statements in the Python file will be executed first and can be used to initialize the game state and configure **Pygame Zero** itself. **Pygame Zero** will then open a window and repeat the event loop continuously.

Handle input events
Pygame Zero will check for input events, such as key presses, mouse movements, and button presses each time through the loop. It will call the appropriate handler function (see p.185) when one of these events occurs.

Handle clock events
The **Pygame Zero** clock allows users to schedule calls to handler functions in the future. These delayed function calls will be made during this part of the event loop.

Update game state
At this point, **Pygame Zero** allows the user to do any work that they want done on every loop iteration by calling the update handler function. This is an optional function.

Draw interface
Finally, **Pygame Zero** calls the draw handler function, which will redraw the contents of the game window to reflect the current game state.

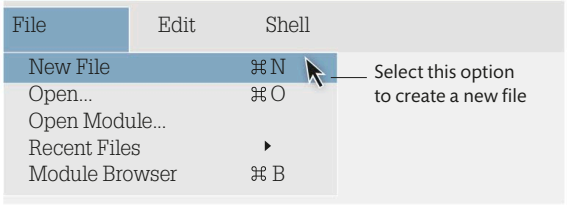
1 Setting up

To get started with this project, you will first need to create the folders to hold all the files. The next step will be to write some code to draw the background and the players onscreen.

1.1 CREATE THE GAME FILE

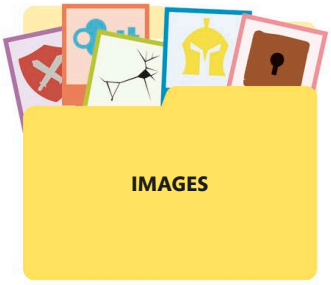
First, create a new folder on your desktop and name it "KnightsQuest". Next, open IDLE and create a new file by choosing the New File option from the File menu. Save this file in the KnightsQuest folder by choosing Save As... from the same menu. Name the file "quest.py".

IDLE



1.2 SET UP THE IMAGES FOLDER

You now need a folder to hold the images required for this project. Go to the KnightsQuest folder and create a new folder inside it called "images". Go to *www.dk.com/coding-course* and download the resource pack for this book. Copy the image files for this project into the new "images" folder.



1.3 INITIALIZE PYGAME ZERO

Go to the "quest.py" file you created earlier and type these lines of code into it to define the dimensions of the game grid. This will create a working **Pygame Zero** program. Save the file, then choose Run Module from the Run menu (or press the F5 key on your keyboard) to execute the code. You will only see a black window at this point. Close this window and continue.

Imports the **Pygame Zero** functionality

```
import pgzrun

GRID_WIDTH = 16
GRID_HEIGHT = 12
GRID_SIZE = 50

WIDTH = GRID_WIDTH * GRID_SIZE
HEIGHT = GRID_HEIGHT * GRID_SIZE

pgzrun.go()
```

These define the width and height of the game grid and the size of each tile

`WIDTH` and `HEIGHT` are special **Pygame Zero** variable names

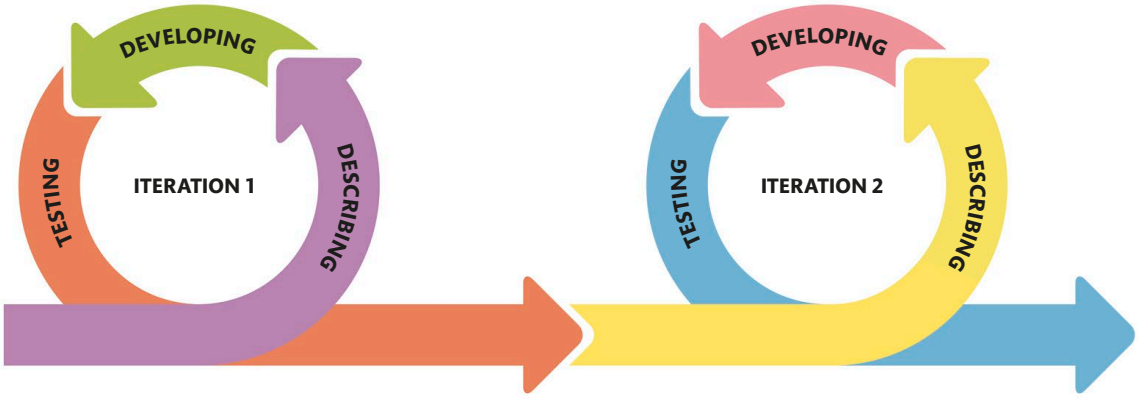
These define the size of the game window

Starts **Pygame Zero**

SAVE

AGILE SOFTWARE DEVELOPMENT

Each step in this project will describe a new piece of functionality that makes the program more useful or interesting, and will then show you the code you need to add to implement this. At the end of each step, you will have a working program that you can run and try out. This process of evolving a program by iteratively describing, developing, and testing small pieces of new functionality is part of a style of programming called Agile Software Development.



1.4 DRAW THE BACKGROUND

In this step, you will draw the floor of the dungeon as a grid of floor tiles filling the game window. Add the following lines of code to your program.

```
HEIGHT = GRID_HEIGHT * GRID_SIZE
def screen_coords(x, y):
    return (x * GRID_SIZE, y * GRID_SIZE)
def draw_background():
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            screen.blit("floor1", screen_coords(x, y))
def draw():
    draw_background()
pgzrun.go()
```

This function converts a grid position to screen coordinates

Loops over each grid row

Loops over each grid column

The draw handler function is called automatically from the game loop (see p.179)

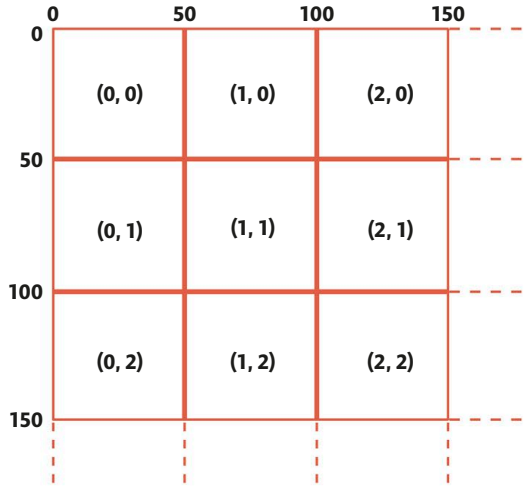
Draws the dungeon floor as a background onscreen

`screen.blit()` draws the named image at the given screen position

GRID AND SCREEN COORDINATES

The playing area in this project is a grid that is 16 squares wide and 12 squares high. Each of these squares is 50 x 50 pixels. The position of a square is denoted by **x** and **y** coordinates, written as a pair in brackets (x, y). The x coordinate refers to a column number and the y coordinate refers to a row number. In programming, counting starts at the number 0, so the top left grid position for this project is (0, 0) and the bottom right grid position is (15, 11). In Python, `range(n)` iterates over the numbers 0 to n-1, so `range(GRID_HEIGHT)` is 0...11 and `range(GRID_WIDTH)` is 0...15.

Nesting one loop inside another allows the program to iterate across every grid position. Multiplying the grid coordinates by the size of the grid squares gives the coordinate of the top left corner of that grid square relative to the top left corner of the game window. **Pygame Zero** refers to this as the screen.



1.5 DEFINE THE SCENERY

You can now draw the walls of the dungeon, add a door, and define the map of the game. Add the following code below the constants in your IDLE file to do this. The map is defined as a list of 12 strings, each representing a row of the grid. Each string is 16 characters wide, with each character describing a single grid square.

```
HEIGHT = GRID_HEIGHT * GRID_SIZE

MAP = ["WWWWWWWWWWWWWWWW",
       "W              W",
       "W              W",
       "W  W  KG       W",
       "W  WWWWWWWWW   W",
       "W              W",
       "W       P      W",
       "W  WWWWWWWWW   W",
       "W       GK  W  W",
       "W              W",
       "W              D",
       "WWWWWWWWWWWWWWWW"]
```

W represents a wall tile

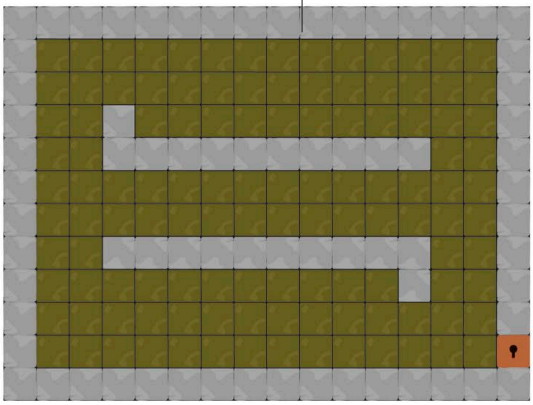
Spaces represent empty squares

K is a key, and G is a guard

P is the player

D is the door

The dungeon has 12 rows and 16 columns of wall tiles



OUTPUT ON THE SCREEN

1.6 ADD A FUNCTION TO DRAW THE SCENERY

Next, add a new `draw_scenery()` function above the `draw()` function. This will draw the background of each square on the map. Because the map is a list of strings, subscripting it as `MAP[y]` selects the string representing the row of the grid specified by y (counting from 0). Subscripting this string with `[x]` selects the character representing the square in the column specified by x (also counting from 0). The second subscript is written immediately after the first as `MAP[y][x]`.

```
            screen.blit("floor1", screen_coords(x, y)
def draw_scenery():
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            square = MAP[y][x]
            if square == "W":
                screen.blit("wall", screen_coords(x, y))
            elif square == "D":
                screen.blit("door", screen_coords(x, y))
def draw():
    draw_background()
    draw_scenery()
```

Loops over each grid position

Extracts the character from the map represented by this grid position

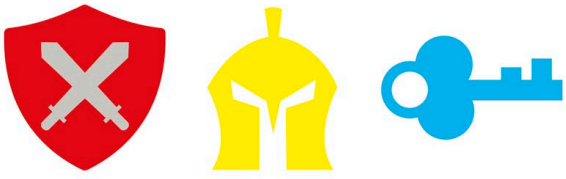
Draws a wall tile at the screen position represented by `W`

Draws a door tile at position `D`

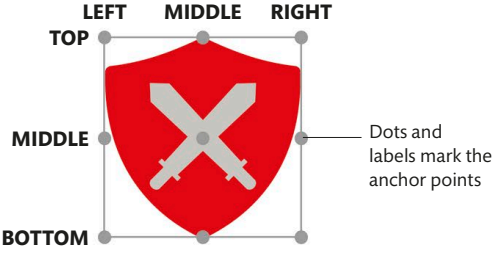
Draws the scenery after (on top of) the background has been drawn

THE ACTOR CLASS

Pygame Zero provides a class called Actor to represent the actors, or the movable items, in games. You can create an Actor object with the name of the image that should be used when drawing it and then keyword arguments that specify other properties of the object, if required. The most important property is **pos**, which specifies the screen coordinates that the image should be drawn at. The **anchor** property specifies what point on the image the **pos** coordinates refer to. It is a pair of strings where the first gives the x anchor point—"left", "middle", or "right"—and the second gives the y anchor point—"top", "middle", or "bottom". You will anchor the actors' **pos** to the top left of the image, as this matches the coordinates returned by the `screen_coords()` function.



THE ACTORS



Dots and labels mark the anchor points

1.7 INITIALIZE THE PLAYER

Create an actor for the player and set its starting position on the map. Add a new setup function below the `screen_coords()` function to do this.

```
def screen_coords(x, y):
    return (x * GRID_SIZE, y * GRID_SIZE)
def setup_game():
    global player
    player = Actor("player", anchor=("left", "top"))
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            square = MAP[y][x]
            if square == "P":
                player.pos = screen_coords(x, y)
```

Defines `player` as a global variable

Creates a new `Actor` object and sets its anchor position

Loops over each grid position

Extracts the character from the map representing this grid position

Checks if this grid position is the player

Sets the position of `player` to the screen coordinates of this grid position

1.8 DRAW THE PLAYER

After initializing the player, you need to draw it onscreen. Add a `draw_actors()` function above the `draw()` function in the code. Next, add a call to it at the end of the `draw()` function. Finally, call the `setup_game()` function just before **Pygame Zero** runs.

```
            screen.blit("door", screen_coords(x, y))
def draw_actors():
    player.draw()
def draw():
    draw_background()
    draw_scenery()
    draw_actors()
setup_game()
pgzrun.go()
```

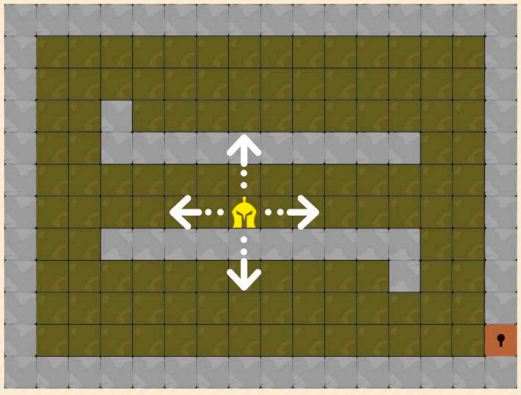
Draws the player actor onscreen at its current position

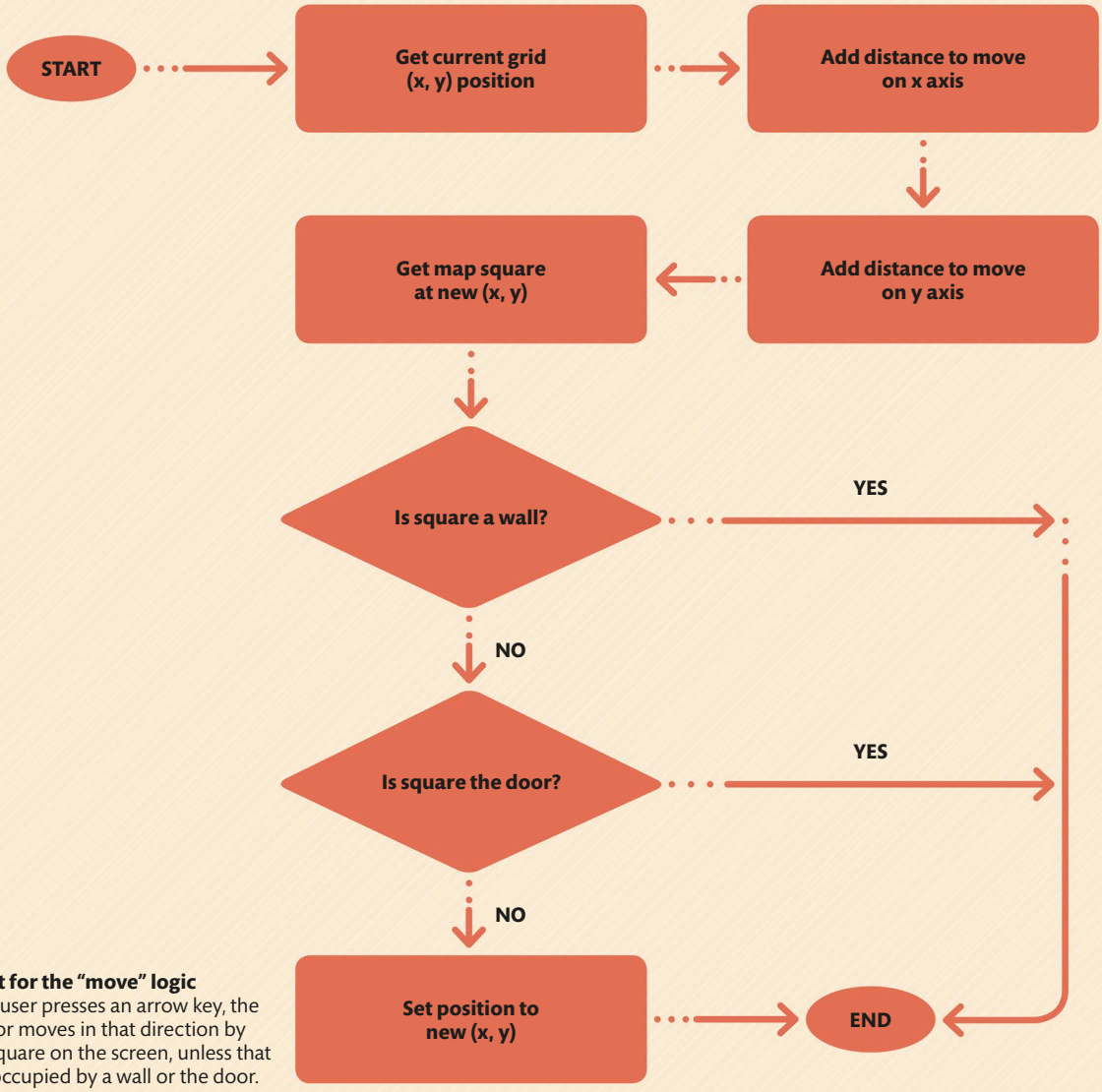
Draws the actors after (on top of) the background and scenery have been drawn

SAVE

2 Moving the player

Now that you have created the player, it is time to write code to move it on the screen. You will use an event-handler function, which reacts to key presses, to do this.





Flowchart for the "move" logic
When the user presses an arrow key, the player actor moves in that direction by one grid square on the screen, unless that square is occupied by a wall or the door.

2.1 ADD A UTILITY FUNCTION

First, you need to define a function to determine which grid square the actor is in. You can do this by dividing the actor's x and y coordinates by the size of a grid square and then using the built-in `round()` function to make sure that the result is the nearest whole number. Add this function below the existing `screen_coords()` function.

```
    return (x * GRID_SIZE, y * GRID_SIZE)

def grid_coords(actor):

    return (round(actor.x / GRID_SIZE), round(actor.y / GRID_SIZE))
```

Determines the position of an actor on the grid

2.2 ADD KEY HANDLER

Now add an event handler function that will react when the user presses an arrow key. This function ensures the player moves in the right direction when any of the four arrow keys are pressed. Add this new function below the `draw()` function.

```
    draw_actors()

def on_key_down(key):

    if key == keys.LEFT:

        move_player(-1, 0)

    elif key == keys.UP:

        move_player(0, -1)

    elif key == keys.RIGHT:

        move_player(1, 0)

    elif key == keys.DOWN:

        move_player(0, 1)
```

Last line of the `draw()` function

Reacts when the user presses down on a key

Player moves left by one grid square

Player moves up by one grid square

Player moves right by one grid square

Player moves down by one grid square

2.3 MOVE THE ACTOR

Next, define the `move_player()` function. This function takes the distance in grid squares that a player moves on the x and y axes, respectively. Add this function immediately after the `on_key_down()` function.

```
def move_player(dx, dy):

    (x, y) = grid_coords(player)

    x += dx

    y += dy

    square = MAP[y][x]

    if square == "W":

        return

    elif square == "D":

        return

    player.pos = screen_coords(x, y)
```

Gets the current grid position of `player`

Adds the x axis distance to `x`

Adds the y axis distance to `y`

Gives the tile at this position

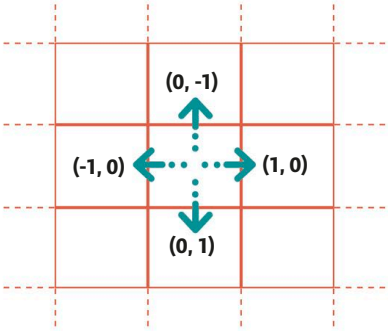
Stops the execution of the `move_player()` function, if the player touches the wall

Returns immediately if it is a door

Updates position of `player` to the new coordinates

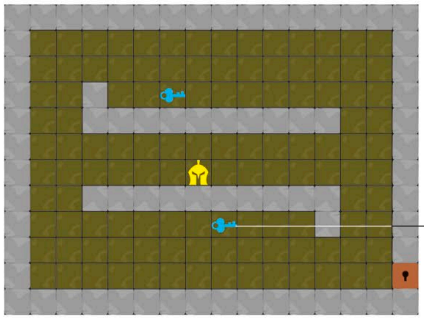
MOVING ON THE GRID

As the grid and screen coordinates start at the top left corner, moving left on the grid represents a negative change on the x axis and moving right represents a positive change. Similarly, moving up is a negative change on the y axis and moving down is a positive change.



3 ADD THE KEYS

You now need to add more actors to the game. Add some keys for the player to collect. For each key marked on the map, create an actor with the key image and set its position to the screen coordinates of that grid position. Add this code to the **`setup_game()`** function to create the key actors.



Keys will appear at the coordinates set in the code

Defines **`keys_to_collect`** as a global variable

```
def setup_game():
    global player, keys_to_collect
    player = Actor("player", anchor=("left", "top"))
    keys_to_collect = []
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            square = MAP[y][x]
            if square == "P":
                player.pos = screen_coords(x, y)
            elif square == "K":
                key = Actor("key", anchor=("left", "top"), \
                            pos=screen_coords(x, y))
                keys_to_collect.append(key)
```

Sets **`keys_to_collect`** to an empty list initially

Creates a key if the square is **K**

Creates the **key** actor with an image, anchor, and position

Adds this actor to the list of keys created above

SPECIAL NAMES

It would have seemed natural to name the global variable with the list of key actors **keys**. However, you need to be careful when choosing names for your variables to avoid confusion with either built-in function names or names that are special to **Pygame Zero**. You may remember from the last step that **keys** is a special object with items representing all of the keys on the keyboard.

3.1 DRAW NEW KEY ACTORS

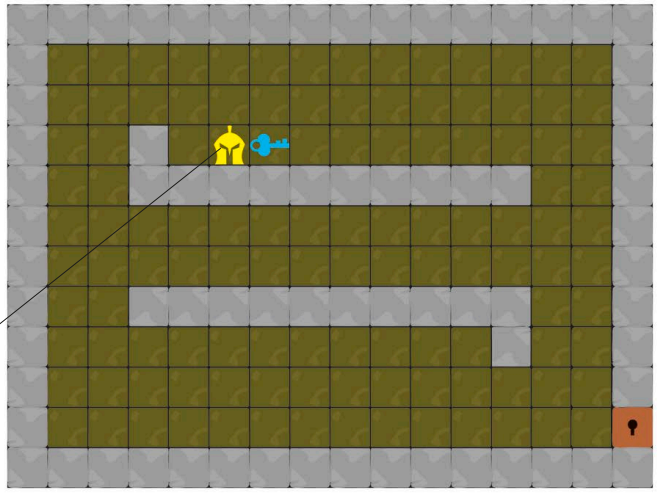
Make the game more interesting by adding multiple keys for the player to collect. Draw the new key actors by adding the following lines to the `draw_actors()` function.

```
def draw_actors():
    player.draw()
    for key in keys_to_collect:
        key.draw()
```

Draws all the actors in the list `keys_to_collect`

3.2 PICK UP THE KEYS

When the player actor moves into a grid square containing a key, the program will remove that key from the list of keys to be collected and stop drawing it onscreen. When there are no more keys to be collected, the player actor will be allowed to move into the grid square containing the door. Make the following changes to the `move_player()` function to do this. Save the code and try running the program to check if you can move around and pick up the keys. You should be able to go into the door square once you pick up all of the keys, but see what happens if you try moving farther—we will fix this problem in the next few steps.



Move the player over the key to pick it up

Checks if the `keys_to_collect` list is not empty

```
    elif square == "D":
        if len(keys_to_collect) > 0:
            return
        for key in keys_to_collect:
            (key_x, key_y) = grid_coords(key)
            if x == key_x and y == key_y:
                keys_to_collect.remove(key)
                break
    player.pos = screen_coords(x, y)
```

Returns immediately if the list is not empty

Loops over each of the key actors in the list

Gets the grid position of a key actor

Checks if the new player position matches the key position

Removes this key from the list if player position matches key position

Breaks out of the `for` loop, as each square can only contain one key

SAVE

3.3 GAME OVER!

If the player actor moves into the grid square that contains the door (after having picked up all of the keys), then the game should come to an end and the player should no longer be allowed to move. To do this, update the `setup_game()` function to define a new global variable that checks whether the game is over or not.

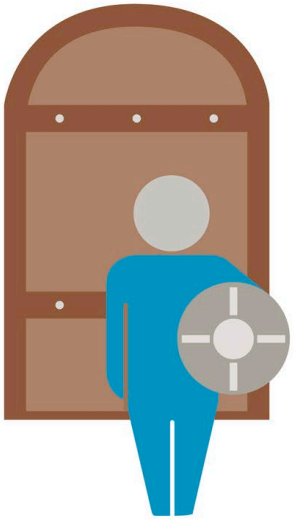
```
def setup_game():
    global game_over, player, keys_to_collect
    game_over = False
    player = Actor("player", anchor=("left", "top"))
    keys_to_collect = []
    for y in range(GRID_HEIGHT):
```

Defines `game_over` as a global variable

Sets the variable to `False` initially

3.4 TRIGGER GAME OVER

Now set the game to be over when the player gets to the door. Make the following changes to the `move_player()` function. Run the program to try it out. You should not be able to move when you get to the door, so the program will not crash.



```
def move_player(dx, dy):
    global game_over
    if game_over:
        return
    (x, y) = grid_coords(player)
    x += dx
    y += dy
    square = MAP[y][x]
    if square == "W":
        return
    elif square == "D":
        if len(keys_to_collect) > 0:
            return
        else:
            game_over = True
    for key in keys_to_collect:
```

Checks if `game_over` is set

Returns immediately without moving

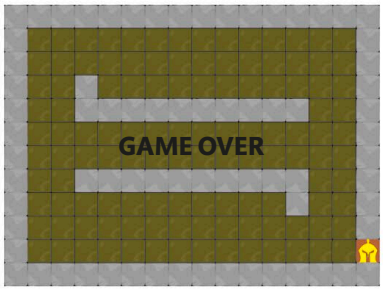
Checks if all of the keys have been picked

Sets `game_over` to `True` and continues the move

SAVE

3.5 GAME OVER MESSAGE

When the player gets to the door, the program stops, but it is not clear to the user that the game is over. You need to add a `GAME OVER` message to your code that is displayed onscreen when the game ends. Define a new function, `draw_game_over()`, to draw a `GAME OVER` overlay on the screen. Add the code above the `draw()` function.



Sets the position of the `GAME OVER` message onscreen

Anchors the text by its bottom edge

Draws the text at this location

```
def draw_game_over():
    screen_middle = (WIDTH / 2, HEIGHT / 2)
    screen.draw.text("GAME OVER", midbottom=screen_middle, \
                     fontsize=GRID_SIZE, color="cyan", owidth=1)
def draw():
    draw_background()
    draw_scenery()
    draw_actors()
    if game_over:
        draw_game_over()
```

Draws the text at this location

DRAWING TEXT WITH PYGAME ZERO

The `screen.draw.text()` function allows you to draw a piece of text onscreen. This function takes a string with the text to be drawn and then some optional keyword arguments, as shown here. See the **Pygame Zero** online documentation for other keywords.

KEYWORD ARGUMENTS	
Property name	**Description**
fontsize	The font size in pixels.
color	A string giving a color name or an HTML-style "#FF00FF" color, or an (r, g, b) "tuple" such as (255, 0, 255).
owidth	A number giving a relative width of an outline to draw around each character. Defaults to 0 if not specified; 1 represents a reasonable outline width.
ocolor	The color for the outline (in the same format as **color**). Defaults to "black" if not specified.
topleft, **bottomleft**, **topright**, **bottomright**, **midtop**, **midleft**, **midbottom**, **midright**, **center**	Use one of these with a pair of numbers to give the x and y screen coordinates relative to an anchor point.

3.6 CREATE THE GUARD ACTORS

The game is pretty easy to win so far. To make it more difficult, add some guards as obstacles. For each guard on the map, create an actor with a guard image and set its position to the screen coordinates of that grid position. Update the `setup_game()` function to do this.

Defines **guards** as a global variable

```
def setup_game():
    global game_over, player, keys_to_collect, guards
    game_over = False
    player = Actor("player", anchor=("left", "top"))
    keys_to_collect = []
    guards = []
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            square = MAP[y][x]
            if square == "P":
                player.pos = screen_coords(x, y)
            elif square == "K":
                key = Actor("key", anchor=("left", "top"), \
                            pos=screen_coords(x, y))
                keys_to_collect.append(key)
            elif square == "G":
                guard = Actor("guard", anchor=("left", "top"), \
                              pos=screen_coords(x, y))
                guards.append(guard)
```

Sets **guards** to an empty list initially

Creates the **guard** actor

Creates a guard if the square is **G**

Adds this actor to the list of guards created above

3.7 DRAW THE GUARDS

To add another guard to the game, add this code to the `draw_actors()` function. Save the code and then run the program to check if the guards appear onscreen.

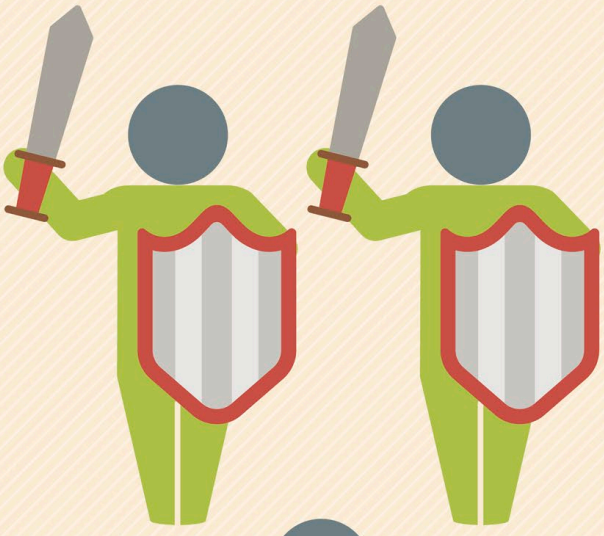
```
    key.draw()
for guard in guards:
    guard.draw()
```

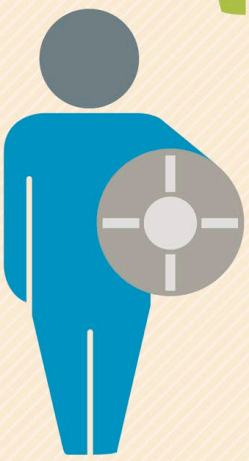
Draws all the actors in the list **guards**

SAVE

4 Moving the guards

Once the guards are in place, they will attempt to move one grid square closer to the player along either the x or y axis every half-second, unless they move into a square occupied by a wall. If the move takes the guard into the same grid position as the player, then the game is over. Like the player, the guards should not move if the game is over. Add the code in this section to make the guards move.





4.1 ADD A FUNCTION TO MOVE A GUARD

Start by adding a new function, **move_guard()**, to move a single guard. The code will work for any guard actor, so pass an argument for the guard you want to move. Add the following code immediately after the **move_player()** function.

```
            break
    player.pos = screen_coords(x, y)
def move_guard(guard):
    global game_over
    if game_over:
        return
    (player_x, player_y) = grid_coords(player)
    (guard_x, guard_y) = grid_coords(guard)
    if player_x > guard_x and MAP[guard_y][guard_x + 1] != "W":
        guard_x += 1
    elif player_x < guard_x and MAP[guard_y][guard_x - 1] != "W":
        guard_x -= 1
    elif player_y > guard_y and MAP[guard_y + 1][guard_x] != "W":
        guard_y += 1
    elif player_y < guard_y and MAP[guard_y - 1][guard_x] != "W":
        guard_y -= 1
    guard.pos = screen_coords(guard_x, guard_y)
    if guard_x == player_x and guard_y == player_y:
        game_over = True
```

Gets the grid position of the player actor

Checks if the player is to the right of the guard and whether the square to the right is a wall

Defines **game_over** as a global variable

Returns immediately, without moving, if the game is over

Gets the grid position of this guard actor

Increases the guard's x grid position by **1** if the above condition is true

Checks if the player is to the left of the guard

Updates the guard actor's position to the screen coordinates of the (possibly updated) grid position

Ends the game if the guard's grid position is the same as the player's grid position

4.2 MOVE ALL OF THE GUARDS

Next, add a function to move each of the guards in turn. Add this code just below the lines you typed in the previous step.

```
def move_guards():
    for guard in guards:
        move_guard(guard)
```

Loops through each guard actor in **guards** list

Moves all the guard actors in the list

4.3 CALL THE FUNCTION

Finally, add this code to call the `move_guards()` function every half-second. You need to add a new constant at the top of the file to specify this interval.

```
GRID_SIZE = 50
GUARD_MOVE_INTERVAL = 0.5
```

Sets the time interval for a guard to move onscreen

4.4 SCHEDULE THE CALL

To ensure that the guards move smoothly after every half-second, you need to add a timer that calls the `move_guards()` function repeatedly during the course of the program. Add the following code at the bottom of the file. This calls the `move_guards()` function after every `GUARD_MOVE_INTERVAL` seconds. Run the program and check if the guards chase the player. You should be able to see the `GAME OVER` message if a guard catches the player. Try changing the value of `GUARD_MOVE_INTERVAL` to make the game easier or harder.

```
setup_game()
clock.schedule_interval(move_guards, GUARD_MOVE_INTERVAL)
pgzrun.go()
```

Schedules regular calls to the `move_guards()` function

SAVE

THE CLOCK OBJECT

The clock object has methods for scheduling function calls in a program. Some of them are given here. When calling a function, make sure you use the name of the function without the brackets. This is because you can only schedule calls to functions that take no arguments, as there is no way to specify what arguments would be used when the call is made in the future.

METHODS FOR SCHEDULING FUNCTION CALLS

Method	Description
clock.schedule(function, delay)	Call the **function** in **delay** seconds—multiple calls to this will schedule multiple future calls to the **function**, even if the previous ones have not yet happened.
clock.schedule_unique(function, delay)	Similar to **clock.schedule()**, except that multiple calls to this will cancel any previously scheduled calls that have not yet happened.
clock.schedule_interval(function, interval)	Call the **function** every **interval** seconds.
clock.unschedule(function)	Cancel any previously scheduled calls to the **function**.

5 TRACK THE RESULT

When the game finishes and the **GAME OVER** message is displayed, you can show an additional message to indicate whether the player unlocked the door and won or was caught by a guard and lost. Create a new global variable to track whether the player won or lost. Add this code to the **`setup_game()`** function.

```
def setup_game():
    global game_over, player_won, player, keys_to_collect, guards
    game_over = False
    player_won = False
    player = Actor("player", anchor=("left", "top"))
```

Defines **`player_won`** as a global variable

Sets the variable to **`False`** initially

5.1 SET A VARIABLE

Now set the global variable when the game finishes because the player reached the door with all of the keys. Add this code to the **`move_player()`** function to do this.

```
def move_player(dx, dy):
    global game_over, player_won
    if game_over:
        return
    (x, y) = grid_coords(player)
    x += dx
    y += dy
    square = MAP[y][x]
    if square == "W":
        return
    elif square == "D":
        if len(keys_to_collect) > 0:
            return
        else:
            game_over = True
            player_won = True
    for key in keys_to_collect:
```

Sets it to **`True`** when the player wins the game

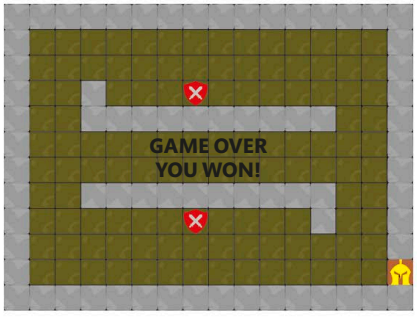
5.2 ADD THE MESSAGES

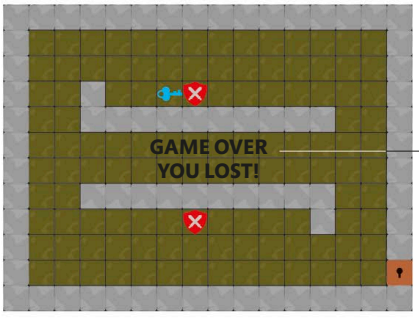
Collecting all of the keys and reaching the door is the only way that the player can win the game, so it is safe to assume that if the game finishes any other way, the player has lost. You need to display an appropriate message in each case. Add the following code to the **`draw_game_over()`** function. You will be using the **`midtop`** property to set the location of the new message. This anchors the top edge of the message to the center of the screen. As the **`GAME OVER`** message is anchored by its bottom edge, this new message will appear centered below it. Try running the game and deliberately losing. Now close the window and run it again, but try to win this time. It should not be too hard.

```
    screen.draw.text("GAME OVER", midbottom=screen_middle, \
                     fontsize=GRID_SIZE, color="cyan", owidth=1)
    if player_won:
        screen.draw.text("You won!", midtop=screen_middle, \
                         fontsize=GRID_SIZE, color="green", owidth=1)
    else:
        screen.draw.text("You lost!", midtop=screen_middle, \
                         fontsize=GRID_SIZE, color="red", owidth=1)
```

Draws the message onscreen

Use different colors to emphasize the different outcomes





The new message appears here

SAVE

5.3 REPLAY THE GAME

At the moment, the only way to have another try at the game is to close the window and run the program again. Add the following code after the **`draw()`** function to allow the user to press the spacebar when the game ends to play again. To reset the game to the beginning, you just need to call the **`setup_game()`** function again. It contains all of the code that is necessary to initialize the game and will recreate all of the actors in their starting positions. It will also reset the variables that track game progress.

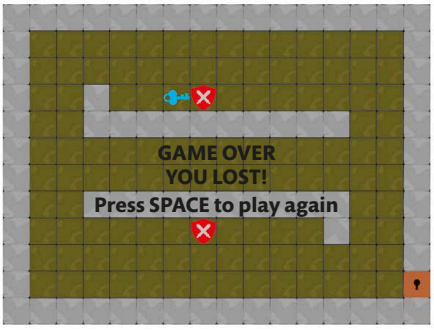
```
    if game_over:
        draw_game_over()
def on_key_up(key):
    if key == keys.SPACE and game_over:
        setup_game()
```

Checks if the spacebar has been pressed once the game is over

Calls **`setup_game()`** to reset the game

5.4 ADD ANOTHER MESSAGE

It will be useful to tell the player that they can press the spacebar to restart. To do this, add a new message at the end of the `draw_game_over()` function. You need to use `midtop` anchoring again to position the text `GRID_SIZE` pixels below the center of the screen. Run the game. You should be able to replay it now.




```
    else:
        screen.draw.text("You lost!", midtop=screen_middle, \
                         fontsize=GRID_SIZE, color="red", owidth=1)
    screen.draw.text("Press SPACE to play again", midtop=(WIDTH / 2, \
                     HEIGHT / 2 + GRID_SIZE), fontsize=GRID_SIZE / 2, \
                     color="cyan", owidth=1)
```

Draws the new message onscreen

6 Animating the actors

The game feels a little odd at the moment as the actors jump from one square to another in the grid. It would be much better if they looked more like they were moving on the screen. You can make that happen by using **Pygame Zero**'s `animate()` function.

6.1 ANIMATE THE GUARDS

Start by animating the guards. The `animate()` function creates animations that run automatically on each iteration of the game loop to update the properties of actors. Make the following change to the `move_guard()` function to animate the guards. The parameters of the `animate()` function will include the actor to animate (`guard`), the property that you want to change (`pos`), and how long the animation will run (`duration`). If you save and run the code now, you will see the guards moving smoothly across the screen toward the player.

```
    elif player_y < guard_y and MAP[guard_y - 1][guard_x] != "W":
        guard_y -= 1
    animate(guard, pos=screen_coords(guard_x, guard_y), \
            duration=GUARD_MOVE_INTERVAL)
    if guard_x == player_x and guard_y == player_y:
        game_over = True
```

Moves the actor smoothly instead of changing its position suddenly

ANIMATIONS

The `animate()` function can take two other optional keyword arguments—`tween`, which specifies how to animate the in-between values of the property, and `on_finished`, which allows you to specify the name of a function you want to call after the animation is complete. The value of `tween` should be one of the strings mentioned below.

VALUE OF THE TWEEN KEYWORD ARGUMENT

Value	Description
`"linear"`	Animate evenly from the current property value to the new; this is the default.
`"accelerate"`	Start slowly and speed up.
`"decelerate"`	Start quickly and slow down.
`"accel_decel"`	Speed up and then slow down again.
`"end_elastic"`	Wobble (as if attached to an elastic band) at the end.
`"start_elastic"`	Wobble at the start.
`"both_elastic"`	Wobble at the start and the end.
`"bounce_end"`	Bounce (as a ball would) at the end.
`"bounce_start"`	Bounce at the start.
`"bounce_start_end"`	Bounce at the start and the end.

6.2 ANIMATE THE PLAYER

Now it is time to animate the player actor. Unlike the guards, the player does not have a particular rate at which it moves, so you need to define how quickly a move should be completed. Add a new constant at the top of your file to do this. Choose `0.1` seconds as the duration the user will take to tap the movement keys to quickly evade the guards. Update the `move_player()` function as shown below. Try the game again and check if the player actor slides quickly from square to square.

```
GUARD_MOVE_INTERVAL = 0.5
PLAYER_MOVE_INTERVAL = 0.1
```

Time it takes for the player actor to move from one position to another

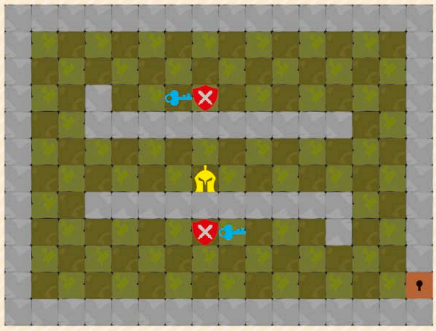
```
            keys_to_collect.remove(key)
            break
    animate(player, pos=screen_coords(x, y), \
            duration=PLAYER_MOVE_INTERVAL)
```

Updates the player's position after `0.1` seconds

SAVE

7 Make a checkerboard background

Now return to some of the earlier graphical elements and see if you can make the game look a little more interesting. At the moment, the background is just a single tile repeated across the entire floor. Add a checkerboard pattern to break things up a little and make the floor look more "tiled."



7.1 UPDATE THE BACKGROUND FUNCTION

For a checkerboard pattern, on the first row, all of the odd squares should be one color and the even squares another; on the second row, the colors need to be swapped. The following rows should repeat this pattern. You can do this by using the first color on either odd columns of odd rows or even columns of even rows, then the second color for the other squares. You can determine if a number is odd or even by using Python's **modulo** operator (see box below). Make the following changes to the `draw_background()` function to select a different floor tile image for alternate squares.

```
def draw_background():
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            if x % 2 == y % 2:
                screen.blit("floor1", screen_coords(x, y))
            else:
                screen.blit("floor2", screen_coords(x, y))
```

Checks if the **x** and **y** values are either both odd or both even

Draws the **floor1** tile at this position if the above condition is true

Draws the **floor2** tile if either of the **x** and **y** values are odd and even

THE MODULO (REMAINDER) OPERATOR

An odd or even number can be determined by dividing the number by two and then looking to see if there is a remainder or not. Python has an arithmetic operator, called the **modulo** operator, that returns the remainder from a division. It is written as **a % b**, which gives the remainder of dividing a by b. Take a look at the remainders after dividing the x and y coordinates by two. If the remainders are the same, then either the row and column are both odd or they are both even. Shown here are some examples of how the modulo operator works.

N	N % 2	N % 3	N % 4	N % 5
0	0	0	0	0
1	1	1	1	1
2	0	2	2	2
3	1	0	3	3
4	0	1	0	4
5	1	2	1	0
6	0	0	2	1
7	1	1	3	2
8	0	2	0	3
9	1	0	1	4

7.2 CRACKING UP!

Finally, make the dungeon look more realistic by adding some cracks in the floor tiles. You can do this by drawing the cracks on top of the floor tile images. Make sure to add cracks on only a few tiles; you can choose these tiles at random. Start by importing Python's **random** module and add it to the top of your file. You need to use the `randint(a, b)` function from this module, which returns a random whole number between **a** and **b** (see box, right). You will choose random numbers in the `draw_background()` function and decide when to draw a crack based on them. Because the same squares need to be picked for the cracks every time the `draw_background()` function is called, set the "seed value" (see box, right) to a specific number at the start of the function.

```
import pgzrun
import random
```

Makes the functionality in the **random** module available

```
PLAYER_MOVE_INTERVAL = 0.1
BACKGROUND_SEED = 123456
```

Adds a new constant for the seed value at the top of the file

Tells the program to pick random numbers starting from `BACKGROUND_SEED`

```
def draw_background():
    random.seed(BACKGROUND_SEED)
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            if x % 2 == y % 2:
                screen.blit("floor1", screen_coords(x, y))
            else:
                screen.blit("floor2", screen_coords(x, y))
            n = random.randint(0, 99)
            if n < 5:
                screen.blit("crack1", screen_coords(x, y))
            elif n < 10:
                screen.blit("crack2", screen_coords(x, y))
```

Picks a random number between **0** and **99**

Checks if **n** is less than **5**

Draws `crack1` on top of the floor tile at this position if **n** is less than **5**

Checks if **n** is less than **10**

Draws `crack2` on top of the floor tile at this position if **n** is less than **10** but not less than **5**

RANDOM NUMBERS AND PROBABILITY

The `randint()` function returns a number in a specific range. Repeated calls will return numbers that are roughly distributed across this range. To be more precise, this function actually returns a pseudo-random number. These are numbers that appear random—in that the numbers are evenly distributed across the range and the sequence does not look obviously predictable—but are actually generated by an algorithm that will always generate the same sequence of numbers from a given starting point. You can call the starting point for a pseudo-random sequence the "seed". If you repeatedly pick random numbers between 0 and 99, then you should get the numbers 0 to 4 about 5 percent of the time. If you look at the example below, you will see that if **n** is a number between 0 and 4, **crack1** image will be drawn. So you can expect this to happen for about 5 percent of the floor tiles. If **n** is greater than 5 and lies between 5 and 9, **crack2** image will be drawn, which should also happen for about 5 percent of the tiles. If you look carefully at the map, you will be able to count 118 exposed floor tiles, so you should expect to see about six of each type of crack (5 percent of 118).

```
n = random.randint(0, 99)
if n < 5:
    screen.blit("crack1", screen_coords(x, y))
elif n < 10:
    screen.blit("crack2", screen_coords(x, y))
```

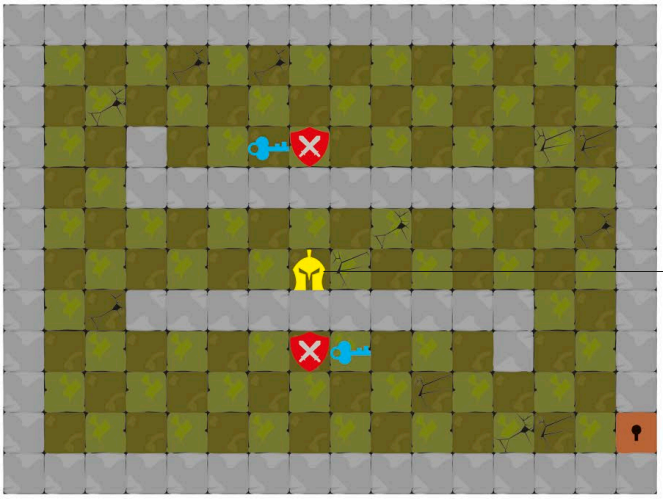
Tells the code to pick a random number between 0 and 99

Draws **crack1** if **n** is between 0 and 4

Draws **crack2** if **n** is between 5 and 9

7.3 TIME TO PLAY

The game should now be ready to play. Run the program to make sure it is working properly. If there is a problem, carefully check your code for bugs (see pp.130–133) and run it again.



Move the player quickly to collect all the keys. Watch out for the guards

SAVE

Hacks and tweaks

Opening the door

While users may immediately realize that they need to collect the keys, it may not be obvious when they can leave the dungeon. If you can visually open the door when the last key has been collected, it will be obvious what to do. The easiest way to do this is to only draw the door when there are no keys left to be collected. In the `draw_scenery()` function, change the logic for deciding when to draw the door as shown here.

```
            screen.blit("wall", screen_coords(x, y))
        elif square == "D" and len(keys_to_collect) > 0:
            screen.blit("door", screen_coords(x, y))
```

Checks if there are any keys left to be collected

Keep moving

It would help if the player could move continuously in one direction by holding down an arrow key instead of repeatedly pressing it. To do this, you can use the `on_finished` argument of `animate()` function. This allows the user to specify a function to be called when the actor has finished moving. Make a change in the `move_player()` function, as shown here, then add a new `repeat_player_move()` function below the `move_player()` function. It uses members of **Pygame Zero**'s `keyboard` object to check if a particular key is pressed. You might find that the game is now too easy. You can change the `PLAYER_MOVE_INTERVAL` constant at the top of the file to slow the player down and make the game more challenging.

```
            break
    animate(player, pos=screen_coords(x, y), \
            duration=PLAYER_MOVE_INTERVAL, \
            on_finished=repeat_player_move)
```

Checks if the arrow key is still pressed and repeats the move

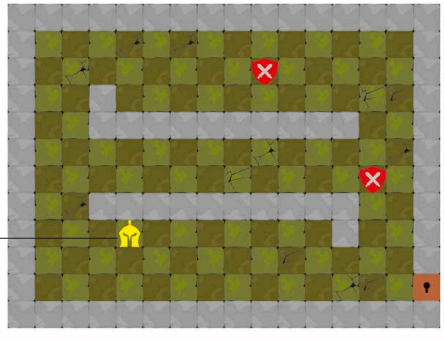
```
def repeat_player_move():
    if keyboard.left:
        move_player(-1, 0)
    elif keyboard.up:
        move_player(0, -1)
    elif keyboard.right:
        move_player(1, 0)
    elif keyboard.down:
        move_player(0, 1)
```

Checks if the left arrow key is still pressed

Calls `move_player()` again to repeat the move

Set this to half the guard's move interval, giving the player less of an advantage

```
GUARD_MOVE_INTERVAL = 0.5
PLAYER_MOVE_INTERVAL = 0.25
```



The player will now move continuously in the chosen direction

Make a bit more room

You can make the game more interesting by designing a larger, more complicated map. You can design your own, but try this one to get the idea. First, change the size of the grid by editing the values of the constants at the top of the file. Next, carefully edit the `MAP` constant as shown here. You can add as many guards or keys as you want because of the way you have written the code.

```
GRID_WIDTH = 20
GRID_HEIGHT = 15
```

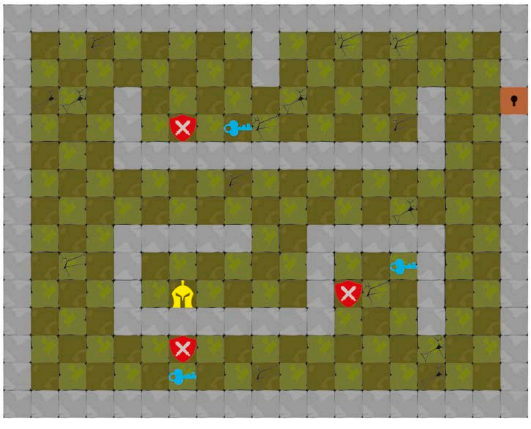
Increase the value of these variables

This dungeon has 20 columns, so there should be 20 `W` characters in this line

Remember to add the door in this line

```
MAP = ["WWWWWWWWWWWWWWWWWWWW",
       "W        w        W",
       "W        W        W",
       "W  W              W  D",
       "W  W G K          W  W",
       "W  WWWWWWWWWWWW   W",
       "W                 W",
       "W                 W",
       "W  WWWWW  WWWWW   W",
       "W  W        W  KW  W",
       "W  W P      WG  W  W",
       "W  WWWWWWWW   W   W",
       "W     G           W",
       "W     K           W",
       "WWWWWWWWWWWWWWWWWWWW"]
```

This dungeon has 15 rows, so there should be 15 lines in total



LARGE PLAYING AREA

Upload new characters

You can upload your own images to add different characters to the game or set it up in a completely new background. Copy the new images into the "images" folder you created earlier, then update the code so that the actor image names match the new file names.

ENEMY

PLAYER

ITEM