User input

To get user input from the keyboard, Python uses the `input()` function. This function normally takes a string as a parameter, which it displays to prompt the user to enter the required data. The function echoes the user's input to the screen as a string but does not save it. It therefore makes sense to assign the result of the input function to a variable so that it can be used later in the code.

This string is displayed onscreen asking the user for an input

```
>>> input("Enter your name: ")
Enter your name: Claire
'Claire'
```

The user enters a name, but the value is not saved, so it cannot be used later

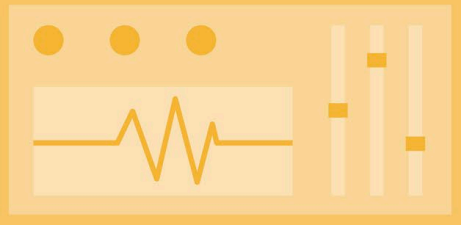
The user's response is saved, and the program can use it to do something else

```
>>> name = input("Enter your name: ")
Enter your name: Claire
>>> print("Hello " + name)
Hello Claire
```

The program prints the string "Hello" followed by the user's name

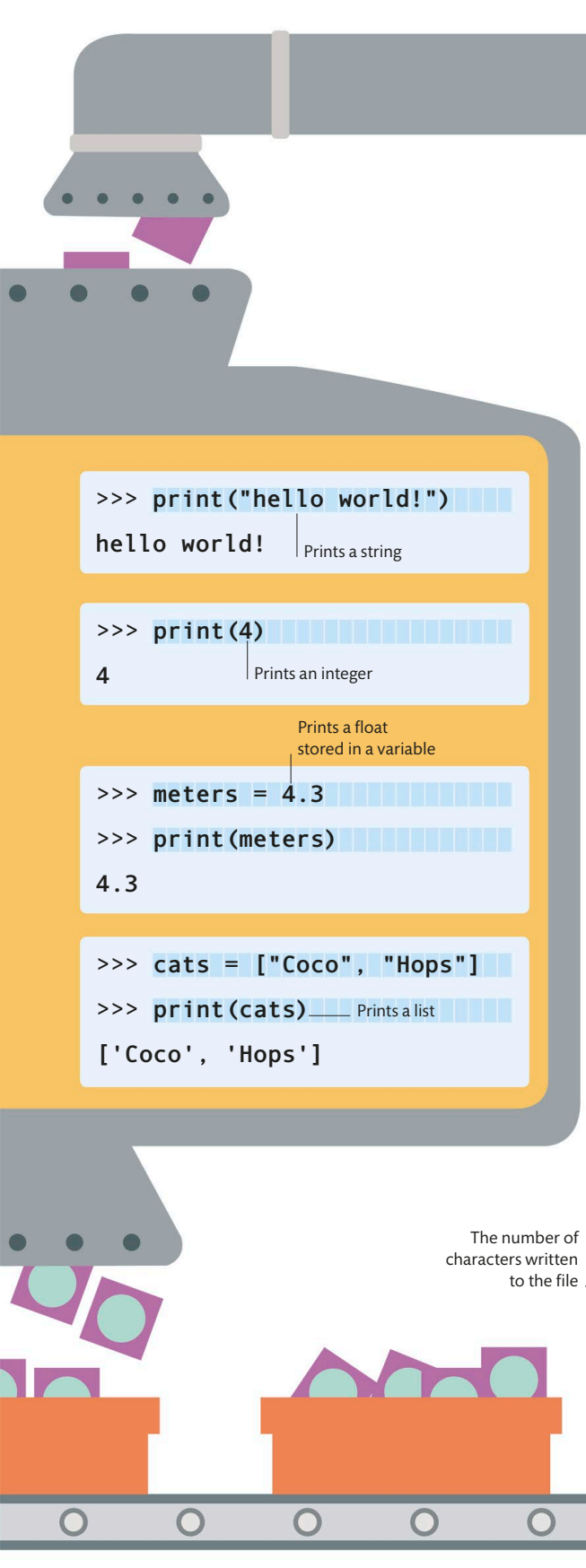
Output onscreen

The `print()` function is used to display a value on the screen. It is a versatile function that prints the value of any variable or expression regardless of its data type. Programmers often use it to debug code if their editor does not include a debugger (see pp.130–133). Printing out the value of variables at different points in the code can be useful, but it can also make the code untidy.



Input and output

There is no point writing a program unless it produces an output that can be read and understood. In Python, programs often require some input, either from a user interacting with the program as it runs or from a file.



```
>>> print("hello world!")
hello world!
```
Prints a string

```
>>> print(4)
4
```
Prints an integer

Prints a float stored in a variable

```
>>> meters = 4.3
>>> print(meters)
4.3
```

```
>>> cats = ["Coco", "Hops"]
>>> print(cats)
['Coco', 'Hops']
```
Prints a list

Input from file

Python has the ability to input data directly from a file. This is especially useful in programs that require a lot of data, and where it would not be feasible to have a user type in the required information every time the program runs. In Python, opening a file creates a file object. This file object can be saved in a variable and used to carry out various operations on the file's contents.

Opens the file and saves the file object in the variable **file**

```
>>> file = open("/Desktop/List.txt")
>>> file.read()
'High Street\nCastle Street\nBrown \
Street\n\n'
>>> file.close()
```
Reads the data from the file

"\n" prints each output on a new line

Closes the file

```
>>> file = open("/Desktop/List.txt")
>>> file.readline()
'High Street\n'
>>> file.readline()
'Castle Street\n'
```
Opens the file and reads one line at a time

Output to file

In Python, it is also possible to output data to a file. This is useful if a program produces a lot of data or if it is updating or adding to an existing input file. When opening the file, the programmer must indicate that data will be added to it and whether it should be written before or after existing data.

The number of characters written to the file

Opens the file for data to be appended to the end

This means "append"

```
>>> file = open("/Desktop/List.txt", "a")
>>> file.write("Queen Street")
12
>>> file.close()
```
Writes the new value "Queen Street" to the file

Loops in Python

Programs often contain blocks of code that are repeated several times. Rather than typing the same lines over and over again, programmers use loops. The type of loop they choose depends on a number of factors.

For loop

If a programmer knows the number of times a block of code will be repeated, a **for** loop is used. The code that gets repeated is called the body of the loop, and each execution is called an iteration. The body is always indented from the **for** statement and begins exactly four spaces from the start of the next line. Indentation can also be done manually.

Loop variable
This example loop counts from one to three, printing each number on a new line, followed by a line saying "Go!". The loop variable keeps track of loop iterations. It takes the value of each item in `range(1,4)` in a set order, starting with the first value for the first iteration.

```
for counter in range(1,4):
    print(counter)
print("Go!")
```

This is like a list that has the values 1, 2, 3

This statement is the loop body

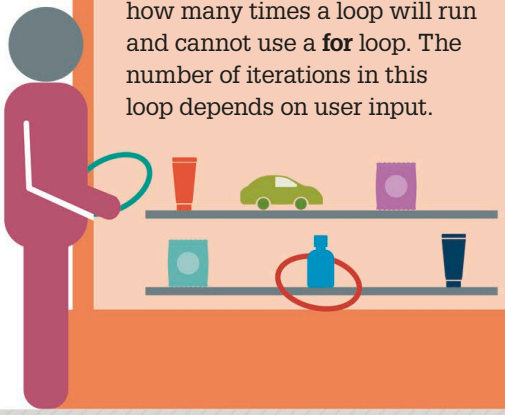
For loop with a list
To process a list using a **for** loop, there is no need to use the `range()` function. Python can simply set the value of the loop counter to each item in the list in turn. In this example, the loop prints out each name in the list `red_team`.

```
red_team = ["Sue", "Anna", "Emily", "Mateo"]

print("The Red Team members are:")
for player in red_team:
    print(player)
```

`player` is the temporary loop counter

While loops

Another type of loop in Python is the **while** loop. It is used when a programmer does not know how many times a loop will run and cannot use a **for** loop. The number of iterations in this loop depends on user input.



This question will appear, asking for user input

```
answer = input("Should I keep going? (y/n)")
while answer == "y":
    answer = input("Should I keep going? (y/n)")
```

The question is asked again

Loop condition
A **while** loop includes a question called a loop condition, the answer to which is either True or False. The body of the loop is executed only if the loop condition on that iteration is True. If the loop condition is False, the **while** loop ends.

Infinite loops

While loops are also used in situations where the loop is supposed to run as long as the program is running. This is called an infinite loop and is commonly used in programs for games. To create an infinite loop, the user needs to set the loop condition to True.

```
while True:
    print("There's no stopping me now!")
```

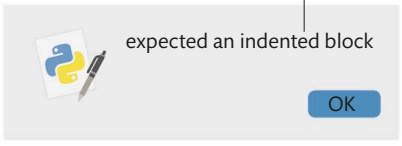
Prints the line repeatedly

Getting stuck
Unintentional infinite loops are the coding equivalent of a black hole. Program execution gets stuck, making the computer unresponsive.

INDENT THE BODY

Similar to a **for** loop, the body of a **while** loop is also indented four spaces from the **while** keyword. If this indentation is missing, Python produces an error message saying "expected an indented block".

Missing indentation produces this error



Emergency escape

Sometimes programmers create infinite loops by accident. This can happen if the loop condition never becomes False in any iteration. An infinite loop can be easily stopped from Python's shell window by holding down the "Control" key and pressing "C".

Sets the variable `number` to 1

Variable value
The loop condition will always be True if there is no instruction to increase the value of the variable `number` in the loop body.

```
number = 1
while number < 10:
    print(number)
```

Nested loops

The body of a loop can contain another loop within itself. This loop inside a loop is called a nested loop. In Python, any type of loop can be contained inside any other type of loop—so a **while** loop can contain another **while** loop or a **for** loop, and vice versa. In this example, every time the body of the outer loop runs, the body of the nested loop is executed 10 times, printing out a countdown to launch.

```
answer = input("Launch another spacerocket? (y/n)")
while answer == "y":
    for count in range(10, 0, -1):
        print(count)
    print("LIFT OFF!")
    answer = input("Launch another spacerocket? (y/n)")
```

The nested loop

This contains the list 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 and counts down from 10

This line updates the variable **answer** during each iteration, making it possible to exit the loop when required

The loop stops if user input is **n**

Escaping loops

While coding, a programmer might want to exit a loop early, or may decide to abandon the current iteration of the loop body and move onto the next one. In such scenarios, there are two related commands, **break** and **continue**, that can be used to escape the loop.

Break

The **break** command abandons the loop completely, and the code continues to execute from the instruction after the loop. In this example, the loop stops when a negative value is encountered in the list **sensor_readings**. The code discards this negative value and any values following it.

```
sensor_readings = [3, 5, 4, -1, 6, 7]
total = 0

for sensor_reading in sensor_readings:
    if sensor_reading < 0:
        break
    total = total + sensor_reading
print("total is: " +str(total))
```

Gives a total of all **sensor_readings** up until the negative reading

Lists of lists

Nested loops are often used when processing lists within lists. The example below prints the names of team members for three different teams. When the code runs, the loop variable **team** in the outer loop takes the value of the first list, which is ["Red Team", "Adam", "Anjali", "Colin", "Anne"]. The loop variable **name** in the inner loop then takes the values in **team** one after the other, until every name in the red team has been printed. Each team listing is separated by an empty line.

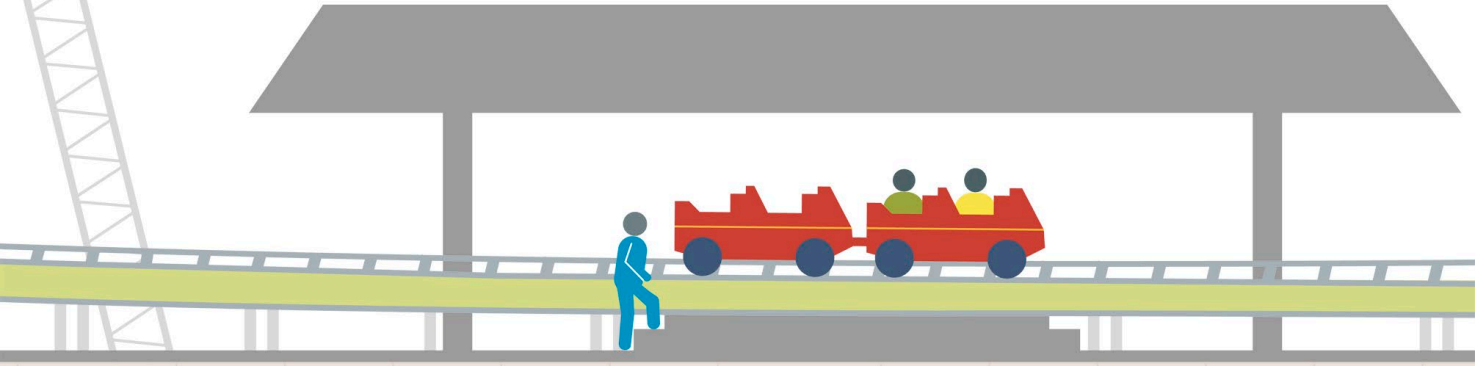
```
teams = [["Red Team", "Adam", "Anjali", "Colin", "Anne"], \
         ["Blue Team", "Greg", "Sophie", "June", "Sara"], \
         ["Green Team", "Chloe", "Diego", "Mia", "Jane"]]

for team in teams:
    for name in team:
        print(name)
    print("\n")
```

The list **teams** contains three different lists enclosed within square brackets

Selects a team to process

Prints an empty line between each team listing

Prints the names in a team one after the other



```
readings = [3, 5, 4, -1, 6, 7]
total = 0

for reading in readings:
    if reading < 0:
        continue
    total = total + reading
print("total is: " +str(total))
```

Initializes the variable **total** by setting its value to **0**

Triggers the **continue** command after reading a negative value

Gives a sum of all the non-negative readings

Continue

The **continue** command is less drastic and only abandons the current iteration of a loop. The loop then skips to the next iteration. In this example, if a negative value is encountered, it is simply ignored, and the loop jumps to the next value in the list.

Functions

Pieces of code that carry out a specific task are called functions. If the task is executed often, it is possible to separate it from the main code to avoid typing identical instructions multiple times. Breaking the code into sections this way also makes it easier to read and test the program.

Using functions

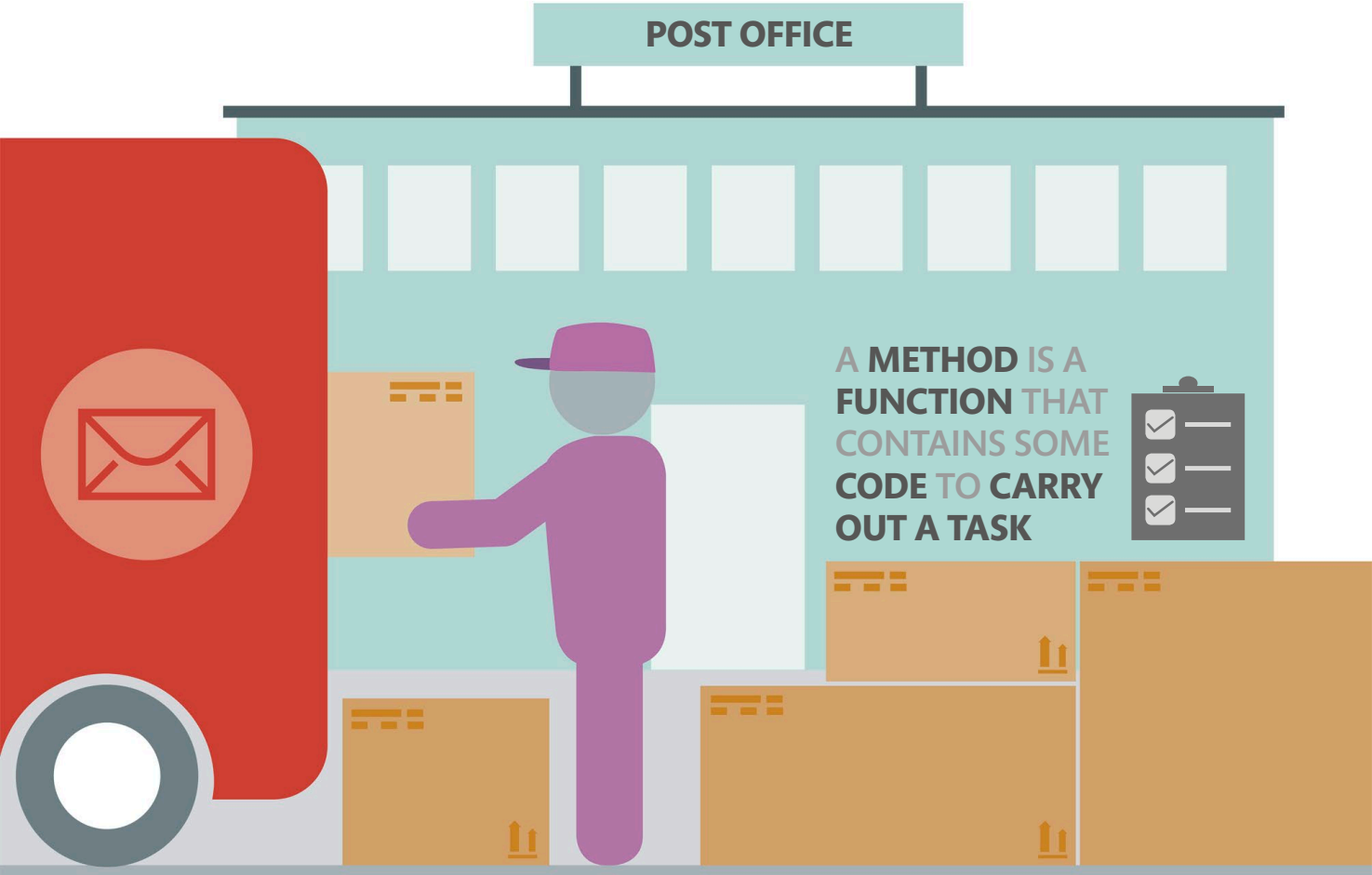
Using a function is also known as "calling" it. Most of the time, this is as simple as typing the function's name followed by a pair of brackets. If the function takes a parameter, it goes inside the brackets. A parameter is a variable or value that is given to the function to allow it to carry out its task.

Defining a function
When a function is defined (see pp.114–115), it always has the keyword "def" and the function's name at the start of the code block.

```
def greeting():
    print("Hello!")
```

The keyword **def** tells Python that this block of code is a function

The parameter of the function



Built-in functions

Python includes a range of built-in functions that enable the completion of basic tasks. These include getting input from the user, displaying output onscreen, simple arithmetic, and determining the length of a string or list. The examples below can all be tried in IDLE's shell window.

`input()` function's parameter is a question that prompts the user for input

```
>>> name = input("What is your name? ")
What is your name? Tina
>>> print("Hello " + name)
Hello Tina
```

User inputs response

`print()` function's parameter is a string that is displayed onscreen

input() and print()
The `input()` function gets data from the user, and the `print()` function displays it as output on the screen. The parameter for `input()` is displayed onscreen to prompt the user.

Number to be rounded

```
>>> pi = 22/7
>>> pi
3.142857142857143
>>> round(pi, 2)
3.14
```

Decimal places

round()
This function rounds off a float to a specific number of decimal places. It takes two parameters—the number to be rounded, and the number of decimal places to shorten it to.

Calling another way

Built-in functions, such as `print()` or `len()`, can be easily called because they accept parameters of various types. A method is a function associated with a particular object and can only be used on that object (see pp.156–157). Calling a method is different from calling a built-in function. A method call has the object's name, a period, and the method name followed by a pair of brackets.

upper() method
This method transforms all the lowercase letters in a string to uppercase letters. The `upper()` method can only be used with strings.

Object name

```
>>> city = "London"
>>> city.upper()
'LONDON'
```

The bracket may take a parameter

Method name

Adding to a list
The list method `append()` adds a value to the end of a list. It has one parameter—the value that needs to be appended to the list.

```
>>> mylist = [1,2,3,4]
>>> mylist.append(5)
>>> print(mylist)
[1, 2, 3, 4, 5]
```

The new value is added to the end of the list

Creating functions

Python has a Standard Library that contains a lot of premade functions. Most programs, however, include functions that have to be specifically made for them. In Python, creating a function is known as "defining" it.

```
def print_temperature_in_Fahrenheit(temperature_in_Celsius):
    temperature_in_Fahrenheit = temperature_in_Celsius * 1.8 + 32
    print(temperature_in_Fahrenheit)
```

Defines a function that takes a temperature in Celsius and prints it in Fahrenheit

This formula converts Celsius to Fahrenheit

Prints the temperature in Fahrenheit

A function that completes a task

Some functions simply carry out a task without returning any information to the code that called them. This is similar to sending a letter by regular mail. A postal worker delivers the letter and completes his task but does not inform the sender that it has been delivered.

Top-down coding

In Python, functions are normally defined at the top of the program, before the main code. This is because it is important to define a function before it is called either by another function or by the main part of the code.

```
def function_a():
    return 25

def function_b():
    answer = 2 * function_a()
    return answer

number = function_a() + function_b()
print(number)
```

`function_a` is called from inside `function_b`

This runs because both `function_a` and `function_b` have already been defined

Order of definition

Because the main part of the code calls both `function_a` and `function_b`, they must be defined at the start of the program. As `function_a` relies on `function_b`, `function_a` must be defined before `function_b`.

NAMING FUNCTIONS

Similar to the rules for naming variables, Python also has a number of rules for naming functions. Just as it is important for a function to have a clear task or purpose, it is also important for the function to have a name that clearly describes what it does. So `get_number()` is not as good a name as `get_number_of_winners()` for a function that returns the value of the number of people who have won in a competition. If there are several words in a name, they should be typed in lowercase and separated by underscores.

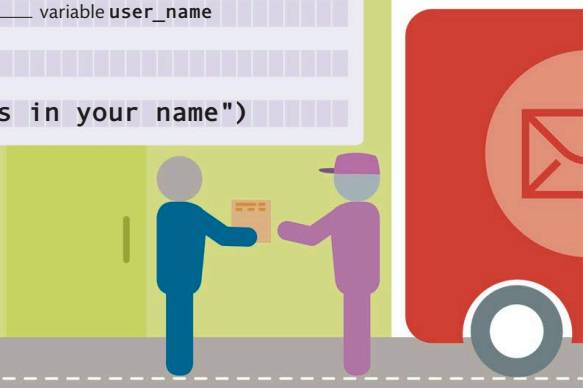
```
def count_letter_e(word):
    total_e = 0
    for letter in word:
        if letter == "e":
            total_letter_e = total_letter_e + 1
    return total_letter_e

user_name = input("Enter your name: ")
total_es_in_name = count_letter_e(user_name)
print("There are " + str(total_es_in_name) + "E's in your name")
```

Defines a function that counts and returns the number of times the letter "e" appears in a particular word

This loop examines each letter in the word being investigated

Asks users to enter their name, then stores it in the variable `user_name`

A function that returns a value
There are also functions that carry out a task and produce a value, which is then returned to the code that called them. This enables the calling code to store the value in a variable, if necessary.



Local and global variables

A global variable is declared in the main part of the code and is visible everywhere. A local variable, on the other hand, is declared inside a function and is only visible there. For example, global variables are like divers: visible to everyone under the sea, including people in submarines. Local variables, however, are like people in the submarines: only visible to other people in that submarine. Global variables can be read by other functions in the code, but local variables cannot. The code will return an error message if a local variable is used outside of its function. A function must declare the global variable it intends to use or else Python will create a new local variable with the same name.

```
def reset_game():
    global score, charms, skills
    score = 0
    charms = 0
    skills = 0
```

Declares the global variables that this function will be using

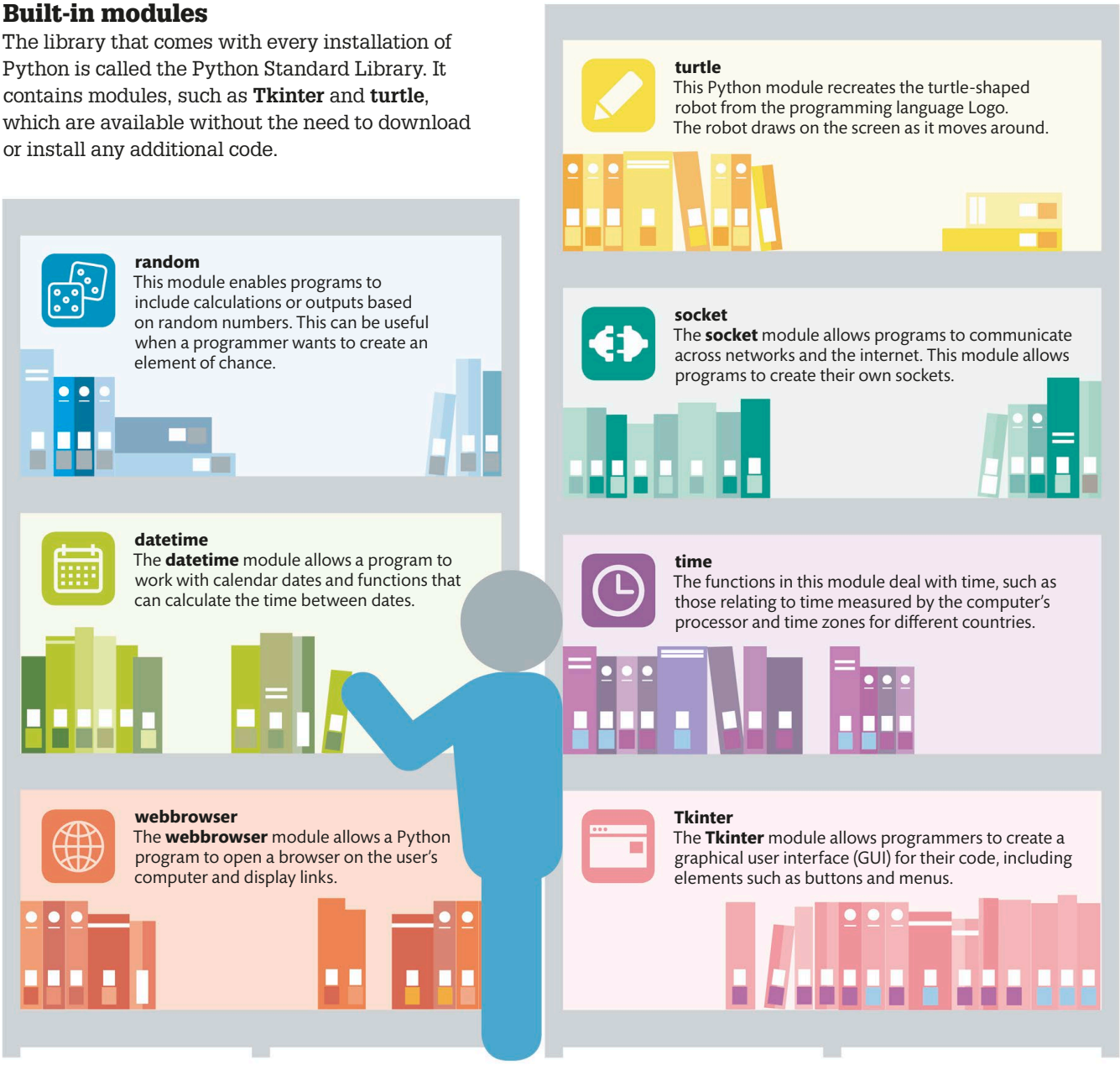
reset_game()
This function resets a game by setting the value of the global variables `score`, `charms`, and `skills` back to `0`.

Libraries

A Python library is a collection of files, known as modules, that are available for other programmers to use. These modules contain code for common programming tasks, ranging from interacting with hardware to accessing web pages.

Built-in modules

The library that comes with every installation of Python is called the Python Standard Library. It contains modules, such as **Tkinter** and **turtle**, which are available without the need to download or install any additional code.



Importing and using modules

The process of adding a module to a program so that its functions and definitions can be used is called "importing." In Python, it is possible to import either an entire module or just certain functions of a module. The method used for carrying out the import depends on the requirement of the program. The examples below illustrate the different methods for importing and the required syntax in each case.

import ...
The keyword import followed by the module's name makes all of the module's code available to the program. To access the module's functions, it is necessary to type the imported module's name followed by a period before the function name in order to call that function.

```
import time

offset = time.timezone
print("Your offset in hours from \
UTC time is: ", offset)
```

Calls the `timezone` function of the `time` module

Prints the value in the variable `offset`

from ... import ...
If a program only needs to use one or two functions from a module, it is considered better just to import these and not the whole module. When functions are imported in this way, it is not necessary to include the name of the module before the function name.

```
from random import randint

dice_roll = randint(1,6)
print("You threw a", dice_roll)
```

The `randint()` function produces a random integer between 1 and 6

from ... import ... as ...
If the name of a function in the module is too long or is similar to other names in the code, it can be useful to rename it. Just as in "from ... import ...", this allows the programmer to refer to the function simply by its new name without preceding it with the name of the module.

```
from webbrowser import open as show_me

url = input("enter a URL: ")
show_me(url)
```

Displays the user's choice of web page

PYGAME

The **pygame** library contains a large number of useful modules for coding games. Because **pygame** is not part of the Standard Library, programmers have to download and install it before they can import it to their code. **pygame** is very powerful but can be challenging for new programmers. One solution to this is the **Pygame Zero** tool (see pp.176–177), which makes the functions in **pygame** easier to use.

Team allocator

When playing team sports, the first thing you have to do is to pick the teams. One way of doing this is to choose team captains and let them choose the players for their teams. However, it might be fairer to pick people randomly. In this project, you'll automate this process by building a tool in Python that picks teams randomly.

How it works

This project will use Python's **random** module to form teams with randomly selected players. You will use lists (see p.103) to store the player's names. The **random** module will then shuffle this list into a different order. Loops will be used to iterate through the list and display the players. Finally, an **if** statement (see p.105) checks to see if the user is happy with the selection.

Random allocation
This project will pick two teams and a captain for each team. When you run the program, it will display the chosen teams and captains on the screen.

Python 3.7.0 shell

```
Welcome to Team Allocator!
Team 1 captain: Rose
Team 1:
Jean
Ada
Sue
Claire
Martin
Harry
Alice
Craig
Rose
James
```

The list of players is displayed in the shell window

YOU WILL LEARN

- How to use the **random** module
- How to use lists
- How to use loops
- How to use branching statements

Time: 15–20 mins

Lines of code: 24

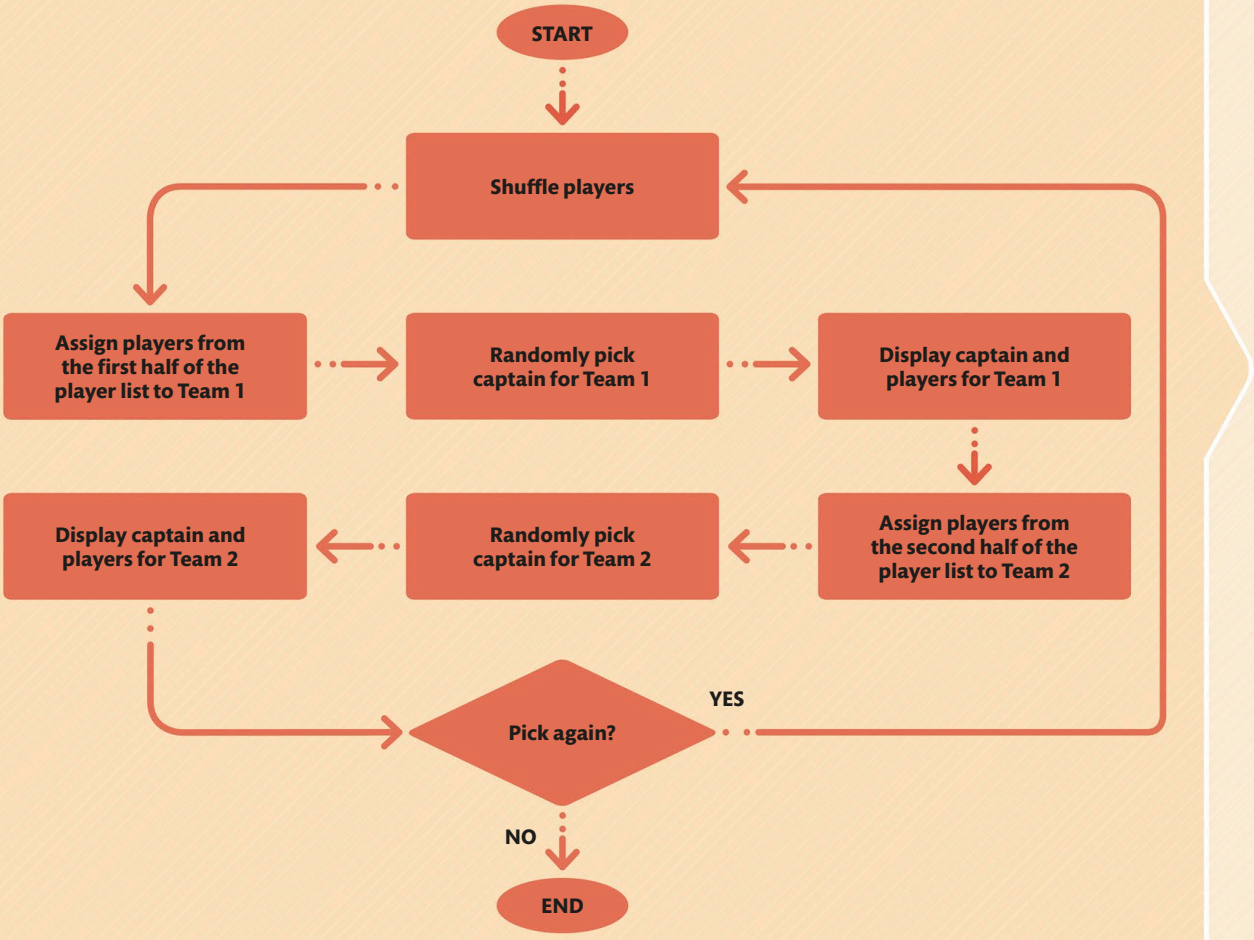
Difficulty level

WHERE THIS IS USED

The code in this project can be reused for tasks that require random allocations. This includes staff shift scheduling, assigning tasks in the workplace, matching people to projects, selecting teams for a quiz, and many more. This can be a quick and fair way of allocating people to teams/tasks.

Program design

The program begins by shuffling the player list. It then allocates the first half as Team 1, randomly selects a captain, and displays the name of the captain along with the names of the rest of the team. The steps are then repeated for the second half of the list, forming Team 2. If you want to pick the teams again, the program repeats the steps; otherwise, the program ends.

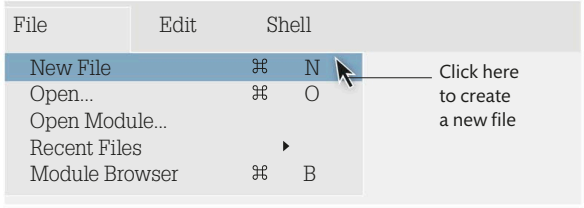


1 Create a team

This program will simplify the process of picking, or allocating, teams. In this section, you will create the file that will contain the code, import a module, and then make a list of players.

1.1 CREATE A NEW FILE

The first step is to open IDLE. A shell window will appear. Ignore it and click on File in the IDLE menu. Next, choose New File and save the file as "team_selector.py". This will create an empty editor window where you can write your program.



1.2 ADD THE MODULE

Now, import the **random** module. Type this line at the top of your file so that you can use the module later. This module contains functions that will allow you to pick players randomly from a list.

```
import random
```

The **random** module can pick random numbers or shuffle a list in a random order

1.3 WELCOME THE USER

Next, create a message to welcome the user to the program. This will show a message to the user when the program executes. Save the file and then run the program to ensure your code works. From the Run menu, choose Run Module. If you have typed in the code successfully, the welcome message should appear in the shell window.

```
print("Welcome to Team Allocator!")
```

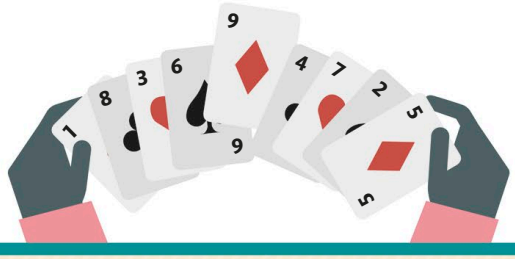
This phrase will appear as the welcome message in the shell window

SAVE

```
Welcome to Team Allocator!

>>>
```

RANDOM NUMBERS

Random numbers can be used to simulate anything that can be random or is meant to be random—for example, rolling dice, tossing a coin, or picking a card from a deck. Python's **random** module helps add an element of chance to a program. You can read more about how to use this module in the Docs section of IDLE's Help menu.



1.4 MAKE A NAME LIST

You will need names for all of the players to generate your teams randomly. In Python, you can keep groups of related items together in a list (see p.103). First, create the variable `players` to store the list by typing this new block of code below the import statement. Put the list contents in square brackets and separate each item in the list with a comma.

```
import random

players = ["Martin", "Craig", "Sue",
           "Claire", "Dave", "Alice",
           "Luciana", "Harry", "Jack",
           "Rose", "Lexi", "Maria",
           "Thomas", "James", "William",
           "Ada", "Grace", "Jean",
           "Marissa", "Alan"]
```

The list is assigned to the variable `players`

You do not need to use a backslash (\) to split a list across two lines. Pressing return or Enter indents the next line in a list

Each item in the list is a string enclosed in quotation marks

This project has 20 players in the list. You can change the number of players if you like (see p.127)

1.5 SHUFFLE THE PLAYERS

There are a few ways in which the players can be randomly selected. You could randomly keep picking players and assign them to the two teams until you run out of players. This program assumes the number of players is even. However, an even simpler way would be to just shuffle the list of players randomly and assign the first half of the list to "Team 1" and the second half to "Team 2". To do this, the first thing you have to do is shuffle the players. Use the **`shuffle()`** function from the **random** module. Type this code below the print command.

```
print("Welcome to Team Allocator!")

random.shuffle(players)
```

This will shuffle the list of players just like you would shuffle a deck of cards

2 Pick teams

Now that the list of players is ready, you can split the players into two teams. You will then assign the team captains. The teams and the names of their captains will be displayed onscreen when the program is executed.

SPLITTING LISTS

In Python, when splitting or taking subsets of a list, you need to provide two arguments: the start index (position) and the index after the last item in the new list. Remember, indexes start from 0 in Python (see p.103). For example, `players[1:3]` would take the players from index 1 up to index 2. The first index is inclusive (it is included in the new list) and the second index is exclusive (it is included up to the item before it in the new list). If you are splitting the list from the first position up to the last position, then you can leave those indexes blank, as Python will understand this. For example, `players[:3]` will take the first three players from the list and `players[4:]` will take the players from index 4 up to the end of the list.

2.1 SELECT THE FIRST TEAM

You now need to split the list into two equal parts. To do this, take the items in the list from position 0 up to the last item in the list and divide it by two. Add the following code at the end of the file for welcoming the user. This will create a new list with the first half of the players list.

```
team1 = players[:len(players)//2]
```

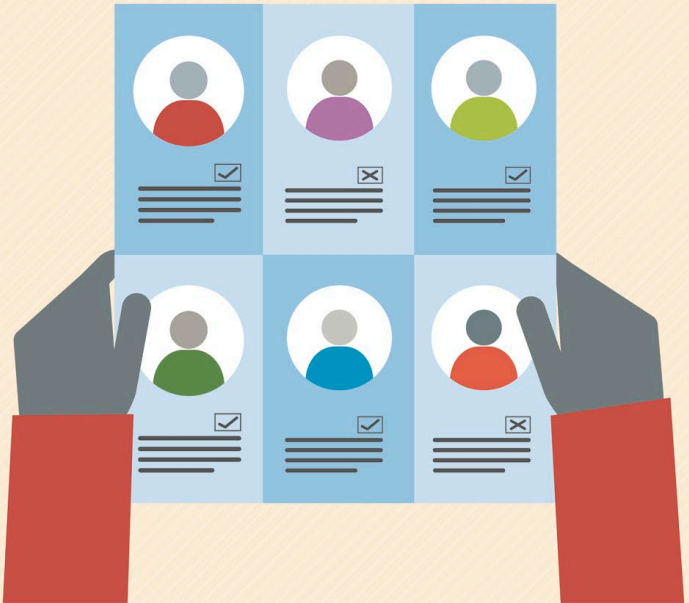
This new list will be assigned to the variable `team1`

2.2 SELECT TEAM 1 CAPTAIN

Once you have allocated the first team, you need to choose the team captain. To make it a fair selection, this will also be done randomly. A player from `team1` will be picked and assigned to be the team captain. Use the `choice()` function to pick a player randomly from `team1`. Type this code at the end of the file. The captain is randomly selected from the `team1` list using the `choice()` function and appended to the string to be displayed.

```
print("Team 1 captain: " + random.choice(team1))
```

Prints the message stating who the team captain is



2.3 DISPLAY TEAM 1

After the captain is assigned, you need to display all of the players from "Team 1" onscreen. Remember, you can use a **for** loop (see p.108) to iterate through a list. Type the following code at the end of the file.

Prints a message to tell the user that the players for Team 1 are being displayed

```
print("Team 1:")
for player in team1:
    print(player)
```

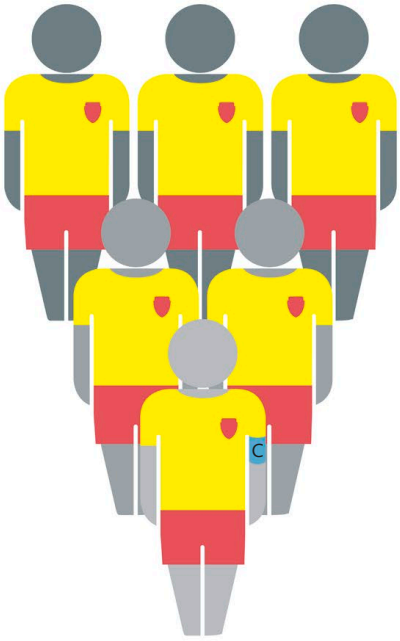
This loop iterates through `team1`

Prints the current player's name

SAVE

2.4 TEST THE PROGRAM

This is a good point to test your code. Run the code and look in the shell window to see the result. Does it display the players for Team 1? Does it display the number you expected? Is it randomly selecting a team captain that is actually part of Team 1? Run the code a few times to ensure it is random. If you have any errors, look back over your code carefully to spot any mistakes.



```
Welcome to Team Allocator!
Team 1 captain: Claire
Team 1:
Maria
Jean
William
Alice
Claire
Jack
Lexi
Craig
James
Alan
>>>
```

2.5 SELECT THE SECOND TEAM

Now you can allocate players for the second team by repeating steps 2.1–2.3. Type the following code at the end of the file.

Assigns the second half of the list to the variable **team2**. The players in this list will be part of the second team

```
team2 = players[len(players)//2:]
print("\nTeam 2 captain: " + random.choice(team2))
print("Team 2:")
for player in team2:
    print(player)
```

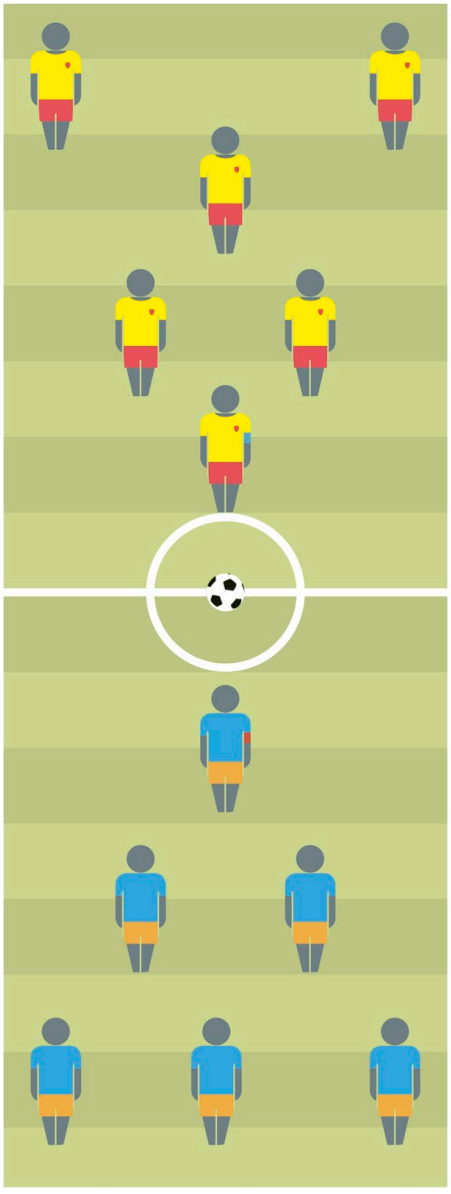
"\n" prints the name of the team captain for Team 2 on a new line

This loop iterates through **team2**

SAVE

2.6 TEST THE PROGRAM

Run the code to test the program again. Ensure that it is working as expected for both teams. You should be able to see the list of players for both of the teams along with the names of their captains.



```
Welcome to Team Allocator!
Team 1 captain: Marissa
Team 1:
Harry
Claire
Jack
Sue
Dave
Craig
Marissa
Grace
Alan
Maria

Team 2 captain: James
Team 2:
Martin
Jean
Alice
Ada
William
Rose
Lexi
James
Luciana
Thomas
>>>
```

The name of the captain will be displayed before the list of players

3 PICK NEW TEAMS

You can now use a **while** loop to keep selecting teams until you are happy with them. Add a new line of code below the welcome message and remember to add indents for all of the lines of code following this new line, as shown below. This will ensure that the existing code is part of the **while** loop.

```
print("Welcome to Team Selector!")
while True:
    random.shuffle(players)
    team1 = players[:len(players)//2]
    print("Team 1 captain: "+random.choice(team1))
    print("Team 1:")
    for player in team1:
        print(player)
    team2 = players[:len(players)//2:]
    print("\n Team 2 captain: "+random.choice(team2))
    print("Team 2:")
    for player in team2:
        print(player)
```

Adds the loop that allows selecting the teams again

Add indents to these lines of code to make them part of the loop

3.1 REDRAW PLAYERS

Finally, add the following code to ask users if they would like to pick teams again. Store the reply in a variable called `response`. If you choose to redraw the players, the main loop will run again and display the new teams.

Displays a message to ask users if they would like to redraw the players

```
    response = input("Pick teams again? Type y or n: ")
    if response == "n":
        break
```

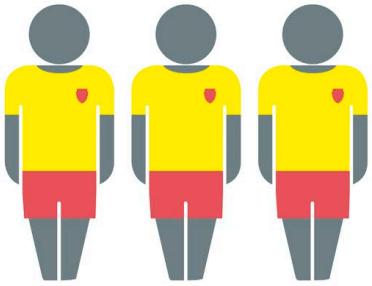
Breaks out of the main loop if the response is **n**

SAVE

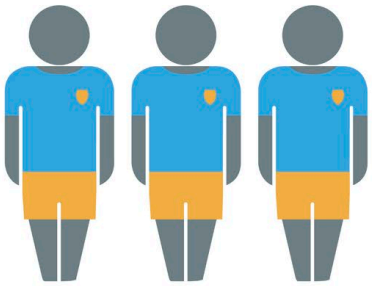
3.2 **RUN THE CODE**

The program is now ready. Test the program again. You will see the list of both teams with the team captains and a message at the end asking if you would like to redraw the players.

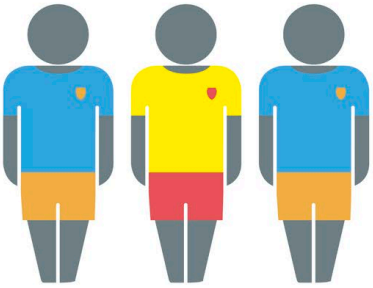
```
Welcome to Team Allocator!
Team 1 captain: Rose
Team 1:
Jean
Ada
James
Claire
Martin
Harry
Alice
Craig
Rose
Luciana

Team 2 captain: William
Team 2:
Jack
Maria
Sue
Alan
Dave
Grace
Marissa
Lexi
Thomas
William
Pick teams again? Type y or n:
```



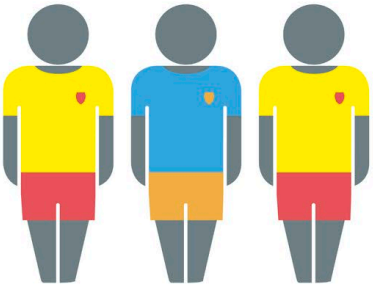
TEAM 1



TEAM 2



RESHUFFLED TEAM 1



RESHUFFLED TEAM 2

Hacks and tweaks

Add more players
The program has a list of 20 names. To add more players to the team selector, try adding some more names to the list. Keep the total number of players even so that the teams have an equal number of players on them.

More teams
Different sports have a different numbers of players on their teams. The code in this project assumes that there will be two teams. However, if you have a longer list of players, you can even have three or more teams. Update the code in the program to ask the user for the number of players they want on each team. You can then split the number of players into the number of teams they can equally be split into. If a team is short of players, make sure to inform the user of this.

```
while True:
    random.shuffle(players)
    team1 = players[:len(players)//3]
    print("Team 1 captain: " + random.choice(team1))
    print("Team 1:")
    for player in team1:
        print(player)
    team2 = players[len(players)//3:(len(players)//3)*2]
    print("\nTeam 2 captain: " + random.choice(team2))
    print("Team 2:")
    for player in team2:
        print(player)
    team3 = players[(len(players)//3)*2:]
    print("\nTeam 3 captain: " + random.choice(team2))
    print("Team 3:")
    for player in team3:
        print(player)
```

Splits the number of players into three equal parts and assigns the first part of the players list to `team1`

Assigns the second part of the players list to `team2`

Assigns the third team with its own list of players and the team captain

Team or tournament

Currently the program assumes that the code is for a team sport. If you want to create a program for individual sports, change the code as shown below. This will ask the user if the players need to be split for an individual or team sport. If you pick team, the code should run as you have already tested. However, if you pick "individual", the code will split the players into random pairs to play against each other.

```
print("Welcome to Team/Player Allocator!")
while True:
    random.shuffle(players)
    response = input("Is it a team or individual sport? \
                      \nType team or individual: ")
    if response == "team":
        team1 = players[:len(players)//2]
```

Displays a message to ask the user if it is a team or an individual sport

Checks for the user's response

```
    for player in team2:
        print(player)
    else:
        for i in range(0, 20, 2):
            print(players[i] + " vs " + players[i+1])
```

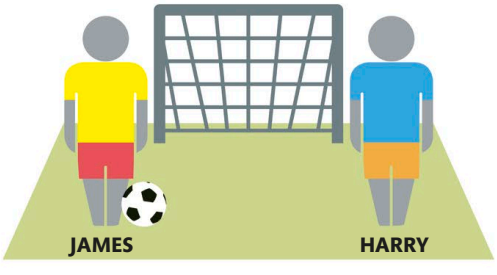
Range will take the value 0-19 and will increment by 2 each time. This is so we go through the list two players at a time to put them in pairs

Prints the name of players that will play against each other

Who starts?

For both team and individual sports, there is usually a method to determine who will go first. Add this extra code to the program from the previous hack to do this for individual sports.

```
            print(players[i] + " vs " + players[i+1])
        start = random.randrange(i, i+2)
        print(players[start] + " starts")
```




```
Welcome to Team/Player Selector!
Is it a team or individual sport?
Type team or individual: individual
James vs Harry
James starts
```

The shell window displays who starts

Change to list of numbers
The current program is only a good solution if you always play with the same people. However, if this is not the case, you can replace the player names with numbers to make it a more general solution. Remember to assign the numbers to the players before you use it.

```
import random

players = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
           11, 12, 13, 14, 15, 16, 17, 18,
           19, 20]

print("Welcome to Team Allocator!")
```

Update the code to replace the names with numbers

Number of players
Instead of having to change the size of the list each time you have more or fewer players, you can update the code to ask the user for the total number of players. This will create the number list, as well as create two equal teams. Update the program as shown here.

```
import random
players = []
print("Welcome to Team Allocator!")
number_of_players = int(input("How many players \
are there? "))
for i in range(1, number_of_players + 1):
    players.append(i)
```

Displays a message for the user to enter the number of players

```
    team1 = players[:len(players)//2]
    print("Team 1 captain: " + str(random.choice(team1)))
    print("Team 1:")
```

Updates code for **team1**

```
    team2 = players[len(players)//2:]
    print("\nTeam 2 captain: " + str(random.choice(team2)))
    print("Team 2:")
```

Updates code for **team2**

Debugging

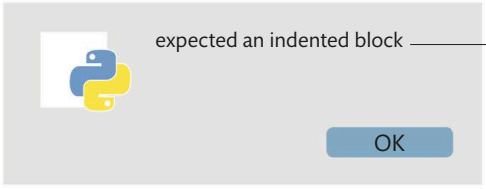
The process of finding and fixing errors in a program is called debugging. Also known as bugs, errors can range from simple mistakes in spelling to problems with the logic of the code. Python has various tools that highlight and help fix these errors.

Syntax errors

Syntax is the term used to describe the arrangement and spelling of words and symbols that make up the code. Syntax errors are the most common and easily fixed errors. They are equivalent to the sort of spelling and grammar mistakes that most word-processing programs highlight. IDLE displays syntax errors in code in a pop-up window.

```
temperature = 25

if temperature > 20:
print("Weather is warm")
```

This line in the code should be indented



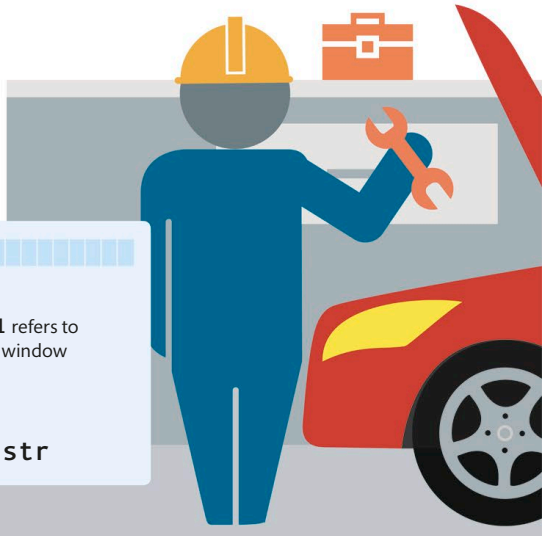
This error message indicates an indentation error in the code

Indentation errors

Indentation is a way of making a program more readable by using space to reflect its structure. The body of a function, loop, or conditional statement should be placed four spaces to the right of the line introducing it. Python makes indentation compulsory in the code.

Runtime errors

These errors affect the fundamental features of a program. They can include accessing a nonexistent file, using an identifier that has not been defined, or performing an operation on incompatible types of values. Runtime errors cannot be found by checking a program's syntax. The Python interpreter discovers them while running the code and displays an error message called a "traceback" in the shell window.



Type errors

These errors occur when a function or operator is used with the wrong type of value. The "+" operator can either concatenate two strings or add two numbers. It cannot, however, concatenate a string and a number, which is what causes the error in this example.

```
>>> "temperature" + 5
Traceback (most recent call last):
  File "<pyshell#3", line 1, in <module>
    "temperature" + 5
TypeError: can only concatenate str (not "int") to str
```

`pyshell` refers to the shell window

CHECK MODULE

IDLE's "Check Module" command can be found in the Run menu. It checks a program file for syntax errors, allowing programmers to identify and eliminate them before the program is run. This tool does not display any message unless it finds an error.

Name errors
Misspelling the name of a variable or function can cause a name error. It can also be a result of using a variable before a value is assigned to it or calling a function before it is defined. In this example, the typographical error is only found at run time, so the message is displayed in the shell window.

Details of where and what the error is

Location of the error in the file

```
>>> pront ("Hello world")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in module>
    pront("Hello world")
NameError: name 'pront' is not defined
```

The command that caused the error

Logic errors

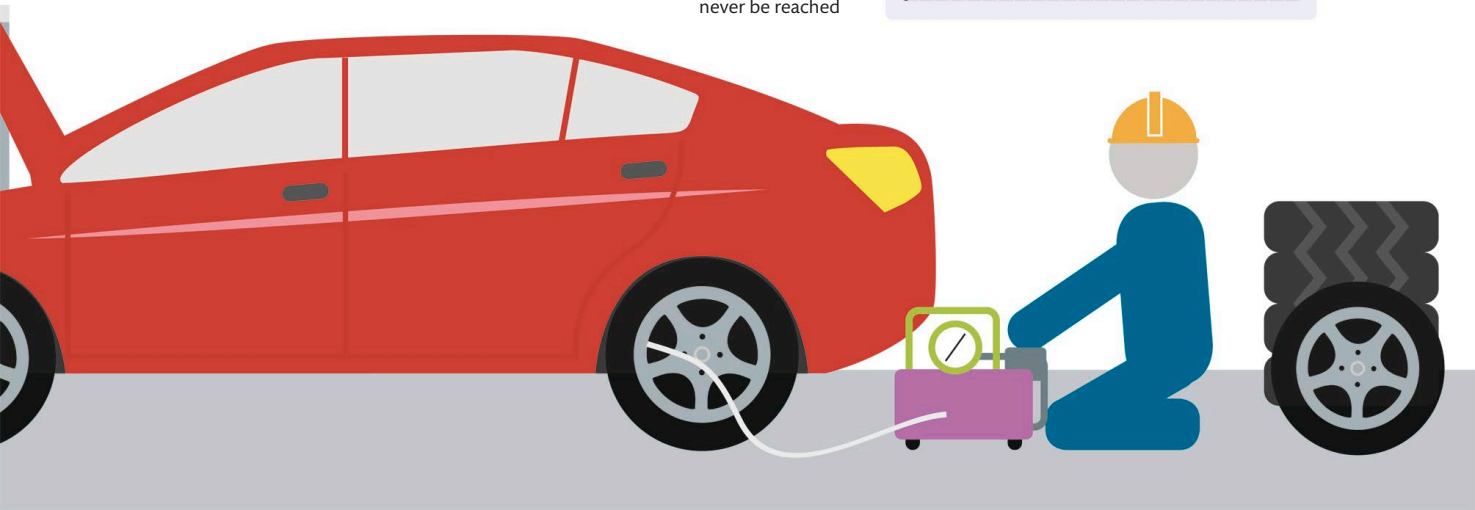
Logic errors are usually the trickiest to spot. The program may run without crashing, but it produces an unexpected result. These errors can be caused by a number of issues, including the use of the wrong variable in an expression.

Infinite results
In this example, an infinite loop prints the word "counting" indefinitely. Because the value in the variable **count** is never updated, the loop condition will never be False.

```
count = 1

while count < 5:
    print("counting")

print("finished")
```

The variable **count** is set to the value **1**

This statement will never be reached



Error messages

While they are the most obvious tools to help programmers with debugging, Python's error messages tend to be slightly cryptic and can appear to add to the mystery of debugging rather than clearing it up. This table lists some of the most common error messages along with their meaning. Programmers tend to become familiar with these errors and their solutions quickly.

ERROR MESSAGES

Error message	Meaning
EOL found while scanning string literal	Closing quotation mark missing for a string on that line
Unsupported operand type(s) for +: 'int' and 'str'	The + operator expects the values on either side of it to be of the same type
Expected an indented block	The body of a loop or conditional is not indented
Unexpected indent	This line is indented too much
Unexpected EOF while parsing	Missing bracket just before the end of the program
Name [name of variable or function] is not defined	Usually caused by misspelling the name of the variable or function

Text coloring

Like most other IDEs (see pp.208–209) and dedicated code editing programs, IDLE colors the text of a Python program. This makes it easier to spot errors. For example, keywords such as "for", "while", and "if" are orange and strings are green (see p.98). A part of the code not appearing in the correct color can be a sign that there is a syntax error. And several lines of code suddenly being colored green is usually the sign of a missing closing quotation mark on a string.

Wrong color
There are four errors in this example. The missing quotation marks and misspelling of the keyword "while" mean that the code will not be colored correctly.

Missing a quotation mark at the start

The keyword "while" has been spelled incorrectly

Misspelling of keyword and missing quotation mark

```
answer = input(Pick a number")
whle answer != 7:
    pritn(Not the right number")
    answer = input("Pick a number")
```

CHECK THE ERROR

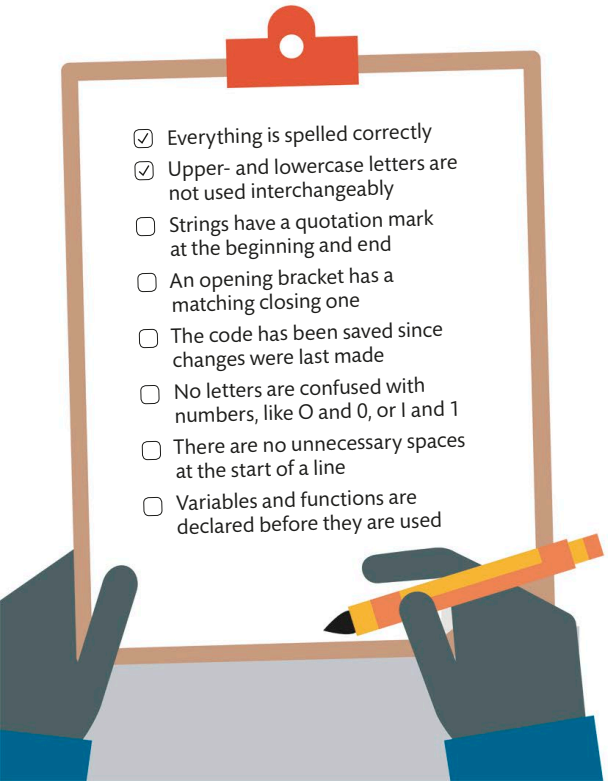
A common issue when debugging is that the actual error may be located just before the place indicated by the error message. It is therefore worth checking for the error earlier in the indicated line or on the line above it.

```
print(hello " + "world")
```

Produces an "Invalid syntax" error due to a missing quote

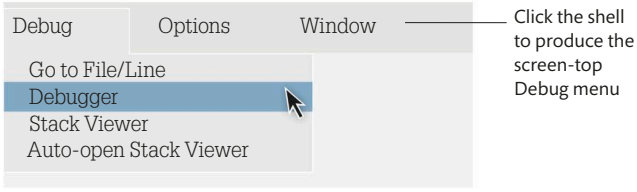
Debugging checklist

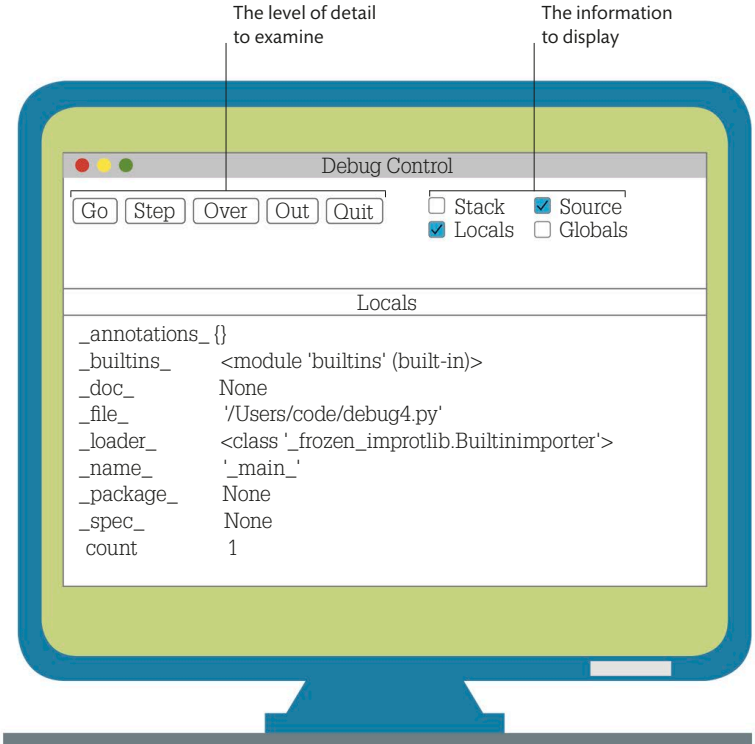
When an error appears but its cause is not immediately clear, there are a few things that can be checked. This might not solve every problem, but many errors are caused by trivial mistakes that can be easily fixed. Here is a list of things to look out for:



Debugger

IDLE also contains a tool called a debugger. This allows programmers to "step through" the execution of their program, running one line at a time. It also shows the contents of variables at each step in the program. The debugger can be started from the shell window, which includes a Debug menu. Selecting Debugger from this menu will start the debugging process the next time a program is run. Choosing it again will turn it off.





IDLE debugger
When a program is run, the debugger will display information about it, including current values of variables. Clicking on the option "Step" will expose the code running behind the scenes, which is normally hidden from programmers.