Python for informatics

Writing a program

When we want to write a program, we use a text editor to write the Python instructions into a file, which is called a script. By convention, Python scripts have names that end with .py.
To execute the script, you have to tell the Python interpreter the name of the file. In a Unix or Windows command window, you would type python hello.py as follows:

```
csev$ cat hello.py
print 'Hello world!'
csev$ python hello.py
Hello world!
csev$
```

The "csev$" is the operating system prompt, and the "cat hello.py" is showing us that the file "hello.py" has a one-line Python program to print a string.
We call the Python interpreter and tell it to read its source code from the file "hello.py" instead of prompting us for lines of Python code interactively.
You will notice that there was no need to have quit() at the end of the Python program in the file. When Python is reading your source code from a file, it knows to stop when it reaches the end of the file.

Terminology: interpreter and compiler

An interpreter reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly. Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python.
Some of the lines of Python tell Python that you want it to remember some value for later. We need to pick a name for that value to be remembered and we can use that symbolic name to retrieve the value later. We use the term variable to refer to the labels we use to refer to this stored data.

```
>>> x = 6
>>> print x
6
>>> y = x * 7
>>> print y
42
>>>
```

It is the nature of an interpreter to be able to have an interactive conversation as shown above. A compiler needs to be handed the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Values and types

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!'
These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.
The print statement also works for integers. We use the python command to start the interpreter.

```
python
>>> print 4
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.
An assignment statement creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

This example makes three assignments. The first assigns a string to a new variable named message; the second assigns the integer 17 to n; the third assigns the (approximate) value of π to pi.
To display the value of a variable, you can use a print statement:

```
>>> print n
17
>>> print pi
3.14159265359
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Boolean expressions

A **boolean expression** is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True and False are special values that belong to the type bool; they are not strings:

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

The == operator is one of the comparison operators; the others are:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
x is y               # x is the same as y
x is not y           # x is not the same as y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols for the same operations. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. There is no such thing as =< or =>.

Logical operators

There are three **logical operators**: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example,
x > 0 and x < 10
is true only if x is greater than 0 *and* less than 10.
n%2 == 0 or n%3 == 0 is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.
Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y.
Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as “true.”

```
>>> 17 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it until you are sure you know what you are doing.

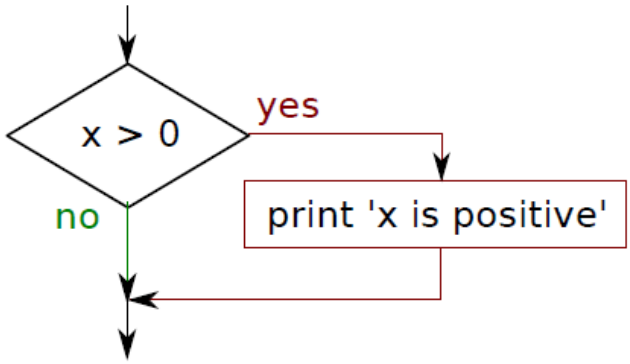
This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Conditional execution

Conditional statements give us this ability. The simplest form is the if statement:

```
if x > 0 :
    print 'x is positive'
```

The boolean expression after the *if* statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the *if* statement are indented.



There is no limit on the number of statements that can appear in the body, but there must be at least one. Occasionally, it is useful to have a body with no statements (usually as a placekeeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

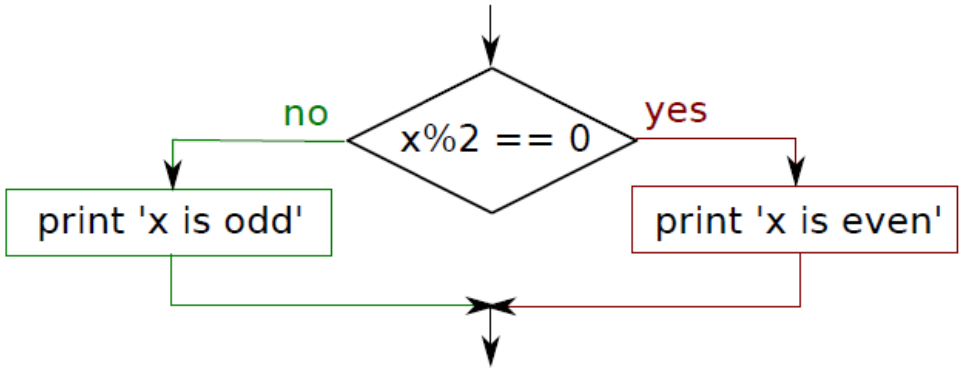
```
if x < 0 :
    pass          # need to handle negative values!
```

Alternative execution

A second form of the if statement is **alternative execution**, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0 :
    print 'x is even'
else :
    print 'x is odd'
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.



Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

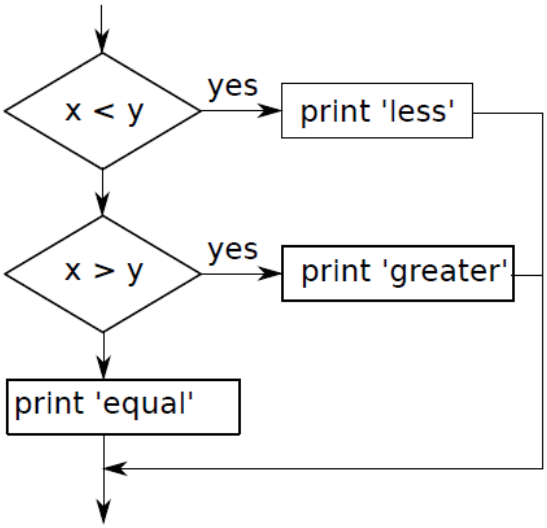
This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is **a chained conditional**:

```
if x < y:
    print 'x is less than y'
elif x > y:
    print 'x is greater than y'
else:
    print 'x and y are equal'
```

elif is an abbreviation of "else if." Again, exactly one branch will be executed.



There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.

Function calls

A function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Example of a function call:

```
>>> type(32)
<type 'int'>
```

The name of the function is type. The expression in parentheses is called the argument of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the type function, is the type of the argument.
The max and min functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

Built-in function is the len function which tells us how many items are in its argument. If the argument to len is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
>>>
```

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

A string is a sequence

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

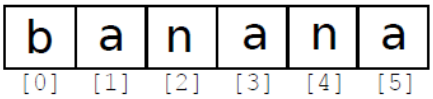
```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement extracts the character at index position 1 from the fruit variable and assigns it to the letter variable.
The expression in brackets is called an index. The index indicates which character in the sequence you want (hence the name).
But you might not get what you expect:

```
>>> print letter
a
```

```
>>> letter = fruit[0]
>>> print letter
b
```

So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th (“two-eth”) letter.



len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print  'All right, bananas.'
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

```
Your word, Pineapple, comes before banana.
```

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

string methods

Strings are an example of Python objects. An object contains both data (the actual string itself) and methods, which are effectively functions that are built into the object and are available to any instance of the object.
Python has a function called dir which lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
>>>
```

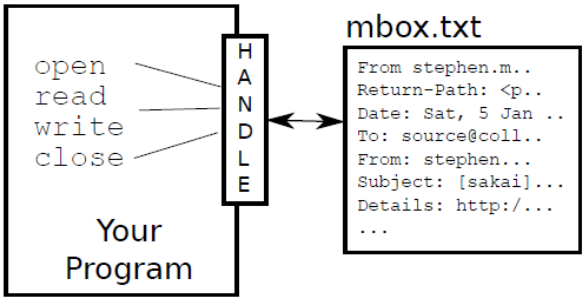
While the dir function lists the methods, and you can use help to get some simple documentation on a method, a better source of documentation for string methods would be https://docs.python.org/2/library/stdtypes.html#string-methods.

Opening files

When we want to read or write a file (say on your hard drive), we first must **open** the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file mbox.txt, which should be stored in the same folder that you are in when you start Python. You can download this file from www.py4inf.com/code/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print fhand
<open file 'mbox.txt', mode 'r' at 0x1005088b0>
```

If the open is successful, the operating system returns us **a file handle**. The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.



This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Reading files

While the file handle does not contain the data for the file, it is quite easy to construct a for loop to read through and count each of the lines in a file:

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
print 'Line Count:', count

python open.py
Line Count: 132045
```

The reason that the open function does not read the entire file is that the file might be quite large with many gigabytes of data. The open statement takes the same amount of time regardless of the size of the file. The for loop actually causes the data to be read from the file.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the read method on the file handle.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print len(inp)
94626
>>> print inp[:20]
From stephen.marquar
```

In this example, the entire contents (all 94,626 characters) of the file mbox-short.txt are read directly into the variable inp. We use string slicing to print out the first 20 characters of the string data stored in inp.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.