



- Python is a general-purpose, interpreted, interactive, object-oriented, and high-level programming language.
- Created by **Guido van Rossum** during 1985- 1990, and first released in 1991.
- “Python is general purpose programming language that also works nicely as a scripting language.”

It is used for:

- web development (server-side)
- software development
- system scripting
- handle database and big data
- complex mathematics
- rapid prototyping



Characteristics of Python Programming:

- It supports functional/structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python Syntax compared to other programming languages:

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Python Features

1) Easy to Learn and Use

Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language. There is no use of the semicolon or curly-bracket, the indentation defines the code block.

2) Expressive Language

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type **print("Hello World")**. It will take only one line to execute, while Java or C takes multiple lines.

3) Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

4) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

5) Free and Open Source

Python is freely available for everyone. It is freely available on its official website ***www.python.org***. The open-source means, "Anyone can download its source code without paying."

6) Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc.

7) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.

8) Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

9) GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQt5, Tkinter, Kivy are the libraries which are used for developing the web application.

10) Integrated

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

11) Embeddable

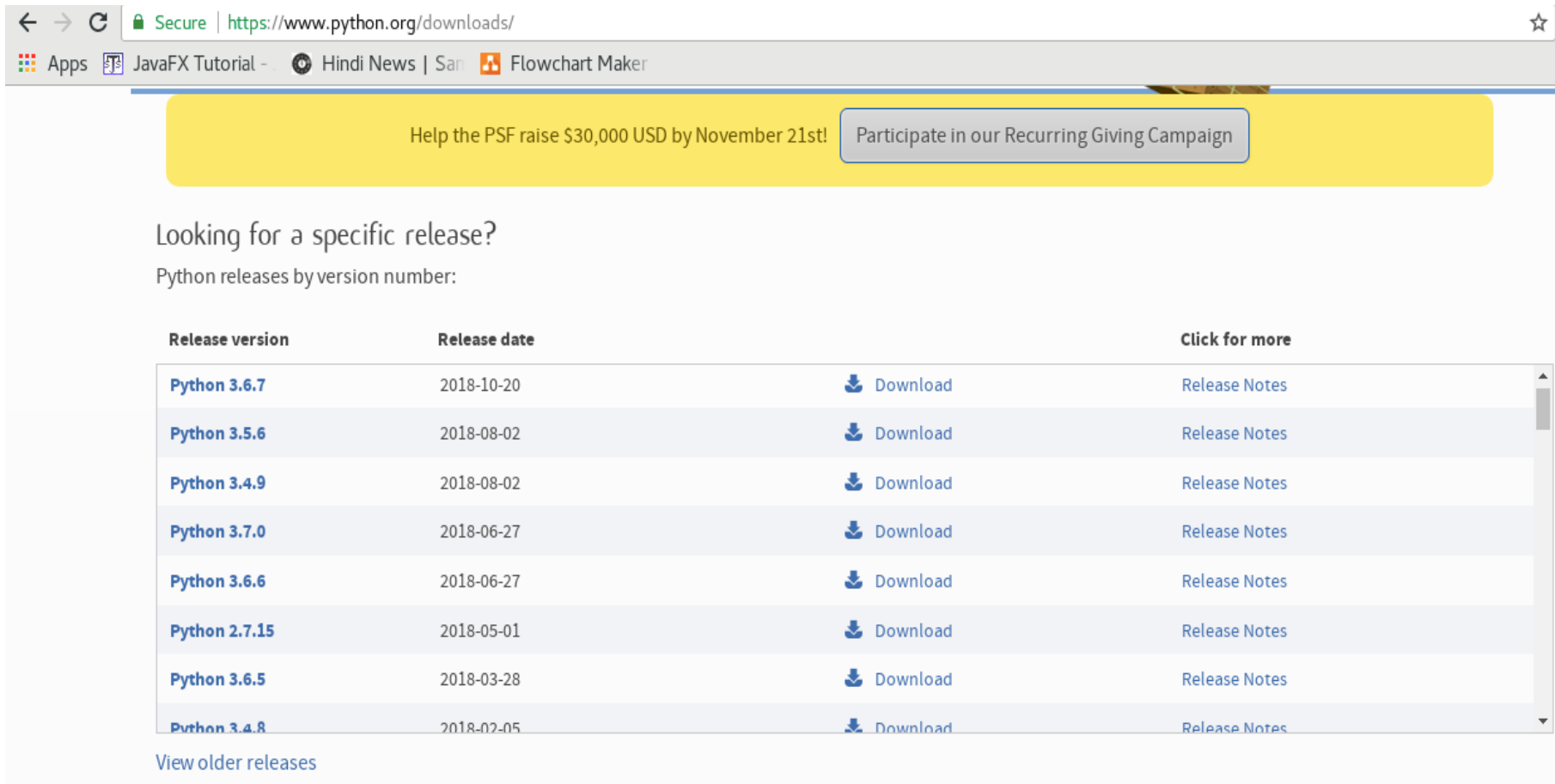
The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

12) Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to **x**, then we don't need to write **int x = 15**. Just write **x = 15**.

Installation on Windows

1. Visit the link <https://www.python.org/downloads/> to download the latest release of Python.



The screenshot shows the Python.org downloads page. At the top, there's a yellow banner with a message about the PSF fundraising campaign. Below that, a section titled "Looking for a specific release?" lists Python releases by version number. A table displays the following data:

Release version	Release date	Click for more	
Python 3.6.7	2018-10-20	Download	Release Notes
Python 3.5.6	2018-08-02	Download	Release Notes
Python 3.4.9	2018-08-02	Download	Release Notes
Python 3.7.0	2018-06-27	Download	Release Notes
Python 3.6.6	2018-06-27	Download	Release Notes
Python 2.7.15	2018-05-01	Download	Release Notes
Python 3.6.5	2018-03-28	Download	Release Notes
Python 3.4.8	2018-02-05	Download	Release Notes

Below the table, there is a link to [View older releases](#).

2. Double-click the executable file, which is downloaded; the following window will open.



Do not forget to check on “Add python to PATH”, otherwise you have to set the path manually.

3. To check if you have python installed on a Windows PC, search in the start bar for Python **or** run the following on the Command Line (cmd.exe):

```
C:\Users\Dir Name>python --version
```

4. Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

Python provides us two ways to run a program

- 1)** Using Interactive interpreter prompt
(Directly from Command Prompt)
- 2)** Using a script file
(Saving program in separate file having .py extension)

i) Interactive Mode Programming:

Invoking the interpreter without passing a script file as a parameter brings up the following prompt –

```
$ python  
>>>
```

Type the following text at the Python prompt and press the Enter –

```
>>>print ("Hello, Python!")
```

//we have used print() function to print the message on the console.

ii) Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Type the following source code in a “test.py” file –

```
print ("Hello, Python!")
```

Now, try to run this program as follows –

```
$ python test.py
```

Running the program using IDLE

- IDLE is Python's Integrated Development and Learning Environment. It has two main window types, the **Shell window** and the **Editor window**. It is possible to have multiple editor windows simultaneously.
- From start menu you can open IDLE **shell window**. You can again type in `print("hello!")` and so forth, and the shell will do the printing. As you can see, it's interactive. Python responds to every line of code you enter.

- Opening up a new window (from “**File->New File**” option) will create a **script file** in editor window. Here, `print("hello!")` does not immediately produce output. That is because this is a script file **editing window**, which means the commands won't execute until the file is saved and run.
- You can run the script by going "**Run --> Run Module**" or simply by hitting **F5** (on some systems, Fn + F5).
- Before running, IDLE prompts you to save the script as a file. Choose a name ending in .py ("hello.py") and save it.

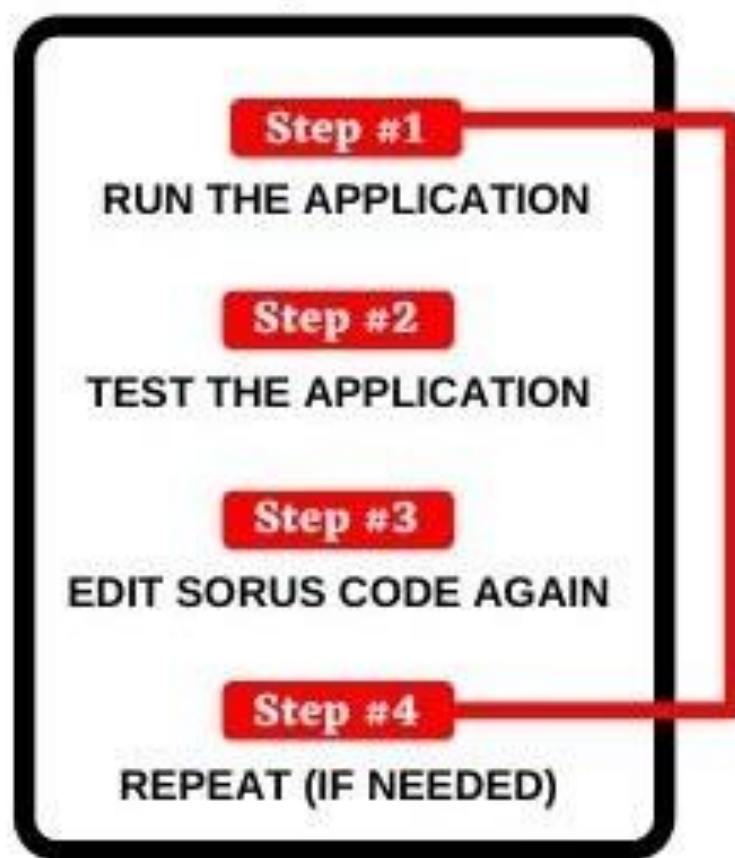
- The script will then run in the IDLE shell window. Since you now have a saved script, you can run it again (and again, and again...).
- You can also open IDLE directly from your Python script file. Right click the file, then choose "Edit with IDLE".
- Rather than going through the "Run..." menu, learn to use F5 (on some systems, Fn + F5) to run your script. It's much quicker.

Programming Cycle for Python

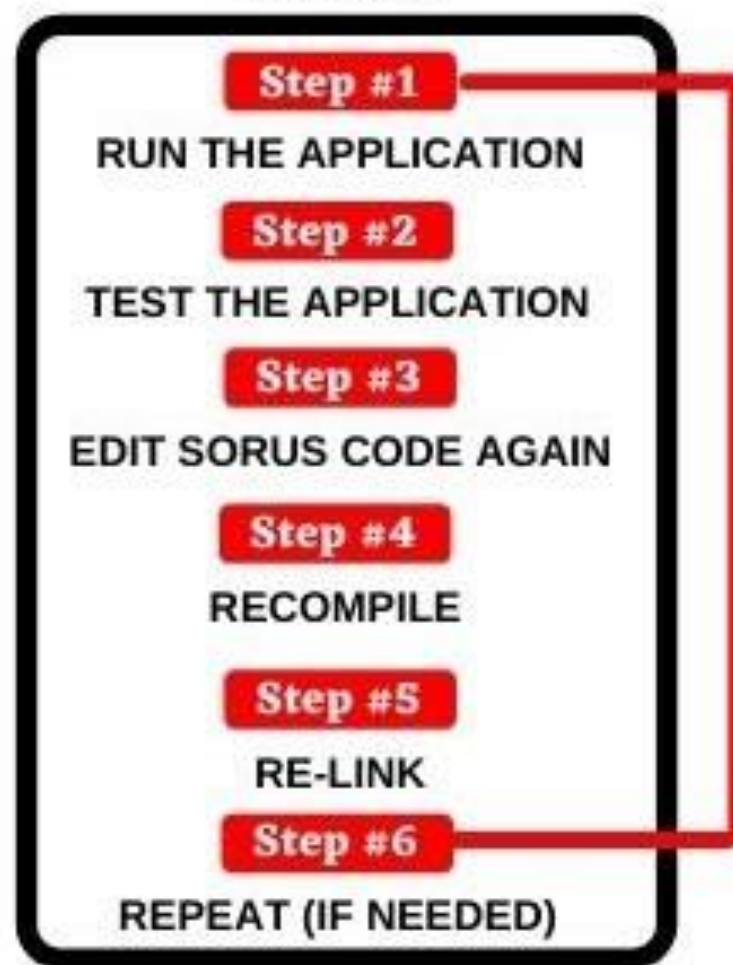
- The development cycle of Python is considerably shorter than that of traditional tools. There are no compilation or linking steps in Python.
- Python programs simply import modules at runtime and use the objects they contain. Because of this, Python programs run immediately after changes are made.
- Also because Python is interpreted, there's a rapid turnaround after program changes. And because Python's parser is embedded in Python-based systems, it's easy to modify programs at runtime.

PROGRAMMING CYCLE FOR PYTHON

Python



Other's



Python | Compiled or Interpreted ?

- In various books of python programming, it is mentioned that python language is interpreted. But that is half correct the python program is first compiled and then interpreted.
- The compilation part is hidden from the programmer. The compilation part is done first when we execute our code and this will generate byte code and internally this byte code gets converted by the python virtual machine(p.v.m) according to the underlying platform(machine+operating system).
- The compiled part is get deleted by the python(as soon as you execute your code) just it does not want programmers to get into complexity.

Interpreter

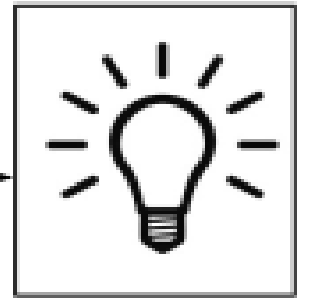


Source code

Compiler

Byte
code

Virtual
machine



Running code

`python -m first.py`
(Compile)

`python first.cpython-38.pyc`
(Run)

Library
modules

To manually check the compilation process-

first.py

```
print("i am learning python")  
print("i am enjoying it")
```

Let, “first.py” is in a folder named “pyprog” in D drive. Then compile using following-

```
D:\pyprog>python -m py_compile first.py
```

Or we can only write: “*python -m first.py*”

As you press enter the byte code will get generated. A folder created and this will contain the byte code of your program.

Now to **run** the compiled byte code just type the following command:

```
D:\pyprog\__pycache__>python first.cpython-37.pyc  
i am learning python  
i am enjoying it
```

the extension .pyc is python compiler.

Use -38.pyc (in place of -37.pyc) if python version is 3.8.

Python Syntax

print() Function:

- The print() function prints the specified message to the screen, or other standard output device. The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Syntax:

`print(object(s), sep=separator, end=end, file=file, flush=flush)`

- **object(s)**- Any object, and as many as you like. Will be converted to string before printed
- **sep**- Optional. Specify how to separate the objects, if there is more than one. Default is ' '
- **end**- Optional. Specify what to print at the end. Default is '\n' (line feed)
- **file**- Optional. An object with a write method. Default is sys.stdout
- **flush**- Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False.

EX:

```
print("Hello", "how are you?")           #Hello how are you?
print("Hello", "how are you?", sep="---") #Hello---how are you?
x = ("apple", "banana", "cherry")
print(x)                                 #('apple', 'banana', 'cherry')
```

```
a = 5
print("a =", a)                          # a = 5
# print("a =" + a)                       # ERROR
print("a =" + str(a))                    # a =5
b = a
print('a =', a, '= b')                   # a = 5 = b
```

```
a = 5
print("a =", a, sep='00000', end='\n\n\n')
print("a =", a, sep='0', end='')
```

O/P: a =000005
a =05

****** If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an ***r*** before the first quote:

```
print('C:\some\name')      # C:\some
                             ame
print(r'C:\some\name')     # C:\some\name
```

Python Indentation:

➤ Python uses indentation to indicate a block of code.

```
if True:
    print ("True")
else:
    print ("False")
```

O/P: True

- The number of spaces in the indentation is variable (but it has to be at least one), All statements within the block must be indented the same amount.

For example –

```
if True:
    print ("True")
    print("yes")
else:
    print ("False")
    print("no")
```

O/P: True
yes

However, the following block generates an error –

```
if True:  
print ("True")  
else:  
print ("False")
```

ERROR: "expected an indented block"

- You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

```
if 5 > 2:  
    print("Five is greater than two!")  
        print("Five is greater than two!")
```

ERROR: "unexpected indent"

```
if 5 > 2:  
    print("Five is greater than two!")  
print("Five is greater than two!")
```

O/P: Five is greater than two!
Five is greater than two!

Comments in Python:

- Comments start with a #, and Python will render the rest of the line as a comment:

EX:

```
# This is a comment.  
print("Hello, World!")
```

- Triple-quoted string is also ignored by Python interpreter and can be used as a multiline comment:

```
'''
```

```
This is a multiline  
comment.
```

```
'''
```

Multi-Line Statements:

- Statements in Python typically end with a new line. But python allow the use of line continuation character (\) to denote that the line should continue.

For example:

```
total = item_one + \  
item_two + \  
item_three
```

- Statements contained within the [], {}, or () brackets do not need to use the line continuation character.

For example:

```
days = [ 'Monday' , 'Tuesday' , 'Wednesday' ,  
         'Thursday' , 'Friday' ]
```

Multiple Statements on a Single Line:

- The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon –

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites:

- A group of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.
- Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example-

```
if expression:
```

```
    suite
```

```
elif expression:
```

```
    suite
```

```
else:
```

```
    suite
```

Quotation in Python:

- Python accepts single ('), double (") and triple ('' or ''') quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes are used to span the string across multiple lines. For example, all the following are legal –

```
word = 'word'
```

```
sentence = "This is a sentence."
```

```
paragraph = """This is a paragraph. It is  
made up of multiple lines and sentences."""
```

Python Identifiers:

- A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- Python does not allow punctuation characters such as @, \$, and % within identifiers. Python is a case sensitive programming language.

Here are naming conventions for Python identifiers –

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words:

- The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Python Variables

- Python variables do not need explicit declaration to reserve memory space.
- Variables do not need to be declared with any particular type and can even change type after they have been set.
- String variables can be declared either by using single or double quotes

```
counter =100          # An integer assignment
miles    =1000.0       # A floating point
name     ="John"       # A string
```

```
x = 4                 # x is of type int
x = "Sally"          # x is now of type str
print(x)
```

Example

#Legal variable names:

```
myvar = "John"  
my_var = "John"  
_my_var = "John"  
myVar = "John"  
MYVAR = "John"  
myvar2 = "John"
```

#Illegal variable names:

```
2myvar = "John"  
my-var = "John"  
my var = "John"
```

Python allows you to assign values to multiple variables in one line:

Ex: `x, y, z = "Abc", 20, "Xyz"`

Ex: `x = y = z = "India"`

To combine both text and a variable, Python uses the ‘+’ character:

Ex: `x = "Easy"`
 `print("Python is " , x)`

Ex: `x = "Python is "`
 `y = "easy"`
 `z = x + y`

*For numbers, the + character works as a mathematical operator.
If you try to combine a string and a number, Python will give you an error:*

Ex: `x = 5`
 `y = "John"`
 `print(x + y) # ERROR`

Global Variables:

- Variables that are created outside of a function (as in all of the examples above) are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

Ex:

```
x = "easy"
def myfunc():
    print("Python is " , x)
myfunc()
```

- If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function.
- The global variable with the same name will remain as it was, global and with the original value.

Ex:

```
x = "easy"
def myfunc():
    x = "programming language"
    print("Python is " , x)
myfunc()
print("Python is " , x)
```

O/P: Python is programming language
Python is easy

The “global” keyword:

- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
- To create a global variable inside a function, you can use the “global” keyword.

Ex:

```
def myfunc():  
    global x  
    x = "easy"  
  
myfunc()  
  
print("Python is " , x)
```


Also, use the **global** keyword if you want to change a global variable inside a function.

Ex:

```
x = "easy"

def myfunc():
    global x
    x = "programming language"

myfunc()

print("Python is " , x)

O/P: Python is programming language
```

Data Types in Python

Python has the following data types built-in by default, in these categories:

Text Type:	Str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	Dict (Dictionary)
Set Types:	set, frozenset
Boolean Type:	Bool
Binary Types:	bytes, bytearray, memoryview

You can get the data type of any object by using the **type()** function:

```
x = 5  
print(type(x))
```

O/P: <class 'int'>

```
x = "5"  
print(type(x))
```

O/P: <class 'str'>

```
x = 2.5
```

```
print(x)  
print(type(x))
```

O/P: 2.5
 <class 'float'>

```
x = b"Hello"  
print(x)  
print(type(x))
```

O/P: b'Hello'
 <class 'bytes'>

In Python, the data type is set when you assign a value to a variable

Example	Data Type
<code>x = "Hello World"</code>	str
<code>x = 20</code>	int
<code>x = 20.5</code>	float
<code>x = 1j</code>	complex
<code>x = ["apple", "banana", "cherry"]</code>	list
<code>x = ("apple", "banana", "cherry")</code>	tuple
<code>x = range(6)</code>	range
<code>x = {"name" : "John", "age" : 36}</code>	dict
<code>x = {"apple", "banana", "cherry"}</code>	set
<code>x = frozenset({"apple", "banana", "cherry"})</code>	frozenset
<code>x = True</code>	bool
<code>x = b"Hello"</code>	bytes
<code>x = bytearray(5)</code>	bytearray
<code>x = memoryview(bytes(5))</code>	memoryview

If you want to specify the data type, you can use the following constructor functions

Example	Data Type
<code>x = str("Hello World")</code>	str
<code>x = int(20)</code>	int
<code>x = float(20.5)</code>	float
<code>x = complex(1j)</code>	complex
<code>x = list(("apple", "banana", "cherry"))</code>	list
<code>x = tuple(("apple", "banana", "cherry"))</code>	tuple
<code>x = range(6)</code>	range
<code>x = dict(name="John", age=36)</code>	dict
<code>x = set(("apple", "banana", "cherry"))</code>	set
<code>x = frozenset(("apple", "banana", "cherry"))</code>	frozenset
<code>x = bool(5)</code>	bool
<code>x = bytes(5)</code>	bytes
<code>x = bytearray(5)</code>	bytearray
<code>x = memoryview(bytes(5))</code>	memoryview

Python Numbers:

There are three numeric types in Python:

1. **int** - Integer value can be any length such as integers 10, 2, 29, -20, -150 etc.
Python has no restriction on the length of an integer. Its value belongs to **int**
2. **float** - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc.
It is accurate upto 15 decimal points.
3. **complex** - A complex number contains an ordered pair, i.e., $x + iy$ where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

Ex.

```
x = 1      # int
y = 2.8    # float
z = 1j     # complex
```

** Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

```
a = 5
print("The type of a", type(a))
b = 40.5
print("The type of b", type(b))
c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```


O/P: The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is a complex number True

**** Float can also be scientific numbers with an "e" to indicate the power of 10.**

```
x = 35e3
y = 12E4
z = -87.7e100

print(x)          # 35000.0
print(y)          # 120000.0
print(z)          # -8.77e+101
```

****** Complex numbers are written with a "j or J" as the imaginary part.

```
x = 3+5j
y = 5j
z = -5j

print(x)          # (3+5j)
print(type(x))    #<class 'complex'>
print(y)          # 5j
print(type(y))    #<class 'complex'>
print(z)          # (-0-5j)
print(type(z))    #<class 'complex'>
```

****** You can convert from one type to another with the **int()**, **float()**, and **complex()** methods:

```
x = 1      # int
y = 2.8    # float
z = 1j     # complex
```

```
#convert from int to float:
```

```
a = float(x)
```

```
#convert from float to int:
```

```
b = int(y)
```

```
#convert from int to complex:
```

```
c = complex(x)
```

```
#convert from float to complex:
```

```
d = complex(y)
```

```
print(a)          # 1.0
print(b)          # 2
print(c)          # (1+0j)
print(d)          # (2.8+0j)

print(type(a))    # <class 'float'>
print(type(b))    # <class 'int'>
print(type(c))    # <class 'complex'>
print(type(d))    # <class 'complex'>
```

```
d=int(z)    #TypeError: can't convert complex to int
d=float(z)  # TypeError: can't convert complex to float
```

Random Number:

- Python has a built-in module called **random** that can be used to make random numbers.
- random module functions depend on function random(), which generates the float number between **0.0** and **1.0**

```
import random
print(random.random())          # Generate number between 0 and 1
print(random.randrange(1, 10))  # Generate number between 1 and 10
print(type(random.random()))    # <class 'float'>
print(type(random.randrange(1,10))) # <class 'int'>
```

The random module has a set of methods, below is list of some commonly used methods

Method	Description
randrange()	Returns a random number between the given range
randint()	Returns a random number between the given range
choice()	Returns a random element from the given sequence
choices()	Returns a list with a random selection from the given sequence
shuffle()	Takes a sequence and returns the sequence in a random order
sample()	Returns a given sample of a sequence

random()	Returns a random float number between 0.0 and 1.0
uniform()	Returns a random float number between two given parameters
seed()	Initialize the random number generator
getstate()	Returns the current internal state of the random number generator
setstate()	Restores the internal state of the random number generator
getrandbits()	Returns a number representing the random bits
triangular()	Returns a random float number between two given parameters, you can also set a mode parameter to specify the midpoint between the two other parameters

random.randrange(*start, stop, step*)

start- Optional. An integer specifying at which position to start. Default 0

stop- Required. An integer specifying at which position to end.

step- Optional. An integer specifying the incrementation. Default 1

random.getrandbits(*n*)

This method returns an integer formed with bit binary sequence. If *n* is 2, then it can generate 0, 1, 2 or 3.

random.sample(*sequence, k*)

This method returns a list with a randomly selection of a specified number of items (*k*) from a sequence.

random.triangular(*low, high, mode*)

This method returns a random floating number between the two specified numbers (both included), but you can also specify a third parameter, the mode parameter.

The ***mode*** parameter gives you the opportunity to weigh the possible outcome closer to one of the other two parameter values.

Ex:

```
import random
print(random.choice([50,41,84,40,31]))
# print from (50,41,84,40,31)

print(random.getrandbits(3))
# Print value b/w (0-7)

print(random.randrange(100, 500, 5))
# Print value b/w (100-500), value will be multiple of 5.

mylist = ["apple", "banana", "cherry"]
random.shuffle(mylist)
print(mylist)
#Print apple,banana,cherry in any order

print(random.sample(mylist, k=2))
#Print any 2 from apple,banana,cherry
```

Python Booleans:

- Boolean type provides two built-in values, True and False. It denotes by the class bool.
- When you compare two values, the expression is evaluated and Python returns the Boolean answer.

```
print(10 > 9)      # True
print(10 == 9)     # False
print(10 < 9)      # False
```

```
a = 200
```

```
b = 33
```

```
if b > a:
```

```
    print("b is greater than a")
```

```
else:
```

```
    print("b is not greater than a")
```

O/P: b is not greater than a

Most Values are True:

- Almost any value is evaluated to True if it has some sort of content.
- Any string is *True*, except empty strings.
- Any number is True, except 0.
- Any list, tuple, set, and dictionary are True, except empty ones.

**** bool() function allows you to evaluate any value, and return *True* or *False*.**

```
bool("abc") # True
bool(123)    # True
bool(["apple", "cherry", "banana"]) # True
```

Some Values are False:

In fact, there are not many values that evaluates to *False*, except:

- empty values, such as (), [], {}, "",
- the number 0, and the value None.
- And the value False evaluates to False.

```
bool(False)      # False
bool(None)       # False
bool(0)          # False
bool("")         # False
bool(())         # False
bool([])         # False
bool({})         # False
```

Python Casting

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1. Implicit Type Conversion
2. Explicit Type Conversion

Implicit Type Conversion:

- In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.
- Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

Example: Converting integer to float

```
num_int = 123
num_flo = 1.23

num_new = num_int + num_flo

print("datatype of um_int:", type(num_int))
print("datatype of um_flo:", type(num_flo))

print("Value of num_new:", num_new)
print("datatype of um_new:", type(num_new))
```

When we run the above program, the output will be:

```
datatype of num_int: <class 'int'>  
datatype of num_flo: <class 'float'>
```

```
Value of num_new: 124.23
```

```
datatype of num_new: <class 'float'>
```

Explicit Type Conversion:

- In Explicit Type Conversion, users convert the data type of an object to required data type.
- We use the predefined functions (Constructors) like `int()`, `float()`, `str()`, etc to perform explicit type conversion.
- This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Example 3: Addition of string and integer using explicit conversion

```
num_int = 123
num_str = "456"

print("Data type of num_int:", type(num_int))
print("Data type of num_str before Type Casting:", type(num_str))

num_str = int(num_str)
print("Data type of num_str after Type Casting:", type(num_str))

num_sum = num_int + num_str

print("Sum of num_int and num_str:", num_sum)
print("Data type of the sum:", type(num_sum))
```

When we run the above program, the output will be:

```
Data type of num_int: <class 'int'>
```

```
Data type of num_str before Type Casting: <class 'str'>
```

```
Data type of num_str after Type Casting: <class 'int'>
```

```
Sum of num_int and num_str: 579
```

```
Data type of the sum: <class 'int'>
```

Casting in python is therefore done using constructor functions:

- **int()** - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
- **float()** - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- **str()** - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Ex:

```
x = int(1)          # x will be 1
y = int(2.8)        # y will be 2
z = int("3")        # z will be 3

a = float(1)        # a will be 1.0
b = float(2.8)      # b will be 2.8
c = float("3")      # c will be 3.0
d = float("4.2")    # d will be 4.2

p = str("s1")       # p will be 's1'
q = str(2)          # q will be '2'
r = str(3.0)        # r will be '3.0'
```

Expressions in Python

- An expression is a combination of **operators** and **operands** that is interpreted to produce some other value.
- In any programming language, an expression is evaluated as per the precedence of its **operators**. So that if there is more than one operator in an expression, their precedence decides which operation will be performed first.
- We have many different types of expressions involving different operators in Python.

Operators:

➤ Operators are used to perform operations on variables and values. Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

➤ Arithmetic Operators:

Operator	Description
+ (Addition)	It is used to add two operands. For example, if $a = 20$, $b = 10 \Rightarrow a + b = 30$
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if $a = 20$, $b = 10 \Rightarrow a - b = 10$
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if $a = 20$, $b = 10 \Rightarrow a / b = 2.0$
* (Multiplication)	It is used to multiply one operand with the other. For example, if $a = 20$, $b = 10 \Rightarrow a * b = 200$
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if $a = 20$, $b = 10 \Rightarrow a \% b = 0$
** (Exponent)	It is an exponent operator represented as it calculates the first operand power to the second operand.
// (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

➤ Comparison Operators: (return True or False)

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal, then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

➤ Assignment Operators:

Operator	Example	Same As
=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 3</code>	<code>x = x + 3</code>
-=	<code>x -= 3</code>	<code>x = x - 3</code>
*=	<code>x *= 3</code>	<code>x = x * 3</code>
/=	<code>x /= 3</code>	<code>x = x / 3</code>
%=	<code>x %= 3</code>	<code>x = x % 3</code>
//=	<code>x //= 3</code>	<code>x = x // 3</code>
**=	<code>x **= 3</code>	<code>x = x ** 3</code>
&=	<code>x &= 3</code>	<code>x = x & 3</code>
 =	<code>x = 3</code>	<code>x = x 3</code>
^=	<code>x ^= 3</code>	<code>x = x ^ 3</code>
>>=	<code>x >>= 3</code>	<code>x = x >> 3</code>
<<=	<code>x <<= 3</code>	<code>x = x << 3</code>

➤ **Logical Operators:(used to combine conditional statements)**

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

➤ Identity Operators:

Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Ex:

```
x = ["apple", "banana"]  
y = ["apple", "banana"]  
z = x  
  
print(x is z)           # True  
print(x is y)           # False  
print(x == y)           # True
```

**** (x==y) this comparison returns True because content of x and y are equal.**

Ex: To demonstrate the difference between "is" and "==":

```
a = 2  
b = 2  
print( a is b)           # True
```

```
a = "abc"  
b = "abc"  
print( a is b)           # True
```



```
a = [1,2,3]  
b = [1,2,3]  
print( a is b)           # False
```

```
a=5  
print (type(a) is int)    # True  
print (type(a)==int)      # True
```

➤ Membership Operators

Used to check the membership of value inside a Python data structure(list, tuple, or dictionary). It tests that a sequence is presented in an object or not.

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Ex:

```
x = ["Java", "Python"]
```

```
print("Python" in x)      # True
```

```
print("PHP" in x)        # False
```

➤ Bitwise Operators

Used to compare (binary) numbers, it performs bit by bit operation on the values of the two operands.

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
 (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different; otherwise, the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Bitwise AND:

Returns 1 if both the bits are 1 else 0.

```
a = 10 = 1010 (Binary)
```

```
b = 4 = 0100 (Binary)
```

```
a & b = 1010
```

```
&
```

```
0100
```

```
= 0000
```

```
= 0 (Decimal)
```

Bitwise OR:

Returns 1 if either of the bit is 1 else 0.

```
a = 10 = 1010 (Binary)
```

```
b = 4 = 0100 (Binary)
```

```
a | b = 1010
```

```
|
```

```
0100
```

```
= 1110
```

```
= 14 (Decimal)
```

Bitwise NOT:

Returns one's complement of the number.

```
a = 10 = 1010 (Binary)
```

```
~a = ~1010
```

```
= -(1010 + 1)
```

```
= -(1011)
```

```
= -11 (Decimal)    # for value 'x' it returns '-(x+1)'
```

Bitwise XOR:

Returns 1 if both bits are different else returns 0.

```
a = 10 = 1010 (Binary)
```

```
b = 4 = 0100 (Binary)
```

```
a & b = 1010
```

```
^
```

```
0100
```

```
= 1110
```

```
= 14 (Decimal)
```


Bitwise Right Shift:

Shifts the bits of the number to the right and fills 0 on left as a result. Similar effect as of dividing the number with some power of two.

```
a = 10
```

```
a >> 1 = 5
```

```
b = -10
```

```
b >> 1 = -5
```

Bitwise left shift:

Shifts the bits of the number to the left and fills 0 on left as a result. Similar effect as of multiplying the number with some power of two.

```
a = 5
```

```
b = -10
```

```
a << 1 = 10
```

```
a << 2 = 20
```

```
b << 1 = -20
```

```
b << 2 = -40
```

Ex:

a = 12 **# 1100**

b = 6 **# 0110**

print(a & b) **# 4**

print(a | b) **# 14**

print(~a) **# -13**

print(a ^ b) **# 10**

print(a >> 2) **# 3**

print(a << 2) **# 48**

Multiple operators in expression (Operator Precedence):

- If there are more than one operator in an expression, it may give different results on basis of the order of operators executed.
- To sort out these confusions, the operator precedence is defined. Operator Precedence simply defines the priority of operators that which operator is to be executed first.
- Here we see the operator precedence in Python, where the operator higher in the list has more precedence or priority:

<u>Precedence</u>	<u>Name</u>	<u>Operator</u>
1	Parenthesis	() [] {}
2	Exponentiation	**
3	Unary plus or minus, complement	-a , +a , ~a
4	Multiply, Divide, Modulo	/ * // %
5	Addition & Subtraction	+ -
6	Shift Operators	>> <<
7	Bitwise AND	&

8	Bitwise XOR	\wedge
9	Bitwise OR	
10	Comparison Operators	$\geq \leq > <$
11	Equality Operators	$== !=$
12	Assignment Operators	$= += -= /= *=$
13	Identity and membership operators	is, is not, in, not in
14	Logical Operators	and, or, not

Ex:

```
a = 10 + 3 * 4
```

```
print(a)
```

```
b = (10 + 3) * 4
```

```
print(b)
```

```
c = 10 + (3 * 4)
```

```
print(c)
```

O/P- 22

52

22

Object References

- What is actually happening when you make a variable assignment? This is an important question in Python, because the answer differs somewhat from what you'd find in many other programming languages.
- Python is a highly object-oriented language. In fact, virtually every item of data in a Python program is an object of a specific type or class.

Consider this code:

```
>>>print(300)  
300
```

When presented with the statement `print(300)`, the interpreter does the following:

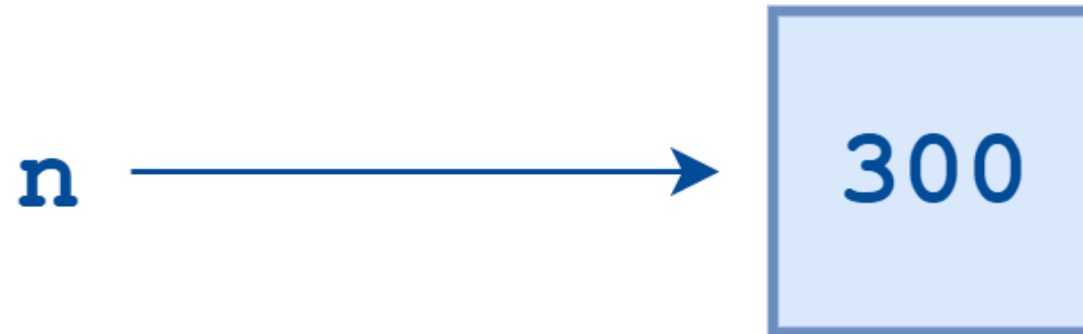
- Creates an integer object
- Gives it the value 300
- Displays it to the console

A Python variable is a symbolic name that is a reference or pointer to an object. Once an object is assigned to a variable, you can refer to the object by that name. But the data itself is still contained within the object.

For example:

```
>>>n=300
```

This assignment creates an integer object with the value 300 and assigns the variable `n` to point to that object.



(Variable Assignment)

Now consider the following statement:

```
>>>m=n
```

What happens when it is executed? Python does not create another object. It simply creates a new symbolic name or reference, *m*, which points to the same object that *n* points to.

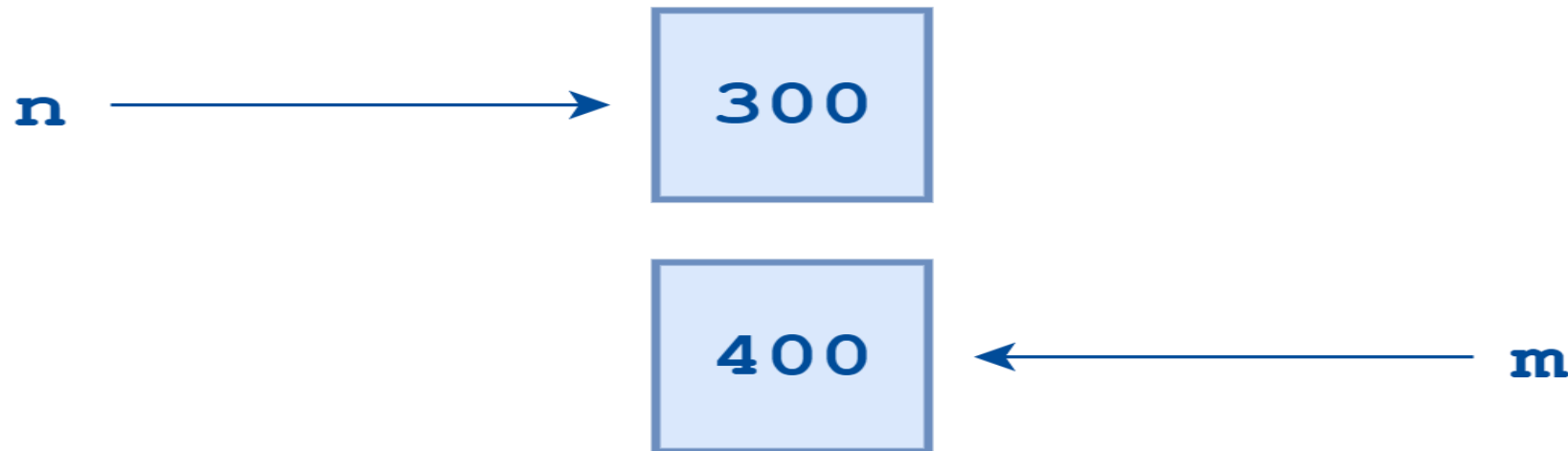


(Multiple References to a Single Object)

Next, suppose you do this:

```
>>>m=400
```

Now Python creates a new integer object with the value 400, and m becomes a reference to it.

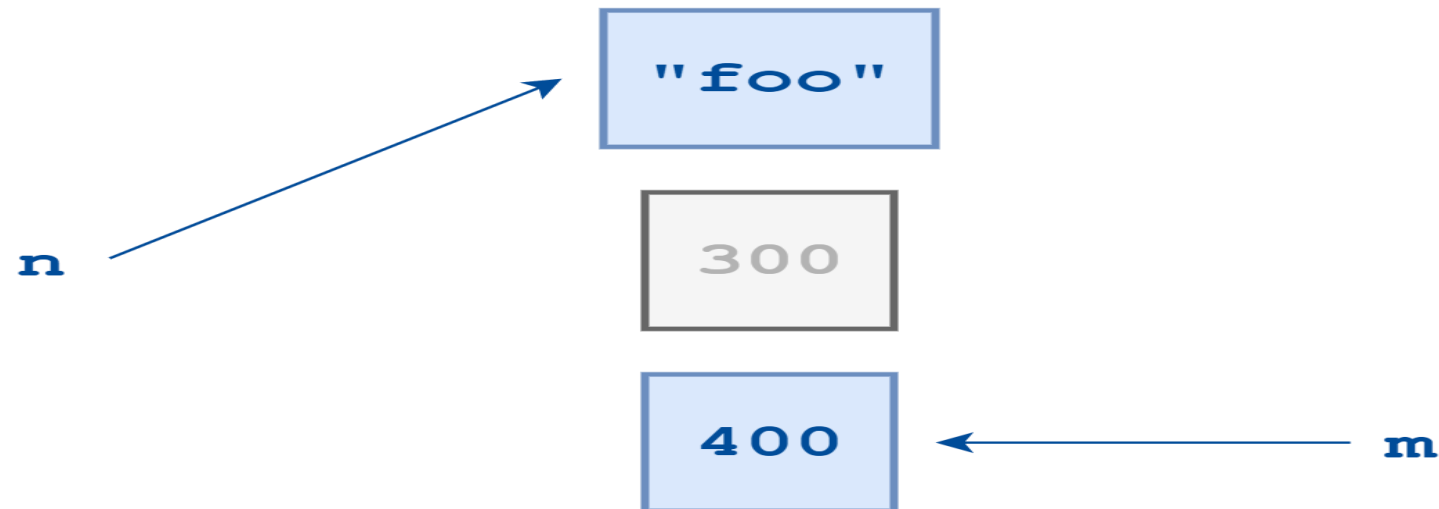


(References to Separate Objects)

Lastly, suppose this statement is executed next:

```
>>>n="foo"
```

Now Python creates a string object with the value "foo" and makes 'n' reference that. There is no longer any reference to the integer object 300. It is orphaned, and there is no way to access it.



(Orphaned Object)

Object Identity

- In Python, every object that is created is given a number that uniquely identifies it. It is guaranteed that no two objects will have the same identifier during any period in which their lifetimes overlap.
- Once an object's reference count drops to zero and it is garbage collected, as happened to the 300 object above, then its identifying number becomes available and may be used again.
- The built-in Python function `id()` returns an object's integer identifier. Using the `id()` function, you can verify that two variables indeed point to the same object:

```
>>>n=300
```

```
>>>m=n
```

```
>>>id(n)
```

```
60127840
```

```
>>>id(m)
```

```
60127840
```

```
>>>m=400
```

```
>>>id(m)
```

```
60127872
```

Ex:

```
a = 3  
b = 3  
c = a  
print(a)           # 3  
print(b)           # 3  
print(c)           # 3  
print(id(a))        # 1360242640  
print(id(b))        # 1360242640  
print(id(c))        # 1360242640  
a=4  
print(a)           # 4  
print(b)           # 3  
print(c)           # 3  
print(id(a))        # 1360242656  
print(id(b))        # 1360242640  
print(id(c))        # 1360242640
```


User Input

➤ Python provides two built-in methods to read the data from the keyboard.

These methods are given below.

- `input(prompt)`
- `raw_input(prompt)`

input():

- The input function is used in all latest version of the Python. It takes the input from the user. The Python interpreter automatically identifies whether a user input a string, a number, or a list.

```
name = input("Enter your name: ")  
print(name)
```

How **input()** function works?

- The flow of the program has stopped until the user enters the input.
- The text statement which also knows as prompt is optional to write in **input()** function. This prompt will display the message on the console.
- The **input()** function automatically converts the user input into string. We need to explicitly convert the input using the type casting.
- The **raw_input()** function is used in Python's older version like Python 2.x.

```
name = raw_input("Enter your name : ")  
print name
```

- By default, the input() function takes input as a **string** so if we need to enter the integer or float type input then the input() function must be type casted.

```
name = input("Enter your name: ")           # String Input
age = int(input("Enter your age: "))         # Integer Input
marks = float(input("Enter your marks: "))   # Float Input

print("The name is:", name)
print("The age is:", age)
print("The marks is:", marks)
```