**PYTHON PROGRAMMING - INTRODUCTION**

**Python** is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language, Python has a design philosophy that emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax that allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Python interpreters are available for many operating systems, allowing Python code to run on a wide variety of systems. CPython, the reference implementation of Python, is open source software and has a community-based development model, as do nearly all of its variant implementations. CPython is managed by the non-profit Python Software Foundation.

**A Quick Introduction to Python 3 Programming**

This **Quick Introduction to Python 3** aims to teach you just enough Python, so that you can get started in computer programming!

First, you’ll want to [install Python 3](https://docs.python.org/3.5/using/index.html) or you can use the in-browser [Trinket](https://trinket.io/) near the bottom of the page – just edit the code directly in there. If you have Python installed, you can use any [text editor](http://python3.codes/editors-ides/) to start with (but do *not* use a word processor). You might find it useful to [start up the Python interpreter](http://python3.codes/hello-world/) and type in the examples as we go.

**The Python Interpreter**

*Experienced programmers in any other language can pick up Python very quickly, and beginners find the clean syntax and indentation structure easy to learn.* – [Python.org](https://docs.python.org/3/tutorial/)

Do you have the Python interpreter already installed on your computer? To find out, open a [command line](http://lifehacker.com/5633909/who-needs-a-mouse-learn-to-use-the-command-line-for-almost-anything) and type “python” (without the quotes). Hopefully you’ll get something like this:

Python 2.7.6 (default, Mar 22 2014, 22:59:56)

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

Python 2 is almost always readily available on Linux; probably Macs too; Windows, no, not available. If your computer doesn’t have it then go to <https://www.python.org/> and get the latest version (3.5 at this time). Follow the instructions to install it. You’ll also want to do this if your installed version is less than 3.

The traditional first program in Python (and other programming languages) is [“Hello world!”](http://python3.codes/hello-world-2-in-python-3/):

|  |  |
| --- | --- |
| 1 | print "Hello world!" |

Type this after the ‘>>>’ prompt in your interpreter. If it’s version 2 it’ll obligingly print ‘Hello World!’, but version 3 will complain:

File "<stdin>", line 1

print 'Hello World!'

^

SyntaxError: invalid syntax

Version 3 changed the print statement to being a function call (more on that later) which requires parentheses around the thing to be printed (as well as the quotes). In your Python 3 interpreter (invoked as python3 if necessary) type print (“Hello World!”), and hopefully it will! You can also get the same result by just typing in “Hello World!” (with the quotes), without the print function call. Try typing in some arithmetic, like ‘2\*3′. No need to say print! However outside the interpreter you’ll need the print function.

**See**[**Hello World 2, in Python 3**](http://python3.codes/hello-world-2-in-python-3/)**.**

**Statements & Expressions**

The line of text in our first program is called a *statement*. It’s like a sentence in English (sort of). The statement usually appears on a line of its own, and comprises ‘smaller’ bits like identifiers (names for things such as data), keywords (reserved words that tell Python what to do), operators (like arithmetic symbols) etc. You can think of it as an instruction for the computer to do something, like print some text. We’ll meet various kinds of statement, like assignment and conditional statements. An *expression* is a part of a statement where a little bit of computation is required, e.g. x = 2 \* 3

**Assignment Statements**

A [program](https://docs.python.org/release/3.4.3/tutorial/introduction.html#first-steps-towards-programming) is a *set of instructions* (‘[algorithm](https://en.wikipedia.org/wiki/Algorithm)‘) and some [data](https://en.wikipedia.org/wiki/Data) that tell a computer how to perform a task, for example, to convert a temperature in [Centigrade](https://en.wikipedia.org/wiki/Celsius), to [Fahrenheit](https://en.wikipedia.org/wiki/Fahrenheit). This data could take many forms, but for now we’ll assume it’s a set of [numbers](https://docs.python.org/release/3.4.3/tutorial/introduction.html#numbers). And we’ll need some working storage (‘[memory](https://en.wikipedia.org/wiki/Computer_memory)‘) where we can save intermediate results. When we put these results in the memory, we’ll give them [names](https://en.wikipedia.org/wiki/Variable_(computer_science)) and let the computer take care of figuring out where they are in the memory. You can store data in memory using *assignment* statements:

|  |  |
| --- | --- |
| 1  2  3  4 | maxit = 100  temperature = 50.1  distance = 100  greeting = "Hello" |

The first text, before the ‘=’, is the name or *identifier* you have chosen for your item of data; it’s often called a *variable* because (normally) your program may change it later. An identifier is a name used to identify a variable, function, class, module or other object. It starts with an uppercase or lowercase letter, or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9). The ‘=’ assigns a value (e.g. 100) to the variable. Data values have particular types, e.g integers, floats, strings, boolean, complex etc.

**Arithmetic**

*Calculations are simple with Python, and expression syntax is straightforward: the operators +, -, \*and / work as expected; parentheses () can be used for grouping.* – [Python.org](http://docs.python.org/3/tutorial/introduction.html#using-python-as-a-calculator).

Python can do [arithmetic](https://docs.python.org/release/3.4.3/tutorial/introduction.html#using-python-as-a-calculator) such as add, subtract, multiply, and divide, using these operators: + – \* / We can write arithmetic expressions as you’d expect to see them in mathematics, with some small exceptions, e.g. y = m \* x + c You can’t write variables next to each other to signify multiplication; all operations must be explicit. Operands must be one of:

* **Integer**: whole [numbers](https://docs.python.org/release/3.4.3/tutorial/introduction.html#numbers), zero, positive, or negative, e.g. 1, 2, -3, 0
* **Floating Point**: [numbers](https://docs.python.org/release/3.4.3/tutorial/introduction.html#numbers) with decimal point, e.g. 3.14, -0.8

If x is 2, and y is 3, then:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** |  | **Description** | **Example** |
| + | Add | Adds the values on either side of the operator. | x + y -> 5 |
| – | Subtract | Subtracts right side operand from left side operand. | x – y -> -1 |
| \* | Multiply | Multiplies values on either side of the operator | x \* y -> 6 |
| / | Divide | Divides left side operand by right side operand | x / y -> 0.666… |
| % | Modulus | Divides left side operand by right side operand and returns remainder | x % y -> 0 |
| \*\* | Exponent | Raise left operand to power of right operand | x \*\* y -> 8 |
| // | Floor Division | Division where the result is rounded away from zero (towards negative infinity) | 7//2 = 3 and 7.0//2.0 = 3.5 |

**A Programming Example**

Let’s have an example. Let’s convert a temperature given in Centigrade degrees,°C, to Fahrenheit,°F. From Physics, we know that there’s a simple formula: F = C \* 9 / 5 +32. That is, multiply the value in Centigrade by 9, divide that by 5, and then add 32. So what is 0°C in F? It’s 32. The freezing point of water is 0° in C, but 32° in F. How about 100°C? The order of operations in C \* 9 / 5 isn’t important, so let’s divide 100 by 5 first. That’s 20. Now multiply by 9; that’s 180. Now add 32. That’s 212°, the boiling point of water in Fahrenheit.

**Note** that in the following, I’m a bit carefree about integers and floats; I use integers for simplicity, but in serious applications you might do better to use floats if they better represent the real-world values!

|  |  |
| --- | --- |
| 1  2  3 | C = 50  F = C \* 9 / 5 + 32  print(F) |

Our program (let’s call it C2F.py) is a *sequence* of instructions, executed one after the other, starting with the first one, C = 50:

1. Set the variable C to the integer value of 50 (If you prefer, you can write ‘.0’ after each number.)
2. Multiply C by 9, divide the result by 5, and add 32 to that
3. [Print](https://docs.python.org/release/3.4.3/tutorial/inputoutput.html) the answer

**String Theory**

In our “Hello, World!” example, we wrote it one time with *“double quotes”*, and one time with *‘single quotes’*: ‘Hello, World!’. Text, when enclosed in quotes (either kind) is called a [**string**](https://docs.python.org/release/3.4.3/tutorial/introduction.html#strings). Strings are very useful not only for greetings, but also for programs to tell you the results of computations, or to tell you what input is needed next, or for storing information like people’s names, or for labelling graphs, etc. Note that if you need a string to contain quotes, you must *either* use the other style of quote (single or double) to enclose it, *or* ‘escape’ the internal quote with a backslash: “This is Freddie’s cat” or ‘This is Freddie\’s cat’.

You can do a limited amount of ‘arithmetic’ on strings – you can add or multiply them! So if a contains ‘a’, and b contains ‘b’, then a + b is ‘ab’ (this is called concatenation), and a \* 3 is ‘aaa’. You can’t subtract or divide them however.

**Input & Output**

Outputting a result can be done with print() as we’ve just seen. We could make it a little more helpful if we write it like this: print(“The temperature is “, F, “F”).

This is all well and good, whenever we want to convert 50°C to F. But it’d be much more useful if we could somehow tell it any value of C! We can do this just by replacing the first line, C = 50, with: C = input(“Enter the Centigrade temperature:”) It prints out “Enter the Centigrade temperature:” (without the quotes) and waits for the user to type in something.

But if we try to do arithmetic on C, Python will complain about ‘unsupported operand type(s)’. This is because input() delivers the user’s input as a string of characters – it doesn’t know that we want it as a number. So we feed that string into int() (for integer) or float() (for decimal-pointed numbers). When it has that number it does the arithmetic, and prints out the result.

**Branching Conditions**

*In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test … is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to). –*[*Python.org*](https://docs.python.org/3.5/tutorial/introduction.html#first-steps-towards-programming)

Conditional statements are features of a programming language which perform different computations or actions depending on whether a programmer-specified *boolean condition* evaluates to true or false. If the condition is true, one path of code is taken, but if not, another path is taken. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5 | x = 3  if x > 0:      print("x is positive")  else:      print("x is zero or negative") |

Let’s meet another data type:

* **Boolean**: True or False. For example, 4 > 3 is True, 2 > 3 is False, 2 == 2 is True, 3 == 2 is False.

*Boolean has only 2 values: True or False*. Usually the result of a comparison, e.g. x > 3 is True if x is 4, but False if x is 2. It can also result from testing for equality, e.g. x == 3 is True if x is 3, False otherwise. *Note the double equals!* You will ( I repeat, **will**) at some point in your Python programming career, write x = 3 when you want to know if x is 3!

Here are the comparison operators that you can use in boolean expressions:

* == equality
* != not equal
* > greater than
* >= greater than or equal
* < less than
* <= less than or equal

**If Statements**

*Python knows the usual control flow statements that other languages speak — if, for, while and range— with some of its own twists, of course.* – [Python.org](https://docs.python.org/3/tutorial/controlflow.html)

What if we want to convert temperatures in the other direction – F to C? Well the formula is C = (F – 32) \* 5 / 9. Why don’t you re-write the previous program and call it F2C.py Whenever we want to convert Fahrenheit to Centigrade, we’ll use this one. So whenever you have a temperature conversion to do, you’ll reason something like this:

If I want to convert C to F then

run C2F.py

otherwise (I want to convert F to C)

run F2C.py

Now we have 2 programs that look very similar and have the same physical constants in them (9, 5, and 32). We could simplify things a bit, if we could combine them into one! But how will the computer know which conversion we want? Sounds like we need another input()!

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | which = input("Is your temperature C or F?").lower()  value = int(input("What is its value? "))  if which == 'c':      result = value \* 9 / 5 + 32  elif which == 'f':      result = (value - 32) \* 5 / 9  else:      result = 'Please enter C or F'  print(result) |

*Indentation is Python’s way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount. –*[*Python.org*](https://docs.python.org/3.5/tutorial/introduction.html#first-steps-towards-programming)

1. Let’s assume that the user types a ‘C’ or an ‘F’; .lower() converts to lowercase (so it doesn’t matter if the user types ‘C’ or ‘c’); this goes into the variable ‘which’
2. ‘[if](https://docs.python.org/release/3.4.3/tutorial/controlflow.html#if-statements)‘ test the value of the following Boolean expression, which == ‘c’. Note the ‘:’ at the end of the line. If the expression is True then the following block is run:
3. –> compute the Fahrenheit value, put it in result variable
4. else if ‘which’ contains ‘f’:
5. –> compute the Centigrade value, put it in result variable
6. otherwise:
7. –> store ‘Please enter C or F’ into result
8. print the result

*There can be zero or more [elif](https://docs.python.org/3.5/reference/compound_stmts.html" \l "elif) parts, and the*[*else*](https://docs.python.org/3.5/reference/compound_stmts.html#else)*part is optional. The keyword ‘[elif](https://docs.python.org/3.5/reference/compound_stmts.html" \l "elif)‘ is short for ‘else if’, and is useful to avoid excessive indentation. An*[*if*](https://docs.python.org/3.5/reference/compound_stmts.html#if)*… [elif](https://docs.python.org/3.5/reference/compound_stmts.html" \l "elif) … [elif](https://docs.python.org/3.5/reference/compound_stmts.html" \l "elif) … sequence is a substitute for the switch or case statements found in other languages. –*[*Python.org*](https://docs.python.org/3.5/tutorial/controlflow.html#if-statements)

**Looping**

A [loop](https://en.wikipedia.org/wiki/Control_flow#Loops) is a sequence of statements which is specified once but which may be carried out several times in succession. The code “inside” the loop (the *body* of the loop) is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met, or [indefinitely](https://en.wikipedia.org/wiki/Infinite_loop).

**While Loops**

Suppose that we have several temperatures that we want to convert. Well, we could run the appropriate program (C2F.py or F2C.py) several times. But it would be nicer if the program could ‘start over’ by itself, till I indicate I’m done:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | done = False  while not done:      C = int(input("Temperature in C:"))      F = C \* 9 / 5 + 32      print(F)      done = input("Another?") != 'y' |

1. done is a boolean variable, initialised to ‘False’
2. while means “repeat the following block as long as the test (‘not done’) is True:
3. –> get the user’s value of C
4. –> compute F
5. –> reset the boolean done: if the user answers with ‘y’ then it’s True, otherwise False

**For Loops**

*The*[*for*](https://docs.python.org/3.5/reference/compound_stmts.html#for)*statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s*[*for*](https://docs.python.org/3.5/reference/compound_stmts.html#for)*statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. –*[*Python.org*](https://docs.python.org/3.5/tutorial/controlflow.html#for-statements)

while isn’t the only looping construct in Python; perhaps better-known is the [for-loop](https://docs.python.org/3.5/tutorial/controlflow.html#for-statements), which repeats the enclosed block a fixed number of times:

|  |  |
| --- | --- |
| 1  2 | for i in range(10):      print(i) |

The built-in function [range()](https://docs.python.org/3/library/stdtypes.html#range) iterates over a sequence of numbers, generating arithmetic progressions. In this example, it generates integers from 0 to 9.

**Compound Data**

Sometimes you need several items of data together, representing multiple measurements or different facets of the world; such as arrays, records, files, data bases etc.

**Lists**

*Lists (known as arrays in other languages) are one of the compound data types that Python understands. Lists can be indexed, sliced and manipulated with other built-in functions.* – [Python.org](https://docs.python.org/3/tutorial/introduction.html#lists)

Alternatively, instead of asking the user to supply all the values, we could put them straight into the program. Maybe you’re just writing a quick little program to do some number crunching, so you don’t need the fancy prompts! Well, we can put the data in a list (‘array’ in other languages, and math).

|  |  |
| --- | --- |
| 1  2  3 | temps = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  for temp in temps:      print(temp, "C is ", temp \* 9 / 5 + 32, "F") |

Lists might contain items of different types, but usually the items all have the same type. Also, lists can contain lists (aka, 2D arrays)!

**Files**

Lists are very handy, but you might prefer sometimes to process data that’s in a file rather than right there in your program. We can do this very easily, with very little change to our program:

|  |  |
| --- | --- |
| 1  2  3  4 | temps = open('Temperatures.C')  for temp in temps:      \_ = int(temp)      print(\_, "C is ", \_ \* 9 / 5 + 32, "F") |

Instead of the list, we now have open(‘Temperatures.C’) (or whatever name your file has if not Temperatures.C). What is read from the file, each time the for loop iterates, is a string, hence the \_ = int(temp). The \_ is just some anonymous variable that we needn’t bother giving a name to – but you can, if you prefer (e.g. because it helps document the code).

**Looping Around Conditionals**

A very common pattern in programming is to loop over some data while testing it for some condition so as to process it accordingly. The simplest example is probably where some abnormal value of the data is used to signal the end of the data, e.g. if we’re calculating average hieght, zero or negative indicates end of data:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | sum = 0.0  num = 0  done = False  while not done:      height = float(input("Enter a height or 0 to finish: "))      if height <= 0:          done = True      else:          sum += height          num += 1    print ("Average height is ", sum / num) |

*There is an error in this program.* What is it, and how would you fix it?

**Comments**

Another important thing to know about is ‘comments’. You can put text in your program that isn’t Python code, such as explanations about how the code works. Anything between triple quotes (single or double, so long as they’re the same) will be ignored. Anything after ‘#’ on the same line will be ignored.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | '''   Test the first N numbers for primality.      Count up to N, dividing by primes known so far.      Add this number to the list of primes if none divide into it.  '''  N = 1000                    # N is how many numbers to test  primes = []                 # List of primes so far  for n in range(2,N+1):      # For each number from 2 to N:      for p in primes:            # For each prime so far:          if n % p == 0: break        # If it divides into it, it's not prime      else:                       # No primes divided into it so          primes.append(n)            # Add it to the list  print(primes) |

When range() has 2 arguments, the first one is the starting number, and the second one is the *next* number after the end of the range! So we have to add 1 if we want the range to include the Nth number. The ‘%’ operator is modulus, i.e. the remainder left by dividing the first operand by the second. It’s very useful to tell if one number is evenly divisible by another, because if it is, the remainder is 0.

**Error Handling**

What happens when things go wrong? They *will*! The user might enter a value that isn’t right, such as a value out of range, or ‘K’ when we wanted ‘C’ or ‘F’. In simple cases like that you can just have an ‘else’ clause to catch the values that are ‘none of the above’, and print out an error message. For more difficult cases, e.g. a number has invalid digits, you can use Python’s [exception handling](https://docs.python.org/release/3.4.3/tutorial/errors.html#handling-exceptions). It looks like this:

|  |  |
| --- | --- |
| 1  2  3  4 | try:      number = int (string)  except:      print(string, "isn't a valid number") |

So Python will try to convert string to an integer, but if it can’t, it will perform the except block, otherwise execution continues after the except block.  
Here’s a better version of our temperature converter:

**Functions**

*The core of extensible programming is defining functions. Python allows mandatory and optional arguments, keyword arguments, and even arbitrary argument lists.* – [Python.org](https://docs.python.org/3/tutorial/controlflow.html#defining-functions)

A function is a piece of code ‘wrapped up’ in a way that enables us to use it like we used the range() function – it gets a name and some arguments that supply its input values. So if we wanted, for example, to make our temperature conversions re-usable, we could declare these functions:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def C2F(value):      ''' Centigrade to Fahrenheit '''      return value \* 9 / 5 + 32    def F2C(value):      ''' Fahrenheit to Centigrade '''      return (value - 32) \* 5 / 9    print(C2F(0))  print(C2F(100))  print(F2C(32))  print(F2C(212)) |

[def](https://docs.python.org/3/tutorial/controlflow.html#defining-functions) declares that a function definition is coming. First, the name, then in parentheses, the value or values. The line ends with a colon indicating that an indented block of code will follow. There is then an optional comment, called a *docstring*, which may be used by tools to automatically document your code. When the function is called, the conversion is calculated, and the result appears in the place where the function was called.

Functions are an extremely important addition to your programming repertoire! They enable:

* [Unit testing](https://en.wikipedia.org/wiki/Unit_testing), like the above prints of known correct values;
* [Modularisation](https://en.wikipedia.org/wiki/Modular_programming), dividing your code into easier to understand pieces;
* [Object-oriented programming](https://docs.python.org/3/tutorial/classes.html) which further improves code modularity;
* [Reusability](https://en.wikipedia.org/wiki/Reusability), either in the same program, or in others by use of [modules](https://docs.python.org/3/tutorial/modules.html) (files which can be imported);
* [Recursion](https://en.wikipedia.org/wiki/Recursion_(computer_science)), where a problem can be solved by repeatedly breaking it down into smaller instances until a trivial base case is reached.

**What Next?**

* Visit [Python.org](https://www.python.org/)
* Choose [your editor / IDE](http://python3.codes/editors-ides/)
* Read [The Python Tutorial](https://docs.python.org/release/3.4.3/tutorial/index.html)
* [What’s New In Python 3.0](https://docs.python.org/3/whatsnew/3.0.html)
* Look at [the standard library](https://docs.python.org/3/tutorial/stdlib.html)
* Try [these beginner’s examples](http://python3.codes/category/beginner/)
* Learn about [functions](https://docs.python.org/3.5/tutorial/controlflow.html#defining-functions) and [libraries](http://pythontips.com/2013/07/30/20-python-libraries-you-cant-live-without/)
* [PEP 0008 — Style Guide for Python Code](https://www.python.org/dev/peps/pep-0008/)