**INTRODUCTION TO PYTHON**

[Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation.](https://www.bing.com/ck/a?!&&p=604cfe62ae5b2c5aJmltdHM9MTY5ODAxOTIwMCZpZ3VpZD0xYWE2MGJlNi02ZDEzLTZiZTAtM2IzMi0xOGIzNmM4ZDZhZDkmaW5zaWQ9NTcyMA&ptn=3&hsh=3&fclid=1aa60be6-6d13-6be0-3b32-18b36c8d6ad9&psq=what+is+pyhton+interpreter&u=a1aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvUHl0aG9uXyhwcm9ncmFtbWluZ19sYW5ndWFnZSk&ntb=1" \o "en.wikipedia.org" \t "_blank)

Python is an interpreted language.

Python can be used on server to create web applications.

This is you print in python:

print("Hello, World!")

**Python in the Job market:**

There are various job fields that Python has to offer. Integrating applications with MySQL is in high demand because MySQL and Python are both open-source applications. This means that some companies are switching over from their current expensive systems to open-source systems. YouTube and BitTorrent are examples of companies that use Python.

Network programming in Python is another option, which requires an extensive knowledge of how networking is controlled. Another fields for a Python programmer are Software Engineer, Software developer, Research Analyst, Data Analyst and Data Scientist; you will often be required to have database experience when working in this field.

**History of Python:**

Python was conceived in the late 1980s and Guido van Rossum started implementing it at CWI in the Netherlands in December 1989. It is a relatively simple language that includes a standard library that provides modules for a large number of processes that programs deal with. This approach keeps Python simple yet reliable programming language.

Python is implemented in C and relies on the extensive, well understood, portable C libraries. It fits seamlessly with UNIX, Linux, and POSIX environments. Since these standard C libraries are widely available for the various MS-Windows variants, and other non-POSIX operating systems, Python runs similarly in all environments. The Python programming language was created based on lessons learned during language and operating system support. Python is built from concepts in the ABC and Modula-3 languages.

**Comments in Python:**

In programming, comments are a programming language construct used to insert human-readable text in the source code of a program.

Comments could be used for a wide range of purposes, for example:

Augmenting program code with basic descriptions to generate external documentation.

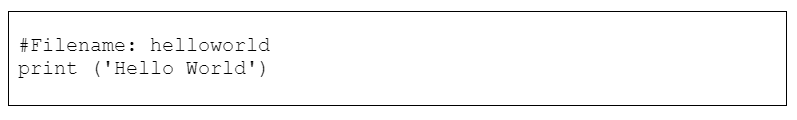
Integration with source code management systems and other kinds of external programming tools.

Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

**Creating application using Python:**

Now that you understand how the Python Interpreter functions, you will create your first application using Python.

Indenting your code correctly is essential. Indentation in Python identifies what code belongs to what section of code.



The first line of code is commented out because this only indicates to the programmer what the file is called

**Escape Sequences**

 Description

 Token

Backslash character (\)

 \\

New line feed

\n

Tab

\t

Vertical tab

\v

Backspace

\b

Carriage return

\r

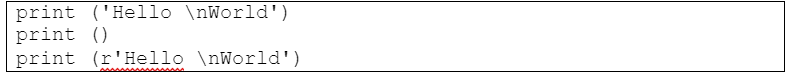
Single quote (useful in strings enclosed in single quotes: **‘hello \ ‘World’**)

\'

Double quote (useful in strings enclosed in single quotes: **“hello \ “World”**)

\"

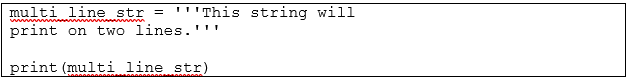
**Example: hello-world**



**Output:**



If you do not want to use the **\n** escape sequence, you can use three single quotes ‘’’ or three double quotes “""characters:

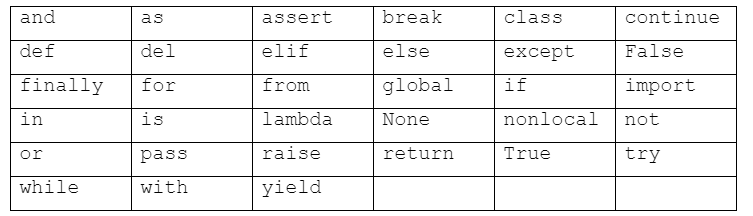


**Output:**



The following identifiers are used as reserved words, or keywords in Python, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

**Reserved words**



**Variables**

Every variable is created with an initial value. A variable can be in three states:

* Variable creation (Declaration)
* Variable assignment (Initialization)
* Variable changed (Execution)

Once the code which created the variable has finished executing, the variable is destroyed.

The rules include the start and continuation characters. Variable names may contain any upper or lower case letter (A–Z, a–z), a number, or the underscore character. They may not begin with a number or contain spaces. Continuation characters are any characters except whitespace characters like tab and space.

Here are a few examples of **valid** variable names:

* c
* ref\_number
* admin
* aVeryLongName

Here are a few examples of **invalid** variable names:

* True
* $name
* 12Graph

In Python identifiers are case sensitive, so for example, firstName, FirstName, FIRSTNAME, and firstname are four different identifiers. A second rule is that variables cannot have the same name as Python’s keywords. We can find out what keywords are in Python, by using the function called dir(). If this function is called with the \_\_builtins\_\_ attribute, it returns a list of Python’s built-in attributes.

The \_\_builtins\_\_ module contains all Python’s built-in attributes, which can be used with the dir()function. The ones that are returned are identified with the following characteristics:

* Python’s built-in exceptions start with a capital letter.
* The rest are either functions or data type names.
* Identifiers that start and end with one or two underscores are special methods.

**NOTE**

 All of the methods, exceptions, and functions contained in

dir(\_\_builtins\_\_), dir(\_\_doc\_\_), dir(\_\_name\_\_), and dir(\_\_package\_\_) cannot be used as variable names.

**Using Variables**

All variables have to be assigned to a data type like a **string** (a series of characters) or an **integer** (positive and negative whole numbers).

**Example 1: Automatic assignment of variables:**

num1 = 15 #num1 is automatically assigned as an integer

#value

num2 = 30 #num2 is also assigned as an integer value

print (num1 + num2) #answer is printed

**Output:**

>>>

45

>>>

**Casting**

Casting can be done in two ways:

* **Implicitly:** The compiler automatically casts a value from one data type to another when assured that there will be no data loss.
  + - * **For example.** casting from an integer variable to a floating-point variable or casting from an integer variable to another integer variable
* **Explicitly:** A value cannot be automatically cast from one data type to another if it will result in data loss. Extra code has to be written to ensure that the value stays the same and only the data type changes.
  + - * **For example,** casting from a floating-point value to an integer value

You often have to convert the values you input, in order to have a correct output. The following example illustrates this.

**Example 2: Explicit string and integer casting:**

num1 = 15 #num1 variable is automatically assigned

#as an integer value

num2 = "30" #num2 variable is automatically assigned

#as a string value

ans = num1 + num2 #ans is assigned to num1 added to num2

print ("answer:", ans) #ans is printed

The previous example will raise an error and will not execute at all. This is because **num1** and **num2** are not defined using the same data types or data types that can be cast automatically: num1 is an **integer** data type and num2 is a **string** data type. This means that and will not know if it should be assigned to an **integer** or a **string.** Either **num1** or **num2** should be cast to match the other one’s data type. The solution below will clarify any misunderstanding.

**Solution:**

num1 = 15 #num1 variable is automatically assigned

#to an integer value

num2 = "30" #num2 variable is automatically  assigned

#to a string value

ans = num1 + int(num2) #num2 is cast to integer type and added #to num1

print ( "num2 cast to integer  ",ans)

ans = str(num1) + num2   #num1 is cast to a string type

print( "num1 cast to string  ",ans) #to num1

print ("num2 cast to integer: ", ans)

ans = str(num1) + num2 #num1 is cast to string and num2 gets

            #added

print ("num1 cast to string:", ans)

**Output:**

>>>

num2 cast to integer: 45

num1 cast to string: 1530

num2 cast to integer:  1530

num1 cast to string: 1530

>>>

Notice that the two print statements have different outputs. The first print statement prints the integer values added together **(15 + 30 = 45)**. The second print statement prints the characters of **num1** and **num2** concatenated/joined together **(‘1’ + ‘5’ + ‘3’ +’0’ = ’1530 ’)**. The difference between using ‘+’ or ‘,’:

>>>            #interpreter

>>> print ("Gau","teng")

Gau teng

>>> print ("Gau"+"teng")

Gauteng

>>>

The following example illustrates how Python’s interpreter automatically casts two values and adds them together:

**Example 3: Casting values:**

num1 = 15.60 #num1 variable is a floating point

#number

num2, num3 = 30, 32 #num2 variable is a integer

ans = num1 + num2 + num3 #converted to floating point

print ("The answer:", ans) #ans is printed

**Output:**

>>>

The answer: 77.6

>>>

The previous piece of code will execute with no errors because it is possible to cast an integer to a floating-point without any data loss.

**Example 4: Casting values:**

i\_number = int(15)

s\_number = str("12.543")

s\_characters = str("five")

f\_number = float(123.5675)

f = float(i\_number)

print ("An Integer cast as a floating point number:", f)

i = int(f\_number)

print ("A floating point number cast as an Integer:", i)

f = float(s\_number)

print ("A number string cast as a floating point number:", f)

f = float(s\_characters)

print ("A number string cast as a floating point number:", f)

**Output:**

>>>

An Integer cast as a floating-point number: 15.0

A floating-point number cast as an Integer: 123

A number string cast as a floating point number: 12.543

Traceback (most recent call last):

 File "C:\float\_casting", line 15, in <module>

f = float(s\_characters)

ValueError: could not convert string to float: five

>>>

Notice that no errors occur when converting from an integer to a float. This will always be the case because an integer can be cast to a float data type **implicitly** because there will be no data loss. However the reverse is not true; a floating-point number cannot be implicitly cast into an integer as this will result in data loss (all data after the precision (.) will be lost), because the floating-point value **does not** get rounded off to the nearest whole number when **implicitly** converting to an integer (as seen in the above example).

The third printed line casts a string to a float successfully, because *s\_number* is in the correct format. The rules to convert a string to a float are:

* The string should only contain numbers.
* Other than numbers the following are allowed:
  + - * Only one dot (.) character. Indicates the decimal starts after the dot (.) character.
      * A ‘+’ or ‘−‘ character at the beginning of the string. This indicates that the number is either positive or negative.