Python - Flow Control

So, you know the basics of individual instructions and that a program is just a series of instructions. But programming's real strength isn't just running one instruction after another like a weekend errand list. Based on how expressions evaluate, a program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements* can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I'll provide flowchart versions of the code discussed in this chapter. Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.

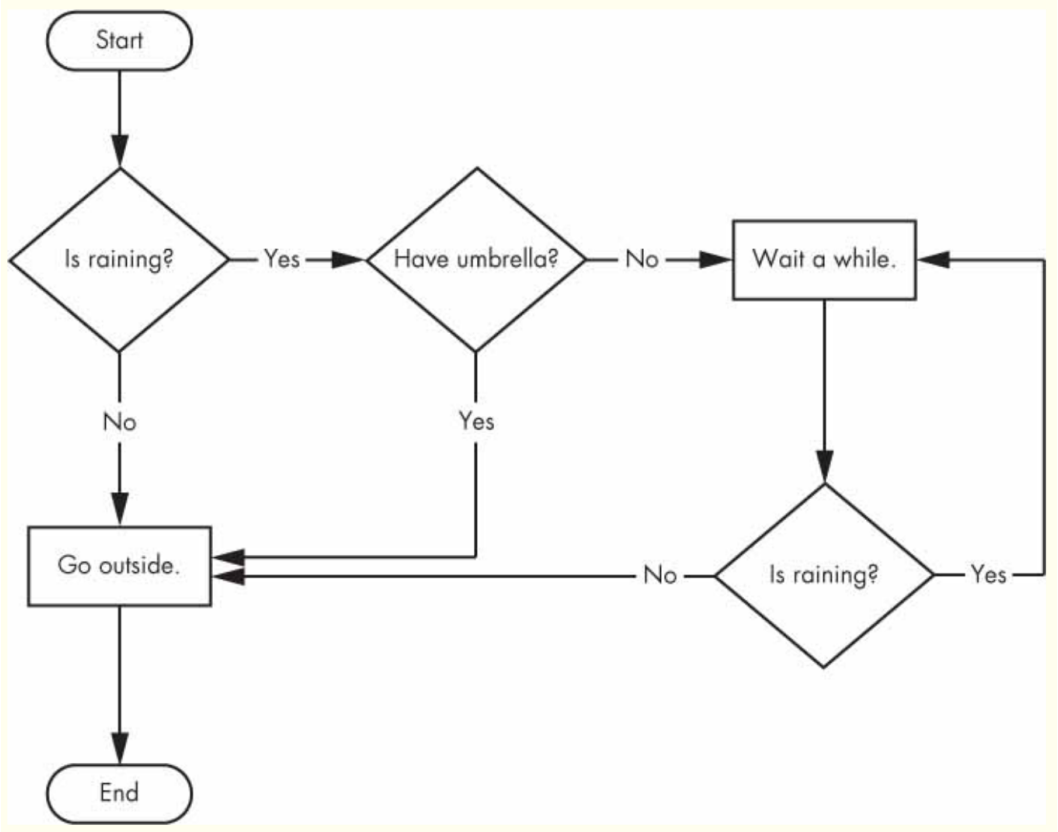


Figure 2-1: A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

Boolean Values

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: True and False. (Boolean is capitalized because the data type is named after mathematician George Boole.) When

entered as Python code, the Boolean values True and False lack the quotes you place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase.

```
>>>spam = True
>>>spam
True
```

```
❷ >>> true
   Traceback (most recent call last):
     File "<pyshell#2>", line 1, in <module>
       true
   NameError: name 'true' is not defined
❸ >>> True = 2 + 2
   SyntaxError: can't assign to keyword
```

Like any other value, Boolean values are used in expressions and can be stored in variables.

1. If you don’t use the proper case
2. or you try to use True and False for variable names, Python will give you an error message.

Comparison Operators

Comparison operators, also called *relational operators*, compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Table 2-1: Comparison Operators

These operators evaluate to `True` or `False` depending on the values you give them. Let's try some operators now, starting with `==` and `!=`.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, `==` (equal to) evaluates to True when the values on both sides are the same, and `!=` (not equal to) evaluates to True when the two values are different. The `==` and `!=` operators can actually work with values of any data type.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
>>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The expression 42 == '42' evaluates to `False`.

THE DIFFERENCE BETWEEN THE == AND = OPERATORS

You might have noticed that the == operator (equal to) has two equal signs, while the = operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- The == operator (equal to) asks whether two values are the same as each other.
- The = operator (assignment) puts the value on the right into the variable on the left.

To help remember which is which, notice that the == operator (equal to) consists of two characters, just like the != operator (not equal to) consists of two characters.

Boolean Operators

The three Boolean operators (`and`, `or`, and `not`) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail, starting with the and operator.

Binary Boolean Operators

The `and` and `or` operators always take two Boolean values (or expressions), so they're considered *binary* operators. The `and` operator evaluates an expression to `True` if *both* Boolean values are True; otherwise, it evaluates to `False`.

```
>>> True and True
True
>>> True and False
False
```

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the and operator.

Expression	Evaluates to . . .
True and True	True
True and False	False
False and True	False
False and False	False

Table 2-2: The and Operator's Truth Table

On the other hand, the `or` operator evaluates an expression to `True` if *either* of the two Boolean values is `True`. If both are `False`, it evaluates to `False`.

The not Operator

Unlike `and` and `or`, the `not` operator operates on only one Boolean value (or expression). This makes it a *unary* operator. The `not` operator simply evaluates to the opposite Boolean value.

```
>>> not True
False
>>> not not not not True
True
```

Mixing Boolean and Comparison Operators

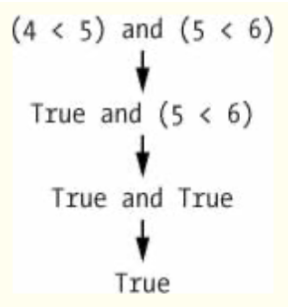
Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the `and`, `or`, and `not` operators are called Boolean operators because they always operate on the Boolean values `True` and `False`. While expressions like `4 < 5` aren't Boolean values, they are expressions that evaluate down to Boolean values.

```
>>> (4 < 5) and (5 < 6)
```

```
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer's evaluation process for (4 < 5) and (5 < 6) as the following:



Elements of Flow Control

Flow control statements often start with a part called the *condition* and are always followed by a block of code called the *clause*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, True or False. A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

- Blocks begin when the indentation increases.
- Blocks can contain other blocks.
- Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
  ❶ print('Hello, Mary')
    if password == 'swordfish':
     ❷ print('Access granted.')
    else:
     ❸ print('Wrong password.')
```

You can view the execution of this program at *https://autbor.com/blocks/*. The first block of code ❶ starts at the line print('Hello, Mary') and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: print('Access Granted.'). The third block ❸ is also one line long: print('Wrong password.').

Flow Control Statements

if Statements

The most common type of flow control statement is the `if` statement.
An `if` statement's clause (that is, the block following the `if` statement) will execute if the statement's condition is `True`. The clause is skipped if the condition is `False`.

In plain English, an `if` statement could be read as, “If this condition is true, execute the code in the clause.” In Python, an if statement consists of the following:

- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `if` clause)

For example, let’s say you have some code that checks to see whether someone’s name is Alice. (Pretend name was assigned some value earlier.)

```
if name == 'Alice':
    print('Hi, Alice.')
```

else Statements

An `if` clause can optionally be followed by an `else` statement. The `else` clause is executed only when the `if` statement’s condition is `False`. In plain English, an `else` statement could be read as, “If this condition is true, execute this code. Or else, execute that code.” An `else` statement doesn’t have a condition, and in code, an `else` statement always consists of the following:

- The `else` keyword
- A colon
- Starting on the next line, an indented block of code (called the `else` clause)

Returning to the Alice example, let’s look at some code that uses an `else` statement to offer a different greeting if the person’s name isn’t Alice.

```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

elif Statements

While only one of the `if` or `else` clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The `elif` statement is an "`else if`" statement that always follows an if or another `elif` statement. It provides another condition that is checked only if all of the previous conditions were `False`. In code, an `elif` statement always consists of the following:

- The `elif` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `elif` clause)

Let's add an elif to the name checker to see this statement in action.

```
name = 'Alice'
age = 10

if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

When there is a chain of `elif` statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be `True`, the rest of the `elif` clauses are automatically skipped.

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

Optionally, you can have an else statement after the last elif statement. In that case, it *is* guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every if and elif statement are False, then the else clause is executed. For example, let's re-create the Alice program to use if, elif, and else clauses.

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

while Loop Statements

You can make a block of code execute over and over again using a `while` statement. The code in a while clause will be executed as long as the `while` statement's condition is `True`. In code, a `while` statement always consists of the following:

- The `while` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `while` clause)

You can see that a `while` statement looks similar to an `if` statement. The difference is in how they behave. At the end of an `if` clause, the program execution continues after the `if` statement. But at the end of a `while` clause, the program execution jumps back to the start of the `while` statement. The `while` clause is often called the *while loop* or just the *loop*.

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a `while` statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

These statements are similar—both `if` and `while` check the value of `spam`, and if it's less than 5, they print a message. But when you run these two code snippets, something very different happens for each one. For the `if` statement, the output is simply `"Hello, world."`. But for the while statement, it's `"Hello, world."` repeated five times! Take a look at the flowcharts for these two pieces of code, Figures 2-8 and 2-9, to see why this happens.

An Annoying while Loop

Here's a small example program that will keep asking you to type, literally, your name.

```
❶ name = ''
❷ while name != 'your name':
       print('Please type your name.')
   ❸ name = input()
❹ print('Thank you!')
```

First, the program sets the name variable ❶ to an empty string. This is so that the name != 'your name' condition will evaluate to True and the program execution will enter the while loop's clause ❷.

The code inside this clause asks the user to type their name, which is assigned to the name variable ❸. Since this is the last line of the block, the execution moves back to

the start of the while loop and reevaluates the condition. If the value in name is *not equal* to the string 'your name', then the condition is True, and the execution enters the while clause again.

But once the user types your name, the condition of the while loop will be 'your name' != 'your name', which evaluates to `False`. The condition is now False, and instead of the program execution reentering the while loop's clause, Python skips past it and continues running the rest of the program ❹.

```
Please type your name.
Al
Please type your name.
Albert
Please type your name.
%#@#%*(^&!!!
Please type your name.
your name
Thank you!
```

If you never enter your name, then the while loop's condition will never be False, and the program will just keep asking forever. Here, the `input()` call lets the user enter the right string to make the program move on.

break Statements

There is a shortcut to getting the program execution to break out of a `while` loop's clause early. If the execution reaches a break statement, it immediately exits the while loop's clause. In code, a break statement simply contains the break keyword.

```
❶ while True:
      print('Please type your name.')
```

```
    ❷ name = input()
    ❸ if name == 'your name':
        ❹ break
❺ print('Thank you!')
```

The first line ❶ creates an *infinite loop*; it is a `while` loop whose condition is always `True`. (The expression `True`, after all, always evaluates down to the value `True`.) After the program execution enters this loop, it will exit the loop only when a `break` statement is executed. (An infinite loop that *never* exits is a common programming bug.)

continue Statements

Like `break` statements, `continue` statements are used inside loops. When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

> **TRAPPED IN AN INFINITE LOOP?**
> If you ever run a program that has a bug causing it to get stuck in an infinite loop, press CTRL-C or select **Shell ▸ Restart Shell** from IDLE's menu. This will send a KeyboardInterrupt error to your program and cause it to stop immediately.

```
while True:
  print('Who are you?')
  name = input()
  if name != 'Joe':
    continue

# if the user enters any name besides Joe the continue statement
# jumps back to the loop and reevaluate the condition

  print('Hello, Joe. What is the passsword?(It is a fish.)')
  password = input()
  if password == 'cuttlefish':
      break
print('Access Granted!')
```

Run this program and give it some input. Until you claim to be Joe, the program shouldn't ask for a password, and once you enter the correct password, it should exit.

```
Who are you?
I'm fine, thanks. Who are you?
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.
```

Execution of this program.

for Loops and the range() Function

The `while` loop keeps looping while its condition is `True` (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a for loop statement and the `range()` function.

In code, a for statement looks something like `for i in range(5):` and includes the following:

- The `for` keyword
- A variable name
- The `in` keyword
- A call to the `range()` method with up to three integers passed to it
- A colon

- Starting on the next line, an indented block of code (called the for clause)

```
print('My name is')
for i in range(5):
  print('Jimmy Five Times (' + str(i) + ')')
# The loop will print this string five times and exit
```

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

Note: You can use `break` and `continue` even inside `for` loops as well. *The continue statement will continue to the next value of the for loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use continue and break statements only inside while and for loops. If you try to use these statements elsewhere, Python will give you an error.*

As another for loop example, consider this story about the mathematician **Carl Friedrich Gauss**. When Gauss was a boy, a teacher wanted to give the class some busywork. The teacher told them to add up all the numbers from 0 to 100. Young Gauss came up with a clever trick to figure out the answer in a few seconds, but you can write a Python program with a for loop to do this calculation for you.

```
total = 0
for num in range(101):
  total = total + num
print(total)
```

The result should be **5,050**.

(Young Gauss figured out a way to solve the problem in seconds. There are 50 pairs of numbers that add up to 101: 1 + 100, 2 + 99, 3 + 98, and so on, until 50 + 51. Since 50 × 101 is 5,050, the sum of all the numbers from 0 to 100 is 5,050. Clever kid!)

The Starting, Stopping, and Stepping Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero.

```
for i in range(12,16):
  print(i)
```

```
12
13
14
15
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the *step argument*. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0,10,2):
  print(i)
```

So calling `range(0, 10, 2)` will count from zero to eight by intervals of two.

```
0
2
4
6
8
```

The `range()` function is flexible in the sequence of numbers it produces for `for` loops. *For* example (I never apologize for my puns), you can even use a negative number for the step argument to make the `for` loop count down instead of up.

```
for i in range(5, -1, -1):
    print(i)
```

This `for` loop would have the following output:

```
5
4
3
2
1
0
```

Modules, Packages and Libraries

If you have some experience with Python, you've likely used modules. For example, you may have used the:

- **random** module to generate pseudo-random number generators for various distributions.
- **html** module to parse HTML pages.

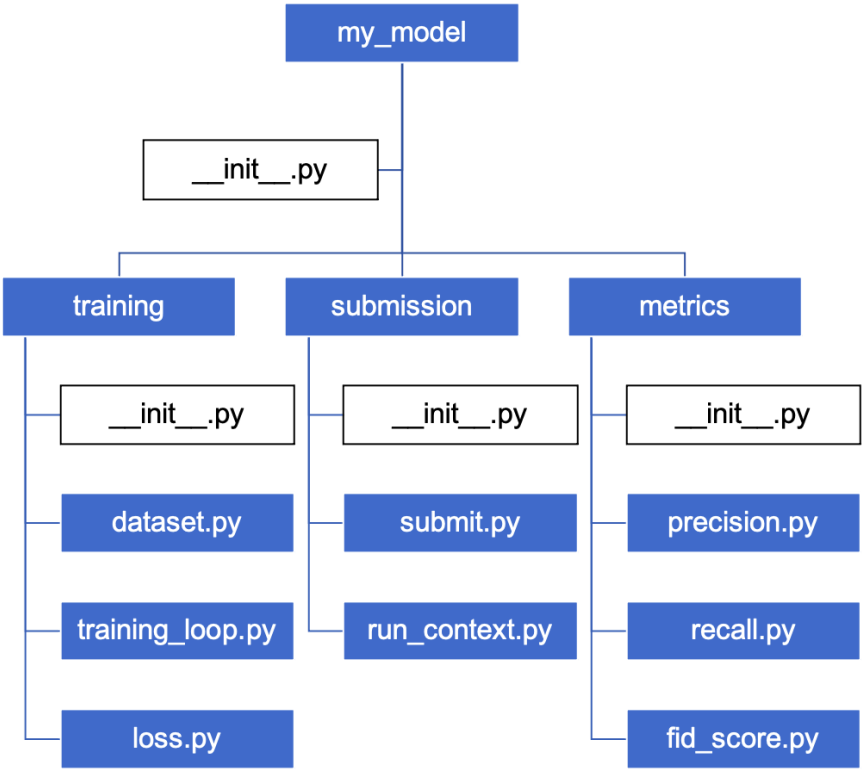
- **datetime** module to manipulate date and time data.
- **re** module to detect and parse regular expressions in Python.

Python Packages

When developing a large application, you may end up with many different modules that are difficult to manage. In such a case, you'll benefit from grouping and organizing your modules. That's when packages come into play.

Python packages are basically a directory of a collection of modules. Packages allow the hierarchical structure of the module namespace. Just like we organize our files on a hard drive into folders and sub-folders, we can organize our modules into packages and subpackages.

To be considered a package (or subpackage), a directory must contain a file named `__init__.py`. This file usually includes the initialization code for the corresponding package.



We can import specific modules from this package using the dot notation. For example, to import the `dataset` module from the above package, we can use one of the following code snippets:

```
import my_model.training.dataset
```

OR

```
from my_model.training import dataset
```

There are a lot of built-in and open-source Python packages that you are probably already using. For example:

- **NumPy** is the fundamental Python package for scientific computing.
- **pandas** is a Python package for fast and efficient processing of tabular data, time series, matrix data, etc.
- **pytest** provides a variety of modules to test new code, including small unit tests or complex functional tests.

As your application grows larger and uses many different modules, Python packages become a crucial component for optimizing your code structure.

Python Libraries

A library is an umbrella term referring to a reusable chunk of code. Usually, a Python library contains a collection of related modules and packages. Actually, this term is often used interchangeably with “Python package” because packages can also contain modules and other packages (subpackages). However, it is often assumed that while **a package is a collection of modules, a library is a collection of packages**.

Oftentimes, developers create Python libraries to share reusable code with the community. To eliminate the need for writing code from scratch, they create a set of useful functions related to the same area.

There are thousands of useful libraries available today. I’ll give just a few examples:

- **Matplotlib** library is a standard library for generating data visualizations in Python. It supports building basic two-dimensional graphs as well as more complex animated and interactive visualizations.

- **PyTorch** is an open-source deep-learning library built by Facebook's AI Research lab to implement advanced neural networks and cutting-edge research ideas in industry and academia.
- **pygame** provides developers with tons of convenient features and tools to make game development a more intuitive task.
- **Beautiful Soup** is a very popular Python library for getting data from the web. The modules and packages inside this library help extract useful information from HTML and XML files.
- **Requests** is a part of a large collection of libraries designed to make Python HTTP requests simpler. The library offers an intuitive JSON method that helps you avoid manually adding query strings to your URLs.
- **missingno** is very handy for handling missing data points. It provides informative visualizations about the missing values in a dataframe, helping data scientists to spot areas with missing data. It is just one of the many great Python libraries for data cleaning.

By the way, the **NumPy** and **pandas** packages that were mentioned before are also often referred to as libraries. That is because these are complex packages that have wide applications (i.e. scientific computing and data manipulation, respectively). They also include multiple subpackages and so basically satisfy the definition of a Python library.

Importing Modules

All Python programs can call a basic set of functions called *built-in functions*, including the `print()`, `input()`, and `len()` functions you've seen before. Python also comes with a set of modules called the *standard library*. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the `math` module has mathematics-related functions, the `random` module has random number-related functions, and so on.

Before you can use the functions in a module, you must import the module with an `import` statement. In code, an import statement consists of the following:

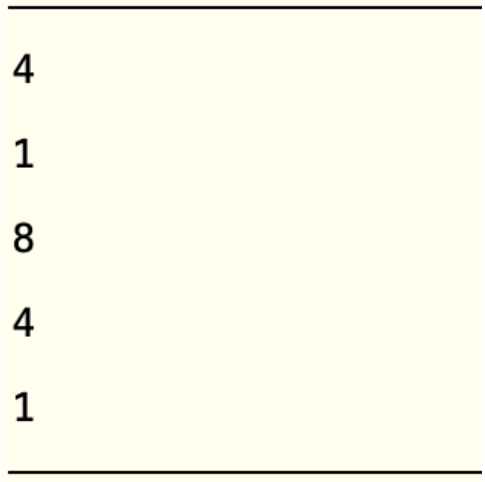
- The `import` keyword

- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you `import` a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

```
import random

for in range(5):
  print(random.randin t(1,10))
```



Output

DON'T OVERWRITE MODULE NAMES

When you save your Python scripts, take care not to give them a name that is used by one of Python's modules, such as *random.py*, *sys.py*, *os.py*, or *math.py*. If you accidentally name one of your programs, say, *random.py*, and use an import random statement in another program, your program would import your *random.py* file instead of Python's random module. This can lead to errors such as `AttributeError: module 'random' has no attribute 'randint'`, since your *random.py* doesn't have the functions that the real random module has. Don't use the names of any built-in Python functions either, such as `print()` or `input()`.

Here's an example of an import statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules.

from import Statements

An alternative form of the `import` statement is composed of the `from` keyword, followed by the module name, the `import` keyword, and a star; for example, `from random import *`.

With this form of import statement, calls to functions in `random` will not need the `random`. prefix. However, using the full name makes for more readable code, so it is better to use the `import random` form of the statement.

A Short Program: Rock, Paper, Scissors

Let's use the programming concepts we've learned so far to create a simple rock, paper, scissors game. The output will look like this:

```
ROCK, PAPER, SCISSORS
0 Wins, 0 Losses, 0 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
p
PAPER versus...
PAPER
It is a tie!
0 Wins, 1 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
s
SCISSORS versus...
PAPER
You win!
1 Wins, 1 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
q
```

Source Code:

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of wins, losses, and ties.
wins = 0
losses = 0
ties = 0

while True: # The main game loop.
    print('%s Wins, %s Losses, %s Ties' % (wins, losses, ties))
    while True: # The player input loop.
        print('Enter your move: (r)ock (p)aper (s)cissors or (q)uit')
        playerMove = input()
        if playerMove == 'q':
            sys.exit() # Quit the program.
        if playerMove == 'r' or playerMove == 'p' or playerMove == 's':
```

```
            break # Break out of the player input loop.
        print('Type one of r, p, s, or q.')

    # Display what the player chose:
    if playerMove == 'r':
        print('ROCK versus...')
    elif playerMove == 'p':
        print('PAPER versus...')
    elif playerMove == 's':
        print('SCISSORS versus...')

    # Display what the computer chose:
    randomNumber = random.randint(1, 3)
    if randomNumber == 1:
        computerMove = 'r'
        print('ROCK')
    elif randomNumber == 2:
        computerMove = 'p'
        print('PAPER')
    elif randomNumber == 3:
        computerMove = 's'
        print('SCISSORS')

    # Display and record the win/loss/tie:
    if playerMove == computerMove:
        print('It is a tie!')
        ties = ties + 1
    elif playerMove == 'r' and computerMove == 's':
        print('You win!')
        wins = wins + 1
    elif playerMove == 'p' and computerMove == 'r':
        print('You win!')
        wins = wins + 1
    elif playerMove == 's' and computerMove == 'p':
        print('You win!')
        wins = wins + 1
    elif playerMove == 'r' and computerMove == 'p':
        print('You lose!')
        losses = losses + 1
    elif playerMove == 'p' and computerMove == 's':
        print('You lose!')
        losses = losses + 1
    elif playerMove == 's' and computerMove == 'r':
        print('You lose!')
        losses = losses + 1
```

Summary

By using expressions that evaluate to `True` or `False` (also called conditions), you can write programs that make decisions on what code to execute and what code to skip.

You can also execute code over and over again in a loop while a certain condition evaluates to `True`. The break and continue statements are useful if you need to exit a loop or jump back to the loop's start.

These flow control statements will let you write more intelligent programs. You can also use another type of flow control by writing your own functions, which is the topic of the next chapter.

📝Practice Questions

1. What are the two values of the Boolean data type? How do you write them?

2. What are the three Boolean operators?

3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).

4. What do the following expressions evaluate to?

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3 == 5)
not ((5 > 4) or (3 == 5))
(True and True) and (True == False)
(not False) or (not True)
```

5. What are the six comparison operators?

6. What is the difference between the equal to operator and the assignment operator?

7. Explain what a condition is and where you would use one.

8. Identify the three block in this code

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

9. Write code that prints `Hello` if `1` is stored in `spam`, prints `Howdy` if `2` is stored in `spam`, and prints `Greetings!` if anything else is stored in `spam`.
10. What is the difference between break and continue statements?