Control Structures

Booleans

Another type in Python is the Boolean type. There are two Boolean values: **True** and **False**. They can be created by comparing values, for instance by using the equal operator ==.

```
>>> my_boolean = True
>>> my_boolean
True

>>> 2 == 3
False
>>> "hello" == "hello"
True
```

Try It Yourself

Be careful not to confuse **assignment** (one equals sign) with **comparison** (two equals signs).

Comparison

Another comparison operator, the **not equal** operator (!=), evaluates to **True** if the items being compared aren't equal, and **False** if they are.

```
>>> 1 != 1
False
>>> "eleven" != "seven"
True
>>> 2 != 10
True
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Comparison

Python also has operators that determine whether one number (float or integer) is greater than or smaller than another. These operators are > and < respectively.

```
>>> 7 > 5
True
>>> 10 < 10
False
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Comparison

The greater than or equal to, and smaller than or equal to operators are >= and <=.
They are the same as the strict greater than and smaller than operators, except that they return **True** when comparing equal numbers.

```
>>> 7 <= 8
True
>>> 9 >= 9.0
True
```

Try It Yourself

Greater than and smaller than operators can also be used to compare strings **lexicographically** (the alphabetical order of words is based on the alphabetical order of their component letters).

---

if Statements

You can use **if** statements to run code if a certain condition holds.
If an expression evaluates to **True**, some statements are carried out. Otherwise, they aren't carried out.
An if statement looks like this:

```
if expression:
  statements
```

Python uses **indentation** (white space at the beginning of a line) to delimit blocks of code. Other languages, such as C, use curly braces to accomplish this, but in Python indentation is mandatory; programs won't work without it. As you can see, the statements in the **if** should be indented.

---

if Statements

Here is an example **if** statement:

```
if 10 > 5:
  print("10 greater than 5")

print("Program ended")
```

Try It Yourself

The expression determines whether 10 is greater than five. Since it is, the indented statement runs, and "10 greater than 5" is output. Then, the unindented statement, which is not part of the **if** statement, is run, and "Program ended" is displayed.

Result:

```
>>>
10 greater than 5
Program ended
>>>
```

Notice the **colon** at the end of the expression in the **if** statement.

> As the program contains multiple lines of code, you should create it as a separate file and run it.

---

if Statements

To perform more complex checks, **if** statements can be nested, one inside the other.
This means that the inner **if** statement is the statement part of the outer one. This is one way to see whether multiple conditions are satisfied.

For example:

```
num = 12
if num > 5:
  print("Bigger than 5")
  if num <=47:
    print("Between 5 and 47")
```

Try It Yourself

Result:

```
>>>
Bigger than 5
Between 5 and 47
>>>
```

> Tap **Try It Yourself** to play around with the code!

---

else Statements

An **else** statement follows an **if** statement, and contains code that is called when the if statement evaluates to **False**.
As with **if** statements, the code inside the block should be indented.

```
x = 4
if x == 5:
  print("Yes")
else:
  print("No")
```

Try It Yourself

Result:

```
>>>
No
>>>
```

Tap **Try It Yourself** to play around with the code!

---

else Statements

You can chain **if** and **else** statements to determine which option in a series of possibilities is true.
For example:

```
num = 7
if num == 5:
  print("Number is 5")
else:
  if num == 11:
    print("Number is 11")
  else:
    if num == 7:
      print("Number is 7")
    else:
      print("Number isn't 5, 11 or 7")
```

Try It Yourself

Result:

```
>>>
Number is 7
>>>
```

Tap **Try It Yourself** to play around with the code!

---

elif Statements

The **elif** (short for else if) statement is a shortcut to use when chaining **if** and **else** statements.
A series of **if** **elif** statements can have a final **else** block, which is called if none of the **if** or **elif** expressions is True.
For example:

```
num = 7
if num == 5:
  print("Number is 5")
elif num == 11:
  print("Number is 11")
elif num == 7:
  print("Number is 7")
else:
  print("Number isn't 5, 11 or 7")
```

Try It Yourself

Result:

```
>>>
Number is 7
>>>
```

In other programming languages, equivalents to the **elif** statement have varying names, including **else if**, **elseif** or **elsif**.

Boolean Logic

Boolean logic is used to make more complicated conditions for **if** statements that rely on more than one condition.
Python's Boolean operators are **and**, **or**, and **not**.
The **and** operator takes two arguments, and evaluates as **True** if, and only if, both of its arguments are **True**. Otherwise, it evaluates to **False**.

```
>>> 1 == 1 and 2 == 2
True
>>> 1 == 1 and 2 == 3
False
>>> 1 != 1 and 2 == 2
False
>>> 2 < 1 and 3 > 6
False
```

Try It Yourself

Python uses words for its Boolean operators, whereas most other languages use symbols such as &&, || and !.

Boolean Or

The **or** operator also takes two arguments. It evaluates to **True** if either (or both) of its arguments are **True**, and **False** if both arguments are **False**.

```
>>> 1 == 1 or 2 == 2
True
>>> 1 == 1 or 2 == 3
True
>>> 1 != 1 or 2 == 2
True
>>> 2 < 1 or 3 > 6
False
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Boolean Not

Unlike other operators we've seen so far, **not** only takes one argument, and inverts it.
The result of **not True** is **False**, and **not False** goes to **True**.

```
>>> not 1 == 1
False
>>> not 1 > 7
True
```

Try It Yourself

You can chain multiple conditional statements in an **if** statement using the Boolean operators.

Operator Precedence

Operator precedence is a very important concept in programming. It is an extension of the mathematical idea of order of operations (multiplication being performed before addition, etc.) to include other operators, such as those in Boolean logic.

The below code shows that == has a higher precedence than **or**:

```
>>> False == False or True
True
>>> False == (False or True)
False
>>> (False == False) or True
True
```

Try It Yourself

Python's order of operations is the same as that of normal mathematics: parentheses first, then exponentiation, then multiplication/division, and then addition/subtraction.

Operator Precedence

The following table lists all of Python's operators, from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~, +, -	Complement, unary plus and minus (method names for the last two are +@ and -@)
*, /, %, //	Multiply, divide, modulo and floor division
+, -	Addition and subtraction

>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR'
|	Bitwise 'OR'
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparison operators, equality operators, membership and identity operators
not	Boolean 'NOT'
and	Boolean 'AND'
or	Boolean 'OR'
=, %=, /=, //=, -=, +=, *=, **=	Assignment operators

> Operators in the same box have the same precedence.

---

while Loops

An **if** statement is run once if its condition evaluates to **True**, and never if it evaluates to **False**. A **while** statement is similar, except that it can be run more than once. The statements inside it are repeatedly executed, as long as the condition holds. Once it evaluates to **False**, the next section of code is executed.
Below is a **while** loop containing a variable that counts up from 1 to 5, at which point the loop terminates.

```
i = 1
while i <=5:
  print(i)
  i = i + 1

print("Finished!")
```

Try It Yourself

Result:

```
>>>
1
2
3
4
5
Finished!
>>>
```

> The code in the body of a **while** loop is executed repeatedly. This is called **iteration**.

---

while Loops

The **infinite loop** is a special kind of while loop; it never stops running. Its condition always remains **True**.
An example of an infinite loop:

```
while 1==1:
  print("In the loop")
```

Try It Yourself

This program would indefinitely print "In the loop".

> You can stop the program's execution by using the Ctrl-C shortcut or by closing the program.

break

To end a **while** loop prematurely, the **break** statement can be used.
When encountered inside a loop, the **break** statement causes the loop to finish immediately.

```
i = 0
while 1==1:
  print(i)
  i = i + 1
  if i >= 5:
    print("Breaking")
    break

print("Finished")
```

Try It Yourself

Result:

```
>>>
0
1
2
3
4
Breaking
Finished
>>>
```

> Using the **break** statement outside of a loop causes an error.

continue

Another statement that can be used within loops is **continue**.
Unlike **break**, **continue** jumps back to the top of the loop, rather than stopping it.

```
i = 0
while True:
  i = i +1
  if i == 2:
    print("Skipping 2")
    continue
  if i == 5:
    print("Breaking")
    break
  print(i)

print("Finished")
```

Try It Yourself

Result:

```
>>>
1
Skipping 2
3
4
Breaking
Finished
>>>
```

Basically, the **continue** statement stops the current iteration and continues with the next one.

> Using the **continue** statement outside of a loop causes an error.

Lists

Lists are another type of object in Python. They are used to store an indexed list of items.
A list is created using **square brackets** with **commas** separating items.
The certain item in the list can be accessed by using its index in square brackets.
For example:

```
words = ["Hello", "world", "!"]
print(words[0])
print(words[1])
print(words[2])
```

Try It Yourself

Result:

```
>>>
Hello
world
!
>>>
```

> The first list item's index is **0**, rather than 1, as might be expected.

Lists

An empty list is created with an empty pair of square brackets.

```
empty_list = []
print(empty_list)
```

Try It Yourself

Result:

```
>>>
[]
>>>
```

Most of the time, a comma won't follow the last item in a list. However, it is perfectly valid to place one there, and it is encouraged in some cases.

Lists

Typically, a list will contain items of a single item type, but it is also possible to include several different types.
Lists can also be nested within other lists.

```
number = 3
things = ["string", 0, [1, 2, number], 4.56]
print(things[1])
print(things[2])
print(things[2][2])
```

Try It Yourself

Result:

```
>>>
0
[1, 2, 3]
3
>>>
```

Lists of lists are often used to represent 2D grids, as Python lacks the multidimensional arrays that would be used for this in other languages.

Lists

Indexing out of the bounds of possible list values causes an IndexError.
Some types, such as **strings**, can be indexed like lists. Indexing **strings** behaves as though you are indexing a list containing each character in the string.
For other types, such as integers, indexing them isn't possible, and it causes a TypeError.

```
str = "Hello world!"
print(str[6])
```

Try It Yourself

Result:

```
>>>
w
>>>
```

Tap **Try It Yourself** to play around with the code!

List Operations

The item at a certain index in a list can be reassigned.
For example:

```
nums = [7, 7, 7, 7, 7]
nums[2] = 5
print(nums)
```

Try It Yourself

Result:

```
>>>
[7, 7, 5, 7, 7]
>>>
```

Tap **Try It Yourself** to play around with the code!

List Operations

Lists can be added and multiplied in the same way as strings.
For example:

```
nums = [1, 2, 3]
print(nums + [4, 5, 6])
print(nums * 3)
```

Try It Yourself

Result:

```
>>>
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
```

Lists and strings are similar in many ways - strings can be thought of as lists of characters that can't be changed.

List Operations

To check if an item is in a list, the **in** operator can be used. It returns **True** if the item occurs one or more times in the list, and **False** if it doesn't.

```
words = ["spam", "egg", "spam", "sausage"]
print("spam" in words)
print("egg" in words)
print("tomato" in words)
```

Try It Yourself

Result:

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

The **in** operator is also used to determine whether or not a string is a substring of another string.

List Operations

To check if an item is not in a list, you can use the **not** operator in one of the following ways:

```
nums = [1, 2, 3]
print(not 4 in nums)
print(4 not in nums)
print(not 3 in nums)
print(3 not in nums)
```

Try It Yourself

Result:

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

Tap **Try It Yourself** to play around with the code!

List Functions

Another way of altering lists is using the **append** method. This adds an item to the end of an existing list.

```
nums = [1, 2, 3]
nums.append(4)
print(nums)
```

Try It Yourself

Result:

```
>>>
[1, 2, 3, 4]
>>>
```

The **dot** before append is there because it is a **method** of the list class. Methods will be explained in a later lesson.

List Functions

To get the number of items in a list, you can use the **len** function.

```
nums = [1, 3, 5, 2, 4]
print(len(nums))
```

Try It Yourself

Result:

```
>>>
5
>>>
```

Unlike **append**, **len** is a normal function, rather than a method. This means it is written before the list it is being called on, without a dot.

List Functions

The **insert** method is similar to **append**, except that it allows you to insert a new item at any position in the list, as opposed to just at the end.

```
words = ["Python", "fun"]
index = 1
words.insert(index, "is")
print(words)
```

Try It Yourself

Result:

```
>>>
['Python', 'is', 'fun']
>>>
```

Tap **Try It Yourself** to play around with the code!

List Functions

The **index** method finds the first occurrence of a list item and returns its index.
If the item isn't in the list, it raises a ValueError.

```
letters = ['p', 'q', 'r', 's', 'p', 'u']
print(letters.index('r'))
print(letters.index('p'))
print(letters.index('z'))
```

Try It Yourself

Result:

```
>>>
2
0
ValueError: 'z' is not in list
>>>
```

There are a few more useful functions and methods for lists.
max(list): Returns the list item with the maximum value
min(list): Returns the list item with minimum value
list.**count**(obj): Returns a count of how many times an item occurs in a list
list.**remove**(obj): Removes an object from a list
list.**reverse**(): Reverses objects in a list

Range

The **range** function creates a sequential list of numbers.
The code below generates a list containing all of the integers, up to 10.

```
numbers = list(range(10))
print(numbers)
```

Try It Yourself

Result:

```
>>>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

The call to **list** is necessary because **range** by itself creates a **range object**, and this must be converted to a **list** if you want to use it as one.

Range

If **range** is called with one argument, it produces an object with values from 0 to that argument.
If it is called with two arguments, it produces values from the first to the second.
For example:

```
numbers = list(range(3, 8))
print(numbers)

print(range(20) == range(0, 20))
```

Try It Yourself

Result:

```
>>>
[3, 4, 5, 6, 7]

True
>>>
```

Tap **Try It Yourself** to play around with the code!

Range

range can have a third argument, which determines the interval of the sequence produced. This third argument must be an integer.

```
numbers = list(range(5, 20, 2))
print(numbers)
```

Try It Yourself

Result:

```
>>>
[5, 7, 9, 11, 13, 15, 17, 19]
>>>
```

Tap **Try It Yourself** to play around with the code!

Loops

Sometimes, you need to perform code on each item in a list. This is called iteration, and it can be accomplished with a **while** loop and a counter variable.
For example:

```
words = ["hello", "world", "spam", "eggs"]
counter = 0
max_index = len(words) - 1

while counter <= max_index:
  word = words[counter]
  print(word + "!")
  counter = counter + 1
```

Try It Yourself

Result:

```
>>>
hello!
world!
spam!
eggs!
>>>
```

The example above iterates through all items in the list, accesses them using their indices, and prints them with exclamation marks.

---

for Loop

Iterating through a list using a **while** loop requires quite a lot of code, so Python provides the **for** loop as a shortcut that accomplishes the same thing.
The same code from the previous example can be written with a **for** loop, as follows:

```
words = ["hello", "world", "spam", "eggs"]
for word in words:
  print(word + "!")
```

Try It Yourself

Result:

```
>>>
hello!
world!
spam!
eggs!
>>>
```

The **for** loop in Python is like the **foreach** loop in other languages.

---

for Loops

The **for** loop is commonly used to repeat some code a certain number of times. This is done by combining for loops with **range** objects.

```
for i in range(5):
  print("hello!")
```

Try It Yourself

Result:

```
>>>
hello!
hello!
hello!
hello!
hello!
>>>
```

You don't need to call **list** on the **range** object when it is used in a **for** loop, because it isn't being indexed, so a list isn't required.

Creating a Calculator

This lesson is about an example Python project: a simple calculator.
Each part explains a different section of the program.
The first section is the overall menu. This keeps on accepting user input until the user enters "quit", so a **while** loop is used.

```
while True:
  print("Options:")
  print("Enter 'add' to add two numbers")
  print("Enter 'subtract' to subtract two numbers")
  print("Enter 'multiply' to multiply two numbers")
  print("Enter 'divide' to divide two numbers")
  print("Enter 'quit' to end the program")
  user_input = input(": ")

  if user_input == "quit":
    break
  elif user_input == "add":
    ...
  elif user_input == "subtract":
    ...
  elif user_input == "multiply":
    ...
  elif user_input == "divide":
    ...
  else:
    print("Unknown input")
```

The code above is the starting point for our program. It accepts user input, and compares it to the options in the **if**/**elif** statements.
The **break** statement is used to stop the while loop, in case the user inputs "quit".

Creating a Calculator

The next part of the program is getting the numbers the user wants to do something with.
The code below shows this for the addition section of the calculator. Similar code would have to be written for the other sections.

```
elif user_input == "add":
  num1 = float(input("Enter a number: "))
  num2 = float(input("Enter another number: "))
```

Now, when the user inputs "add", the program prompts to enter two numbers, and stores them in the corresponding variables.

> As it is, this code crashes if the user enters a non-numeric input when prompted to enter a number. We will look at fixing problems like this in a later module.

Creating a Calculator

The final part of the program processes user input and displays it.
The code for the addition part is shown here.

```
elif user_input == "add":
  num1 = float(input("Enter a number: "))
  num2 = float(input("Enter another number: "))
  result = str(num1 + num2)
  print("The answer is " + result)
```

We now have a working program that prompts for user input, and then calculates and prints the sum of the input.

> Similar code would have to be written for the other branches (for subtraction, multiplication and division).
> The output line could be put outside the **if** statements to omit repetition of code.

End.