

>>> Basic Concepts

Welcome to Python!

Python is a high-level programming language, with applications in numerous areas, including web programming, scripting, scientific computing, and artificial intelligence.

It is very popular and used by organizations such as Google, NASA, the CIA, and Disney.

Python is processed at runtime by the interpreter. There is no need to compile your program before executing it.

Welcome to Python!

The three major versions of Python are 1.x, 2.x and 3.x. These are subdivided into minor versions, such as 2.7 and 3.3.

Code written for Python 3.x is guaranteed to work in all future versions.

Both Python Version 2.x and 3.x are used currently.

This course covers **Python 3.x**, but it isn't hard to change from one version to another.

Python has several different implementations, written in various languages.

The version used in this course, **CPython**, is the most popular by far.

An interpreter is a program that runs scripts written in an interpreted language such as Python.

Your First Program

Let's start off by creating a short program that displays "Hello world!".

In Python, we use the **print** statement to output text:

```
>>> print('Hello world!')
Hello world!
```

Try It Yourself

Congratulations! You have written your first program.

Run, save, and share your Python code on our **Code Playground** without installing any additional software.

When using a computer, you will need to download and install Python from www.python.org.

Note the >>> in the code above. They are the prompt symbol of the Python console.

Python is an interpreted language, which means that each line is executed as it is entered. Python also includes **IDLE**, the integrated development environment, which includes tools for writing and debugging entire programs.

Printing Text

The **print** statement can also be used to output multiple lines of text.
For Example:

```
>>> print("Hello world!")
Hello world!
>>> print('Hello world!')
Hello world!
>>> print("Spam and eggs...")
Spam and eggs...
```

Try It Yourself

Python code often contains references to the comedy group **Monty Python**. This is why the words, "spam" and "eggs" are often used as placeholder variables in Python where "foo" and "bar" would be used in other programming languages.

Simple Operations

Python has the capability of carrying out calculations.
Enter a calculation directly into the Python console, and it will output the answer.

```
>>> 2 + 2
4
>>> 5 + 4 - 3
6
```

The spaces around the plus and minus signs here are **optional** (the code would work without them), but they make it easier to read.

Simple Operations

Python also carries out multiplication and division, using an **asterisk** to indicate multiplication and a **forward slash** to indicate division.

Use **parentheses** to determine which operations are performed first.

```
>>> 2 * (3 + 4)
14
>>> 10 / 2
5.0
```

Using a single slash to divide numbers produces a decimal (or **float**, as it is called in programming). We'll have more about **floats** in a later lesson.

Simple Operations

The minus sign indicates a **negative** number.
Operations are performed on negative numbers, just as they are on positive ones.

```
>>> -7
-7
>>> (-7 + 2) * (-4)
20
```

The plus signs can also be put in front of numbers, but this has no effect, and is mostly used to emphasize that a number is positive to increase readability of code.

Simple Operations

Dividing by zero in Python produces an **error**, as no answer can be calculated.

```
>>> 11 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

In Python, the last line of an error message indicates the error's type.
Read error messages carefully, as they often tell you how to fix a program!

Floats

Floats are used in Python to represent numbers that aren't integers.
Some examples of numbers that are represented as floats are 0.5 and -7.8237591.
They can be created directly by entering a number with a decimal point, or by using operations such as division on integers. Extra zeros at the number's end are ignored.

```
>>> 3/4
0.75
>>> 9.8765000
9.8765
```

Computers can't store floats perfectly accurately, in the same way that we can't write down the complete decimal expansion of 1/3 (0.3333333333333333...). Keep this in mind, because it often leads to infuriating bugs!

Floats

As you saw previously, dividing any two integers produces a **float**.
A **float** is also produced by running an operation on two floats, or on a **float** and an **integer**.

```
>>> 8 / 2
4.0
>>> 6 * 7.0
42.0
```

```
>>> 4 + 1.65
5.65
```

A float can be added to an integer, because Python silently converts the integer to a float. However, this implicit conversion is the exception rather the rule in Python - usually you have to convert values manually if you want to operate on them.

Exponentiation

Besides addition, subtraction, multiplication, and division, Python also supports **exponentiation**, which is the raising of one number to the power of another. This operation is performed using two asterisks.

```
>>> 2**5
32
>>> 9 ** (1/2)
3.0
```

You can chain exponentiations together. In other words, you can rise a number to multiple powers. For example, 2^3^2 .

Quotient & Remainder

To determine the **quotient** and **remainder** of a division, use the **floor division** and **modulo** operators, respectively.

Floor division is done using two forward slashes.

The modulo operator is carried out with a percent symbol (%).

These operators can be used with both floats and integers.

This code shows that 6 goes into 20 three times, and the remainder when 1.25 is divided by 0.5 is 0.25.

```
>>> 20 // 6
3
>>> 1.25 % 0.5
0.25
```

In the example above, $20 \% 6$ will return 2, because $3 \cdot 6 + 2$ is equal to 20.

Strings

If you want to use text in Python, you have to use a **string**.

A **string** is created by entering text between **two single or double quotation marks**.

When the Python console displays a **string**, it generally uses single quotes. The delimiter used for a **string** doesn't affect how it behaves in any way.

```
>>> "Python is fun!"
'Python is fun!'
>>> 'Always look on the bright side of life'
'Always look on the bright side of life'
```

There is another [string](#) type in Python called [docstrings](#) that is used for block commenting, but it is actually a [string](#). You will learn about this in future lessons.

Strings

Some characters can't be directly included in a [string](#). For instance, double quotes can't be directly included in a double quote [string](#); this would cause it to end prematurely.

Characters like these must be escaped by placing a **backslash** before them. Other common characters that must be escaped are newlines and backslashes. Double quotes only need to be escaped in double quote strings, and the same is true for single quote strings.

```
>>> 'Brian\'s mother: He\'s not the Messiah. He\'s a very naughty boy!'
'Brian's mother: He's not the Messiah. He's a very naughty boy!'
```

`\n` represents a new line.

Backslashes can also be used to escape tabs, arbitrary Unicode characters, and various other things that can't be reliably printed. These characters are known as escape characters.

Newlines

Python provides an easy way to avoid manually writing "`\n`" to escape newlines in a [string](#). Create a [string](#) with **three sets of quotes**, and newlines that are created by pressing Enter are automatically escaped for you.

```
>>> """Customer: Good morning.
Owner: Good morning, Sir. Welcome to the National Cheese Emporium."""

'Customer: Good morning.\nOwner: Good morning, Sir. Welcome to the National Cheese
Emporium.'
```

As you can see, the `\n` was automatically put in the output, where we pressed Enter.

Output

Usually, programs take **input** and process it to produce **output**. In Python, you can use the [print function](#) to produce output. This displays a textual representation of something to the screen.

```
>>> print(1 + 1)
2
>>> print("Hello\nWorld!")
Hello
World!
```

Try It Yourself

When a [string](#) is printed, the quotes around it are not displayed.

Input

To get input from the user in Python, you can use the intuitively named **input** function. The **function** prompts the user for input, and returns what they enter as a **string** (with the contents automatically escaped).

```
>>> input("Enter something please: ")
Enter something please: This is what\nthe user enters!
'This is what\\nthe user enters!'
```

Try It Yourself

The **print** and **input** functions aren't very useful at the Python console, which automatically does input and output. However, they are very useful in actual programs.

Concatenation

As with integers and floats, strings in Python can be added, using a process called **concatenation**, which can be done on any two strings. When concatenating strings, it doesn't matter whether they've been created with single or double quotes.

```
>>> "Spam" + 'eggs'
'Spameggs'

>>> print("First string + ", " + "second string")
First string, second string
```

Try It Yourself

You can't concatenate strings with numbers (integers). Find out why in the next lesson.

Concatenation

Even if your strings contain numbers, they are still added as strings rather than integers. Adding a **string** to a number produces an error, as even though they might look similar, they are two different entities.

```
>>> "2" + "2"
'22'
>>> 1 + '2' + 3 + '4'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In future lessons, only the final line of error messages will be displayed, as it is the only one that gives details about the type of error that has occurred.

String Operations

Strings can also be **multiplied** by integers. This produces a repeated version of the original **string**. The order of the **string** and the **integer** doesn't matter, but the **string** usually comes first.

Strings can't be multiplied by other strings. Strings also can't be multiplied by floats, even if the floats are whole numbers.

```
>>> print("spam" * 3)
spamspamspam

>>> 4 * '2'
'2222'

>>> '17' * '87'
TypeError: can't multiply sequence by non-int of type 'str'

>>> 'pythonisfun' * 7.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Try It Yourself

Try to multiply a **string** by 0 (zero) and see what happens.

Type Conversion

In Python, it's impossible to complete certain operations due to the types involved. For instance, you can't add two strings containing the numbers 2 and 3 together to produce the **integer** 5, as the operation will be performed on strings, making the result '23'.

The solution to this is **type conversion**.

In that example, you would use the **int** function.

```
>>> "2" + "3"
'23'
>>> int("2") + int("3")
5
```

Try It Yourself

In Python, the types we have used so far have been **integers**, **floats**, and **strings**. The functions used to convert to these are **int**, **float** and **str**, respectively.

Type Conversion

Another example of type conversion is turning user input (which is a **string**) to numbers (**integers** or **floats**), to allow for the performance of calculations.

```
>>> float(input("Enter a number: ")) + float(input("Enter another number: "))
Enter a number: 40
Enter another number: 2
42.0
```

Passing non-integer or float values will cause an error.

Variables

Variables play a very important role in most programming languages, and Python is no exception. A variable allows you to store a value by assigning it to a name, which can be used to refer to the value later in the program.

To assign a variable, use **one equals sign**. Unlike most lines of code we've looked at so far, it doesn't produce any output at the Python console.

```
>>> x = 7
>>> print(x)
7
>>> print(x + 3)
10
>>> print(x)
7
```

Try It Yourself

You can use variables to perform corresponding operations, just as you did with numbers and strings. As you can see, the variable stores its value throughout the program.

Variables

Variables can be reassigned as many times as you want, in order to change their value. In Python, variables don't have specific types, so you can assign a string to a variable, and later assign an integer to the same variable.

```
>>> x = 123.456
>>> print(x)
123.456
>>> x = "This is a string"
>>> print(x + "!")
This is a string!
```

Try It Yourself

However, it is not good practice. To avoid mistakes, try to avoid overwriting the same variable with different data types.

Variable Names

Certain restrictions apply in regard to the characters that may be used in Python [variable](#) names. The only characters that are allowed are letters, numbers, and underscores. Also, they can't start with numbers.

Not following these rules results in errors.

```
>>> this_is_a_normal_name = 7

>>> 123abc = 7
SyntaxError: invalid syntax

>>> spaces are not allowed
SyntaxError: invalid syntax
```

Try It Yourself

Python is a case sensitive programming language. Thus, **Lastname** and **lastname** are two different [variable](#) names in Python.

Variables

Trying to reference a [variable](#) you haven't assigned to causes an **error**.

You can use the **del** statement to remove a [variable](#), which means the reference from the name to the value is deleted, and trying to use the [variable](#) causes an error. Deleted variables can be reassigned to later as normal.

```
>>> foo = "a string"
>>> foo
'a string'
>>> bar
NameError: name 'bar' is not defined
>>> del foo
>>> foo
NameError: name 'foo' is not defined
```

You can also take the value of the [variable](#) from the user input.

```
>>> foo = input("Enter a number: ")
Enter a number: 7
>>> print(foo)
7
```

Try It Yourself

The variables **foo** and **bar** are called **metasyntactic** variables, meaning that they are used as placeholder names in example code to demonstrate something.

In-Place Operators

In-place operators allow you to write code like `x = x + 3` more concisely, as `x += 3`. The same thing is possible with other operators such as `-`, `*`, `/` and `%` as well.

```
>>> x = 2
>>> print(x)
2
>>> x += 3
>>> print(x)
5
```

Try It Yourself

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

In-Place Operators

These operators can be used on types other than numbers, as well, such as **strings**.

```
>>> x = "spam"
>>> print(x)
spam

>>> x += "eggs"
>>> print(x)
spameggs
```

Try It Yourself

Many other languages have special operators such as `++` as a shortcut for `x += 1`. Python **does not** have these.

Using an Editor

So far, we've only used Python with the console, entering and running one line of code at a time. Actual programs are created differently; many lines of code are written in a file, and then executed with the Python [interpreter](#).

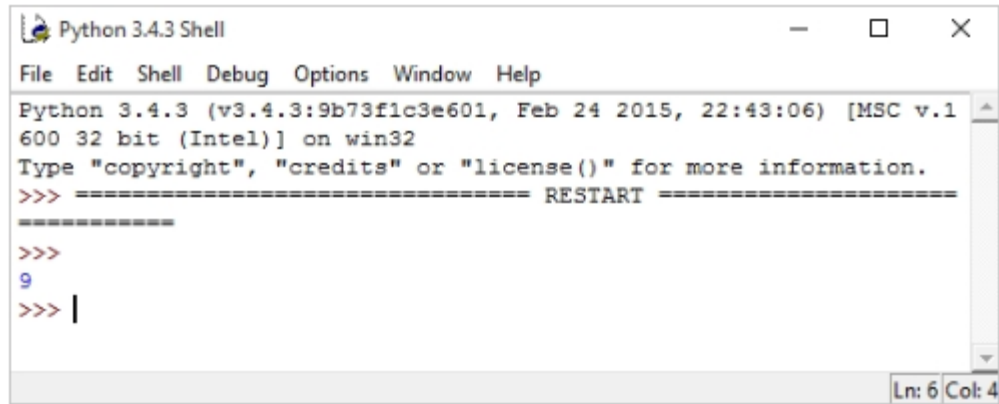
In **IDLE**, this can be done by creating a new file, entering some code, saving the file, and running it. This can be done either with the menus or with the keyboard shortcuts Ctrl-N, Ctrl-S and F5.

Each line of code in the file is interpreted as though you entered it one line at a time at the console.

```
x = 7
x = x + 2
print(x)
```

Try It Yourself

Result:



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1
600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
9
>>> |
```

Ln: 6 Col: 4

Python source files have an extension of `.py`

You can run, save, and share your Python codes on our **Code Playground**, without installing any additional software.
Reference this lesson if you need to install the software on your computer.

End.