Fundamentals of Python Programming

Python is a high-level, interpreted scripting language developed in the late 1980s by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in the Netherlands. The initial version was published at the alt.sources newsgroup in 1991, and version 1.0 was released in 1994.

Python 2.0 was released in 2000, and the 2.x versions were the prevalent releases until December 2008. At that time, the development team made the decision to release version 3.0, which contained a few relatively small but significant changes that were not backwards compatible with the 2.x versions. Python 2 and 3 are very similar, and some features of Python 3 have been backported to Python 2. But in general, they remain not quite compatible.

Both Python 2 and 3 have continued to be maintained and developed, with periodic release updates for both. As of this writing, the most recent versions available are 2.7.15 and 3.6.5. However, an o icial End Of Life date of January 1, 2020, has been established for Python 2, a er which time it will no longer be maintained. If you are a newcomer to Python, it is recommended that you focus on Python 3, as this tutorial will do.

Python is still maintained by a core development team at the Institute, and Guido is still in charge, having been given the title of BDFL (Benevolent Dictator For Life) by the Python community. The name Python, by the way, derives not from the snake, but from the British comedy troupe Monty Python's Flying Circus, of which Guido was, and presumably still is, a fan. It is common to find references to Monty Python sketches and movies scattered throughout the Python documentation.

❑ Python is Popular

Python has been growing in popularity over the last few years. The 2018 Stack Overflow Developer Survey ranked Python as the 7th most popular and the number one most wanted technology of the year. World-class software development countries around the globe use Python every single day.

❑ Python is Interpreted

Many languages are compiled, meaning the source code you create needs to be translated into machine code, the language of your computer's processor before it can be run. Programs written in an interpreted language are passed straight to an interpreter that runs them directly.

This makes for a quicker development cycle because you just type in your code and run it, without the intermediate compilation step.

One potential downside to interpreted languages is execution speed. Programs that are compiled into the native language of the computer processor tend to run more quickly than interpreted programs. For some applications that are particularly computationally intensive, like graphics processing or intense number crunching, this can be limiting.

In practice, however, for most programs, the di erence in execution speed is measured in milliseconds, or seconds at most, and not appreciably noticeable to a human user. The expediency of coding in an interpreted language is typically worth it for most applications.

❑ Python is Free

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

We generally write a computer program using a high-level language. A high-level language is one which is understandable by us humans. It contains words and phrases from the English (or other) language. But a computer does not understand high-level language. It only understands program written in <code>0's</code> and <code>1's</code> in binary, called the machine code. A program written in the high-level language is called a source code. We need to convert the source code into machine code and this is accomplished by compilers and interpreters. Hence, a compiler or an interpreter is a program that converts a program written in a high-level language into machine code understood by the computer.

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes a large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence is memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
A programming language like Python, Ruby use interpreters.	A programming language like C, C++ use compilers.

Python on Different Operating Systems

Python is a cross-platform programming language, which means it runs on all the major operating systems. Any Python program you write should run on any modern computer that has Python installed. However, the methods for setting up Python on di erent operating systems vary slightly.

Python Editors –

- IDLE
- Jupyter notebook
- Geany – ubuntu
- Sublime text – OSX

An Introduction to PEP-8

The Python programming language has evolved over the past year as one of the most favourite programming languages. This language is relatively easy to learn than most of the programming languages. It is a multi-paradigm, it has lots of open source modules that add up the utility of the language and it is gaining popularity in data science and web development community.

However, you can use the benefits of Python only when you know how to express better with your code. Python was made with some goals in mind, these goals can be seen when you type "import this".

The Zen of Python

The Python community's philosophy is contained in "The Zen of Python" by Tim Peters. You can access this brief set of principles for writing good Python code by entering import this into your interpreter.

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
```

```
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

The above are the 20 principles that Python programming uses. You also see "Readability Counts" in the output above, which should be your main concern while writing code: other programmers or data scientists should understand and should be able to contribute to your code so that it can solve the task at hand.

Hello World!

A long-held belief in the programming world has been that printing a Hello world! message to the screen as your first program in a new language will bring you luck.

In Python, you can write the Hello World program in one line:

```
print("Hello world!")
```

Such a simple program serves a very real purpose. If it runs correctly on your system, any Python program you write should work as well.

Example:

```
print("Hello World!!!")
```

```
Hello World!!!
```

Defining a variable

- A variable is a name that refers to a value. The assignment statement creates new variables and gives them values.
- Variables are reserved memory locations to store values. When a variable is created, some space in memory is reserved for the variable based on the datatype of a variable.
- Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.
- The operand to the le of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

Example:

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

- A Python identifier is a name used to identify a variable, function, class, module or another 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. Thus, Manpower and manpower are two different identifiers in Python.

Naming a variable

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable message_1 but not 1_message.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, greeting_message works, but greeting message will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word print.
- Variable names should be short but descriptive. For example, name is better than n, student_name is better than s_n, and name_length is better than length_of_persons_name.
- Be careful when using the lowercase letter l and the uppercase letter O because they could be confused with the numbers 1 and 0.

Python naming conventions

- 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.

Example:

```
x = 10 # Creating a variable

print("Value of x is :", x) # Print the variable
```

```
Value of x is : 10
```

Taking user input - input()

Example:

```
x = input()

print("value of x = ", x)
```

```
10
value of x =  10
```

Example:

```
# input() with a prompt
x = input("Enter a value : ")
print("value of x = ", x)
```

```
Enter a value : 10
value of x =  10
```

type() - tells the data type of value

Example:

```
x = 10
print("type of x = ",type(x))
```

```
type of x =  <class 'int'>
```

Example:

```
x = 10.5
print(type(x))
```

```
<class 'float'>
```

Example:

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

```
<class 'str'>
```

Example:

```
x = False
print(type(x))
```

```
<class 'bool'>
```

Operators in Python

1. Arithmetic Operators

Operator	Description	Example (Let a=10, b=7)
+	Addition: Adds values on either side of the operator.	a + b = 17
-	Subtraction: Subtracts right hand operand from left hand operand.	a – b = 3
*	Multiplication: Multiplies values on either side of the operator	a * b = 70
/	Division: Divides left hand operand by right hand operand (Returns a float value)	b / a = 1.428
//	Floor Division: Divides left hand operand by right hand operand (Returns a int value)	b // a = 1
%	Modulus: Divides left hand operand by right hand operand and returns remainder	b % a = 3
**	Exponent: Performs exponential (power) calculation on operators	a**b =10000000

2. Comparison Operators

Operator	Description	Example (Let a=10, b=7)
==	If the values of two operands are equal, then the condition becomes true.	(a == b) -> False
!=	If values of two operands are not equal, then condition becomes true.	(a != b) -> True
>	If the value of left operand is greater than the value of right	(a > b) -> True
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) -> False
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) □ True
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) □ False

3. Logical Operator

Operator	Description	Example (let a=65 and color="Red")
and	Logical AND: If both the operands are true then condition becomes true.	(a<=75 **and** a>=60) -> True
or	Logical OR: If any of the two operands are non-zero then condition becomes true.	(color == "Red" **or** color == "Blue") -> True

4. Bitwise Operator

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

5. Membership Operator

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	(“m” **in** “Mumbai”) à True
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	(“m” **not in** “Mumbai”) à False (“x” **not in** “Mumbai”) à True

6. Identity Operator

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Operator precedence

	Operator	Description
Lowest precedence	or	Boolean OR
	and	Boolean AND
	not	Boolean NOT
	==, !=, <, <=, >, >=, is, is not	comparisons, identity
	|	bitwise OR
	^	bitwise XOR
	&	bitwise AND
	<<, >>	bit shis
	+, -	addition, subtraction
	*, /, //, %	multiplication, division, floor division, modulo
	+x, -x, ~x	unary positive, unary negation, bitwise, negation
Highest precedence	**	exponentiation

Example: WAP for Addition of 2 numbers taken from user

```
a = input("enter value for a : ")
b = input("enter value for b : ")
c = a + b

print("Addition of ",a,"and",b,"=",c)
```

```
enter value for a : 2
enter value for b : 5
Addition of  2 and 5 = 25
```

```
print(type(a),type(b))
```

```
<class 'str'> <class 'str'>
```

Type Conversion

Example:

```
# Convert to <int>
a = int("2")
print(type(a),a)
```

```
<class 'int'> 2
```

Example:

```
# Modifying previous code for addition of 2 numbers
a = int(input("enter value for a : "))
b = int(input("enter value for b : "))
c = a + b

print("Addition of ",a,"and",b,"=",c)
```

```
enter value for a : 2
enter value for b : 5
Addition of  2 and 5 = 7
```

Conversion to <int>

Example:

```
# Valid conversions

a = int("2")  # str to int
print(type(a),"value of a : ", a)

a = int(2.7)  # float to int
print(type(a),"value of a : ", a)

a = int(True) # bool to int
print(type(a),"value of a : ", a)
```

```
<class 'int'> value of a :  2
<class 'int'> value of a :  2
<class 'int'> value of a :  1
```

Example:

```
# Invalid Conversions

a = int("2.5")
print(type(a),"value of a : ", a)  # - ValueError: invalid literal for int() with base 10:

a = int("abc")
print(type(a),"value of a : ", a)  # - ValueError: invalid literal for int() with base 10:
```

Conversion to <float>

Example:

```
# Valid cases

a = float(10) # int to float
print(type(a),"value of a : ", a)

a = float("10") # str to float
print(type(a),"value of a : ", a)

a = float("10.5") # str to float
print(type(a),"value of a : ", a)

a = float(False) # bool to float
print(type(a),"value of a : ", a)
```

```
<class 'float'> value of a :  10.0
<class 'float'> value of a :  10.0
<class 'float'> value of a :  10.5
<class 'float'> value of a :  0.0
```

Example:

```
# Invalid cases

a = float("abc")

print(type(a),"value of a : ", a)  # - ValueError: could not convert string to float: 'abc'
```

Conversion to <str>

Example:

```
a = str(10)
print(type(a),"value of a : ", a)
```

```
<class 'str'> value of a :  10
```

Conversion to <bool>

Example:

```
a = bool(10)   # int to bool
print(type(a),"value  of a : ", a)

a = bool(0)    # int to bool
print(type(a),"value  of a : ", a)

a = bool(10.2)  # float to bool
print(type(a),"value of a : ", a)

a = bool(0.0)  # float to bool
print(type(a),"value of a : ", a)

a = bool("abc")  # str to bool
print(type(a),"value of a : ", a)

a = bool("False") # str to bool
print(type(a),"value of a : ", a)
```

```
<class 'bool'> value of a :  True
<class 'bool'> value of a :  False
<class 'bool'> value of a :  True
<class 'bool'> value of a :  False
<class 'bool'> value of a :  True
<class 'bool'> value of a :  True
```

Importing a module (3-ways)

Example:

```
import math

math.sqrt(8)
```

```
2.8284271247461903
```

Example:

```
import math as m
m.sqrt(8)
```

```
2.8284271247461903
```

Example:

```
from math import *

sqrt(8)
```

```
2.8284271247461903
```

Working with random module

Example: Write a Python program to generate a random number between 1 to 10

```
import random as r

random_number = r.randint(1,10)

print(random_number)
```

```
3
```

Working with calendar module

Example: Write a Python program to print the calendar of a given month and year.

```
import calendar

y = int(input("Input the year : "))
m = int(input("Input the month in numbers : "))
print(calendar.month(y, m))
```

```
Input the year : 2018
Input the month in numbers : 10
    October 2018
 Mo Tu We Th Fr Sa Su
  1  2  3  4  5  6  7
  8  9 10 11 12 13 14
 15 16 17 18 19 20 21
 22 23 24 25 26 27 28
 29 30 31
```

Example: Write a Python script to display the -

1. Current date and time
2. Current year
3. Month of year
4. Week number of the year
5. Weekday of the week
6. Day of year
7. Day of the month
8. Day of week
9. Week number

```
import time
import datetime
print("Current date and time: " , datetime.datetime.now())
print("Current year: ", datetime.date.today().strftime("%Y"))
print("Month of year: ", datetime.date.today().strftime("%B"))
print("Week number of the year: ", datetime.date.today().strftime("%W"))
print("Weekday of the week: ", datetime.date.today().strftime("%w"))
print("Day of year: ", datetime.date.today().strftime("%j"))
print("Day of the month : ", datetime.date.today().strftime("%d"))
print("Day of week: ", datetime.date.today().strftime("%A"))
print(datetime.date(2015, 6, 16).isocalendar()[1])
```

```
Current date and time:   2018-10-28 10:59:44.637636
Current year:  2018
Month of year:  October
Week number of the year:  43
Weekday of the week:  0
Day of year:  301
Day of the month :  28
Day of week:  Sunday
25
```

Control Structures

Programming often involves examining a set of conditions and deciding which action to take based on those conditions. Python's if statement allows you to examine the current state of a program and respond appropriately to that state.

In the real world, we commonly must evaluate information around us and then choose one course of action or another based on what we observe:

If the weather is nice, then I'll mow the lawn. (It's implied that if the weather isn't nice, then I won't mow the lawn.)

In a Python program, the if statement is how you perform this sort of decision-making. It allows for **conditional** execution of a statement or group of statements based on the value of an expression.

Examples on Decision making

Example: WAP to check entered number is even or odd

```
x = int(input("Enter a no: "))
if (x % 2) == 0:
    print(x, " is an even no")
else:
    print(x, " is an odd no")
```

```
Enter a no: 4
4  is an even no
```

Example: Grade system example

```
marks = int(input("Enter your marks:"))
if (marks >= 75):
    print("You are passed in distinction")
elif (marks < 75 and marks >= 60):  # use and or &
    print("You are passed in First Class")
elif (marks < 60 and marks >= 40):
    print("You are passed in Second class")
else:
    print("You are failed in your exam")
```

```
Enter your marks:78
You are passed in distinction
```

Examples on iteration

Example: Find number of digits in a number and then find sum of digits of a number

```
num = int(input("Enter a number"))
sum = 0
count = 0

while num > 0:
    count += 1
    digit = num % 10
    sum += digit
    num = num // 10

print("No. of digits =", count, "and sum of digits=", sum)
```

```
Enter a number1234
No. of digits = 4 and sum of digits= 10
```

Example: WAP that keeps accepting character from user till user enters 'q'

```
while True:
    ch = input("enter a character")
    if ch == 'q':
        break
print("end of program!")
```

```
enter a charactera
enter a characters
enter a characterq
end of program!
```

Example: WAP that prints all number from 1 to 50 skipping multipes of 3

```
x = 0
while x <= 50:
    x += 1
    if x % 3:
        print(x, end=" ")
    else:
        continue
```

```
1 2 4 5 7 8 10 11 13 14 16 17 19 20 22 23 25 26 28 29 31 32 34 35 37 38 40 4
1 43 44 46 47 49 50
```

range()

range() constructor has two forms of definition:

```
range(stop)range(start, stop[, step])
```

range() Parameters

range() takes mainly three arguments having the same use in both definitions:

- **start** - integer starting from which the sequence of integers is to be returned
- **stop** - integer before which the sequence of integers is to be returned.
  The range of integers end at **stop - 1**.
- **step (Optional)** - integer value which determines the increment between each integer in the sequence

Return value from range()

range() returns an immutable sequence object of numbers depending upon the definitions used:

range(stop)

- Returns a sequence of numbers starting from **0** to **stop - 1**
- Returns an empty sequence if **stop** is **negative or 0**.

range(start, stop[, step])

The return value is calculated by the following formula with the given constraints:

r[n] = start + step*n (for both positive and negative step)where, n >=0 and r[n] < stop (for positive step)where, n >= 0 and r[n] > stop (for negative step)

- (If no **step**) Step defaults to 1. Returns a sequence of numbers starting from **start** and ending at **stop - 1**.
- (if **step** is zero) Raises a **ValueError** exception
- (if step is non-zero) Checks if the **value constraint** is met and returns a sequence according to the formula
  If it doesn't meet the value constraint, **Empty** sequence is returned.

Example:

```
# Examples on range()
for i in range(10): # default start = 0, end = 9, default step = 1
    print(i, end= " ")
print()

for i in range(1, 10): # start = 1, end = 9, default step = 1
    print(i, end= " ")
print()

for i in range(1, 10, 2): # start = 1, end = 9, step = 2
    print(i, end= " ")
print()

# Reverse numbering
for i in range(10, 0, -1): # start = 10, end = 1, step = -1
    print(i, end= " ")
```

```
 0 1 2  3 4 5 6 7 8  9
 1 2 3  4 5 6 7 8 9
 1 3 5  7 9
10  9 8 7 6 5 4 3 2  1
```

Example: Find factorial of a number

In [39]:

```
n=int(input("Enter a number : "))
fact=1
for i in range(1,n+1,1):
    fact *= i
print ("Factorial of",n,"is",fact)
```

```
Enter a number : 5

Factorial of 5 is 120
```