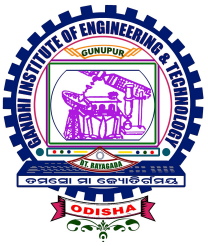
Fundamentals of Python Programming

Ranjit Patnaik, **Assistant Professor**
K. Murali Gopal, **Associate Professor**
Murali Krishna Senapaty, **Assistant Professor**
G.V.S.Narayana, **Assistant Professor**
Department Of Computer Science and Engineering,
GIET MAIN CAMPUS AUTONOMOUS, GUNUPUR

January 3, 2020



Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION TO PYTHON

1.1 Introduction and Installation of Python

Python is a widely used general-purpose, high level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

Finding an Interpreter: Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like ***https://ide.geeksforgeeks.org/, http://ideone.com/ or http://codepad.org/*** that can be used to start Python without installing an interpreter.

Windows:There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from ***http://python.org/***.

Linux:For Linux, Python comes bundled with the linux.

1.2 Installation:

1. **Windows Installation:** In some circumstances, your Windows environment may require administrator privilege. The details are beyond the scope of this book. If you can install software on your PC, then you have administrator privileges. In a corporate or academic environment, someone else

may be the administrator for your PC.
The Windows installation of Python has three broad steps.

(a) Pre-installation: make backups and download the installation kit.

(b) Installation: install Python.

(c) Post-installation: check to be sure everything worked

2. **Macintosh Installation:** Python is part of the MacOS environment. Tiger (Mac OS 10.4) includes Python 2.3.5 and IDLE. Leopard (Mac OS 10.5) includes Python 2.5.1. Snow Leopard (Mac OS 10.6) includes Python 2.6. Generally, you don't need to do much to get started. You'll just need to locate the various Python files. Look in /System/Library/Frameworks/Python.Framework/Versions for the relevant files. In order to upgrade software in the Macintosh OS, you must know the administrator, or "owner" password. If you are the person who installed or initially setup the computer, you had to pick an owner password during the installation. If someone else did the installation, you'll need to get the password from them.
A Mac OS upgrade of Python has three broad steps.

(a) Pre-upgrade: make backups and download the installation kit.

(b) Installation: upgrade Python.

(c) Post-installation: check to be sure everything worked.

3. **GNU/Linux and UNIX Overview:** In Checking for Python we'll provide a procedure for examining your current configuration to see if you have Python in the first place. If you have Python, and it's version 2.6, you're all done. Otherwise, you'll have to determine what tools you have for doing an installation or upgrade.

(a) If you have Yellowdog Updater Modified (YUM)

(b) If you have one of the GNU/Linux variants that uses the Red Hat Package Manager (RPM)

1.3 How Python differs from other programming Languages.

Dynamic vs Static Types : Python is a dynamic-typed language. Many other languages are static typed, such as C/C++ and Java. A static typed language

requires the programmer to explicitly tell the computer what type of "thing" each data value is. For example, in C if you had a variable that was to contain the price of something, you would have to declare the variable as a "float" type. This tells the compiler that the only data that can be used for that variable must be a floating point number, i.e. a number with a decimal point. If any other data value was assigned to that variable, the compiler would give an error when trying to compile the program.

Interpreted vs. Compiled : Many "traditional" languages are compiled, meaning the source code the developer writes is converted into machine language by the compiler. Compiled languages are usually used for low-level programming (such as device drivers and other hardware interaction) and faster processing, e.g. video games. Because the language is pre-converted to machine code, it can be processed by the computer much quicker because the compiler has already checked the code for errors and other issues that can cause the program to fail. The compiler won't catch all errors but it does help. The caveat to using a compiler is that compiling can be a time consuming task; the actual compiling time can take several minutes to hours to complete depending on the program. If errors are found, the developer has to find and fix them then rerun the compiler; this cycle continues until the program works correctly.

Prototyping : Because of interpretation, Python and similar languages are used for rapid application development and program prototyping. For example, a simple program can be created in just a few hours and shown to a customer in the same visit. Programmers can repeatedly modify the program and see the results quickly. This allows them to try different ideas and see which one is best without investing a lot of time on dead-ends. This also applies to creating graphical user interfaces (GUIs). Simple "sketches" can be laid out in minutes because Python not only has several different GUI libraries available but also includes a simple library (Tkinter) by default. Another benefit of not having a compiler is that errors are immediately generated by the Python interpreter. Depending on the developing environment, it will automatically read through your code as you develop it and notify you of syntax errors. Logic errors won't be pointed out but a simple mouse click will launch the program and show you final product. If something isn't right, you can simply make a change and click the launch button again.

Procedural vs. Object-Oriented Programming : Python is somewhat unique in that you have two choices when developing your programs: procedural programming or object-oriented. As a matter of fact, you can mix the two in the same program. Briefly, procedural programming is a step-by-step process of developing

the program in a somewhat linear fashion. Functions (sometimes called subroutines) are called by the program at times to perform some processing, then control is returned back to the main program. C and BASIC are procedural languages. Object-oriented programming (OOP) is just that: programming with objects. Objects are created by distinct units of programming logic; variables and methods (an OOP term for functions) are combined into objects that do a particular thing. For example, you could model a robot and each body part would be a separate object, capable of doing different things but still part of the overall object. OOP is also heavily used in GUI development. Personally, I feel procedural programming is easier to learn, especially at first. The thought process is mostly straightforward and essentially linear. I never understood OOP until I started learning Python; it can be a difficult thing to wrap your head around, especially when you are still figuring out how to get your program to work in the first place. Procedural programming and OOP will be discussed in more depth later in the book. Each will get their own chapters and hopefully you will see how they build upon familiar concepts.

1.4 Creating and Running Python Programs:

A python program is a simple text file contains python statements. For example a file named *hello.py*, is simplest python statement which is a fully functional python program:

```
# Script Begins
print("Hello World")
# Script Ends
```

Writing program: Let us analyze the script line by line.

```
# Script Begins
print("Hello World")
# Script Ends
```

- Line 1 : [# Script Begins] In Python comments begin with #. So this statement is for readability of code and ignored by Python interpreter.
- Line 2 : [print("Hello World")] In a Python script to print something on the console print() function is used – it simply prints out a line (and also includes a newline unlike in C). One difference between Python 2 and Python 3 is the print statement. In Python 2, the "print" statement is not a function,

and therefore can be invoked without a parenthesis. However, in Python 3, it is a function, and must be invoked with parentheses.

- Line 3 : [# Script Ends] This is just another comment like Line 1.

We can create such a file of statements with any text editor you like. By convention, Python program files are given names that end in .py; technically, this naming scheme is required only for files that are "imported", as shown later in this subject, but most Python files have *.py* names for consistency.

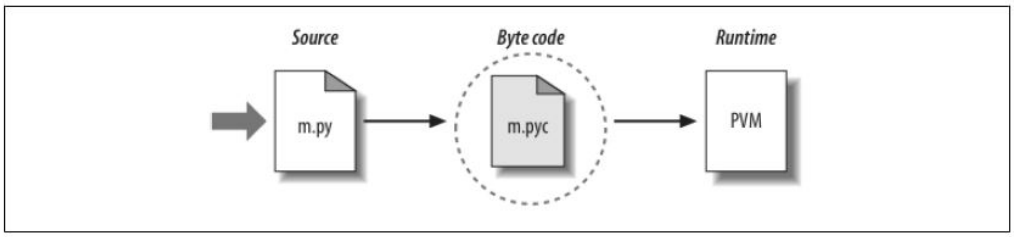


Figure 1.1: Python Virtual Machine

The Python Virtual Machine (PVM) Once a program has been compiled to byte code (or the byte code has been loaded from existing *.pyc* files), it is shipped off for execution to something generally known as the Python Virtual Machine (PVM, for the more acronym-inclined among you).The PVM is the runtime engine of Python; it's always present as part of the Python system, and it's the component that truly runs your scripts. Technically, it's just the last step of what is called the "Python interpreter."

Chapter 2

DATA TYPES: Identifiers and Keywords

Python Variables Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
    x = 5
y = ”John”
print(x)
print(y)
```

Output:

```
    5
John
```

Variables do not need to be declared with any particular type and can even change type after they have been set.

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

2.1 Creating variables and assigning values:

To create a variable in Python, all you need to do is specify the variable name, and then assign a value to it.

⟨*variable_name*⟩ = ⟨*value*⟩ # Integer

```
a = 2
print(a)
# Output: 2
# Integer
b = 9223372036854775807
print(b)
# Output: 9223372036854775807
# Floating point
pi = 3.14
print(pi)
# Output: 3.14
# String
c = 'A'
print(c)
# Output: A
# String
name = 'John Doe'
print(name)
# Output: John Doe
# Boolean
q = True
print(q)
# Output: True
# Empty value or null data type
x = None
print(x)
# Output: None
```


Variable assignment works from left to right. So the following will give you an syntax error.

You can not use python's keywords as a valid variable name. You can see the list of keyword by:
import keyword
print(keyword.kwlist)

2.2 Variable Names:

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

2.3 Output Variables:

The Python *print* statement is often used to output variables.

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

```
x = ”awesome”
print(”Python is ” + x)
```

Output:

```
Python is awesome
```

The above statement can be represented by:

```
x = ”Python is ”
y = ”awesome”
z = x + y
print(z)
```

Output:

```
Python is awesome
```

2.4 Datatypes:

2.4.1 Built-in Types:

Booleans:

bool: A boolean value of either True or False. Logical operations like and, or, not can be performed on booleans.
x or y # if x is False then y otherwise x.
x and y # if x is False then x otherwise y.
not x # if x is True then False, otherwise True.

In Python 2.x and in Python 3.x, a boolean is also an int. The bool type is a subclass of the int type and True and False are its only instances: issubclass(bool, int) # True isinstance(True, bool) # True isinstance(False, bool) # True If boolean values are used in arithmetic operations, their integer values (1 and 0 for True and False) will be used to return an integer result:
True + False == 1 # 1 + 0 == 1
True * True == 1 # 1 * 1 == 1

2.4.2 Numbers:

- int: Integer number
  a = 2
  b = 100
  c = 123456789
  d = 38563846326424324
  Integers in Python are of arbitrary sizes.
  Note: in older versions of Python, a long type was available and this was distinct from int. The two have been unified

- float: Floating point number;
  precision depends on the implementation and system architecture, for CPython the float datatype corresponds to a C double.
  a = 2.0
  b = 100.e0
  c = 123456789.e1

- complex: Complex numbers
  a = 2 + 1j
  b = 100 + 10j

2.5 Strings:

Python 3.x Version ≥ 3.0
str: a unicode string. The type of 'hello'
bytes: a byte string. The type of b'hello'
Python 2.x Version ≤ 2.7
str: a byte string. The type of 'hello'
bytes: synonym for str
unicode: a unicode string. The type of u'hello'

2.6 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

Arithmetic operators are used with numeric values to perform common mathematical operations:

Assignment operators

Assignment operators are used to assign values to variables:

Table 2.1: Arithmetic operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
*	Exponentiation	x ** y
//	Floor division	x // y

Table 2.2: Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
*=	x **= 3	x = x ** 3
&= x	&= 3	x = x & 3
—=	x —= 3	x = x — 3
≙ x	≙ 3	x = x $\hat{3}$
⟩⟩= x	⟩⟩= 3	x = x⟩⟩ 3
⟨⟨=	x ⟨⟨= 3	x = x ⟨⟨ 3

Comparison operators

Comparison operators are used to compare two values:

Table 2.3: Comparison operators

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
≥	Greater than or equal to	x ≥ y
≤	Less than or equal to	x ≤ y

Logical operators

Logical operators are used to combine conditional statements:

Table 2.4: Logical operators

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

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Table 2.5: Identity operators

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

Membership operators

Membership operators are used to test if a sequence is presented in an object:

Table 2.6: Membership operators

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

Bitwise operators

Bitwise operators are used to compare (binary) numbers:

2.7 Collection Types:

There are a number of collection types in Python. While types such as int and str hold a single value, collection types hold multiple values.

Table 2.7: Bitwise operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1.
÷	OR	Sets each bit to 1 if one of two bits is 1.
∧	XOR	Sets each bit to 1 if only one of two bits is 1.
~	NOT	Inverts all the bits
⟩⟩	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off.
⟨⟨	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the right most bits fall off.

2.7.1 Lists:

The list type is probably the most commonly used collection type in Python. List is a collection which is ordered and changeable. Allows duplicate members. Despite its name, a list is more like an array in other languages, mostly JavaScript. In Python, a list is merely an ordered collection of valid Python values. A list can be created by enclosing values, separated by commas, in square brackets:

int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
A list can be empty:
empty_list = []
The elements of a list are not restricted to a single data type, which makes sense given that Python is a dynamic language:
mixed_list = [1, 'abc', True, 2.34, None]
A list can contain another list as its element:
nested_list = [['a', 'b', 'c'], [1, 2, 3]]
The elements of a list can be accessed via an index, or numeric representation of their position. Lists in Python are zero-indexed meaning that the first element in the list is at index 0, the second element is at index 1 and so on:
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
Indices can also be negative which means counting from the end of the list (-1 being the index of the last element).
So, using the list from the above example:
print(names[-1]) # Eric
print(names[-4]) # Bob

Lists are mutable, so you can change the values in a list:
names[0] = 'Ann'
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
Besides, it is possible to add and/or remove elements from a list:

Methods in List:

Append object to end of list with L.append(object), returns None.
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
Add a new element to list at a specific index.
L.insert(index, object)
names.insert(1, "Nikki")
print(names)
# Outputs ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
Remove the first occurrence of a value with L.remove(value), returns None
names.remove("Bob")
print(names) # Outputs ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
List Methods Python has a set of built-in methods that you can use on lists.

Table 2.8: Python methods for List

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

2.7.2 Tuple

Tuple is a collection which is ordered and unchangeable. Allows duplicate members. Lists are enclosed in brackets [ ] and their elements and size can be changed, while tuples are enclosed in parentheses ( ) and cannot be updated. Tuples are immutable.

```
tuple = (123,'hello')
tuple1 = ('world')
print(tuple) # will output whole tuple. (123,'hello')
print(tuple[0]) # will output first value. (123)
print(tuple + tuple1) # will output (123,'hello','world')
tuple[1]='update' # this will give you error.
```

Python has two built-in methods that you can use on tuples.

Table 2.9: Python methods for Tuple

Method	Description
count()	Returns the number of times a specified value occurs in a tuple.
index()	Searches the tuple for a specified value and returns the position of where it was found.

2.7.3 Set:

Sets are unordered collections of unique objects, there are two types of set:

1. Sets - They are mutable and new elements can be added once sets are defined
   basket = 'apple', 'orange', 'apple', 'pear', 'orange', 'banana'
   print(basket) # duplicates will be removed
   Output 'orange', 'banana', 'pear', 'apple'
   a = set('abracadabra')
   print(a) # unique letters in a
   Output {'a', 'r', 'b', 'c', 'd'}

a.add('z')
print(a)
Output {'a', 'c', 'r', 'b', 'z', 'd'}

2. Frozen Sets - They are immutable and new elements cannot added after its defined.
b = frozenset('asdfagsa')
print(b)
frozenset('f', 'g', 'd', 'a', 's')
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
print(cities)
frozenset('Frankfurt', 'Basel', 'Freiburg')

Set Methods:

Python has a set of built-in methods that you can use on sets.

Table 2.10: Python methods for Set.

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

2.7.4 Dictionary:

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']

When the above code is executed, it produces the following result:
dict['Name']: Zara
dict['Age']: 7

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
When the above code is executed, it produces the following result:
dict['Age']: 8
dict['School']: DPS School

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the del statement. Following is a simple example:
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear(); # remove all entries in dict
del dict ; # delete entire dictionary
print "dict['Age']: ", dict['Age']

print ”dict[’School’]: ”, dict[’School’]

This produces the following result. Note that an exception is raised because after del dict dictionary does not exist any more
dict[’Age’]:
Traceback (most recent call last):
File ”test.py”, line 8, in ⟨module⟩
print ”dict[’Age’]: ”, dict[’Age’];
TypeError: ’type’ object is unsubscriptable

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys

1. More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example: *dict = {’Name’: ’Zara’, ’Age’: 7, ’Name’: ’Manni’}*
   print ”dict[’Name’]: ”, dict[’Name’]

   When the above code is executed, it produces the following result:
   dict[’Name’]: Manni

2. Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like [’key’] is not allowed. Following is a simple example:
   dict = {[’Name’]: ’Zara’, ’Age’: 7}
   print ”dict[’Name’]: ”, dict[’Name’]

   When the above code is executed, it produces the following result:
   Traceback (most recent call last):

   File ”test.py”, line 3, in ⟨module⟩
   dict = {[’Name’]: ’Zara’, ’Age’: 7};
   TypeError: unhashable type: ’list’

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions: Table 2.4

Table 2.11: Python methods for Dictionary

Sr.No.	Function with Description
1	cmp(dict1, dict2): Compares elements of both dict.
2	len(dict): Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	str(dict) : Produces a printable string representation of a dictionary.
4	type(variable): Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python includes following dictionary methods:Table 2.5

Table 2.12: All Python methods for Dictionary

Sr.No.	Methods with Description
1	dict.clear():Removes all elements of dictionary dict.
2	dict.copy(): Returns a shallow copy of dictionary dict.
3	dict.fromkeys():Create a new dictionary with keys from seq and values set to value.
4	dict.get(key, default=None): For key key, returns value or default if key not in dictionary.
5	dict.has_key(key): Returns true if key in dictionary dict, false otherwise.
6	dict.items(): Returns a list of dict's (key, value) tuple pairs.
7	dict.keys() : Returns list of dictionary dict's keys.
8	dict.setdefault(key, default=None): Similar to get(), but will set dict[key]=default if key is not already in dict.
9	dict.update(dict2): Adds dictionary dict2's key-values pairs to dict.
10	dict.values(): Returns list of dictionary dict's values.

2.7.5 Python Casting:

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

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

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Table 2.13: Python Data Type Conversion

Sr.No.	Methods with Description
1	int(x [,base]): Converts x to an integer. base specifies the base if x is a string.
2	long(x [,base]): Converts x to a long integer. base specifies the base if x is a string.
3	float(x): Converts x to a floating-point number.
4	complex(real [,imag]): Creates a complex number.
5	str(x): Converts object x to a string representation.
6	repr(x): Converts object x to an expression string.
7	eval(str): Evaluates a string and returns an object.
8	tuple(s): Converts s to a tuple.
9	list(s) : Converts s to a list.
10	set(s) : Converts s to a set.
11	dict(d): Creates a dictionary. d must be a sequence of (key,value) tuples.
12	frozenset(s): Converts s to a frozen set.
13	chr(x): Converts an integer to a character.
14	unichr(x): Converts an integer to a Unicode character.
15	ord(x) : Converts a single character to its integer value.
16	hex(x): Converts an integer to a hexadecimal string.
17	oct(x): Converts an integer to an octal string.

2.7.6 List of Keywords in Python:

Keywords are the reserved words in Python. We cannot use a keyword as variable name, function name or any other identifier.

Here's a list of all keywords in Python Programming:

Table 2.14: List of Keywords in Python

False	class	finally	is	return
None	continues	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

2.8 Python Mathematical Functions:

Learn about all the mathematical functions available in Python and how you can use them in your program.

2.8.1 What is math module in Python?

The math module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using import math.

It gives access to the underlying C library functions. For example,

import math
math.sqrt(4)

Output:

2

This module does not support complex datatypes. The cmath module is the complex counterpart.

2.8.2 Functions in Python Math Module

Here is the list of all the functions and attributes defined in math module with a brief explanation of what they do.

Table 2.15: Python Mathematical Functions.

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y.
fabs(x)	Returns the absolute value of x.
factorial(x)	Returns the factorial of x.
floor(x)	Returns the largest integer less than or equal to x.
fmod(x, y)	Returns the remainder when x is divided by y.
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e).
fsum(iterable)	Returns an accurate floating point sum of values in the iterable.
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number).
isinf(x)	Returns True if x is a positive or negative infinity.
isnan(x)	Returns True if x is a NaN.
ldexp(x, i)	Returns x * (2**i).
modf(x)	Returns the fractional and integer parts of x.
trunc(x)	Returns the truncated integer value of x.
exp(x)	Returns e**x.
expm1(x)	Returns e**x - 1.
log(x[, base])	Returns the logarithm of x to the base (defaults to e).
log1p(x)	Returns the natural logarithm of 1+x.
log2(x)	Returns the base-2 logarithm of x.
log10(x)	Returns the base-10 logarithm of x.
pow(x, y)	Returns x raised to the power y.
sqrt(x)	Returns the square root of x.
acos(x)	Returns the arc cosine of x.
asin(x)	Returns the arc sine of x.
atan(x)	Returns the arc tangent of x.
atan2(y, x)	Returns atan(y / x).
cos(x)	Returns the cosine of x.
hypot(x, y)	Returns the Euclidean norm, sqrt(x*x + y*y).
sin(x)	Returns the sine of x.
tan(x)	Returns the tangent of x.
degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.
acosh(x)	Returns the inverse hyperbolic cosine of x.
asinh(x)	Returns the inverse hyperbolic sine of x.
atanh(x)	Returns the inverse hyperbolic tangent of x.
cosh(x)	Returns the hyperbolic cosine of x.
sinh(x)	Returns the hyperbolic cosine of x.
tanh(x)	Returns the hyperbolic tangent of x.
erf(x)	Returns the error function at x.
erfc(x)	Returns the complementary error function at x.
gamma(x)	Returns the Gamma function at x.
lgamma(x)	Returns the natural logarithm of the absolute value of the Gamma function at x.
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...).
e	mathematical constant e (2.71828...).

2.9 Input and Output in Python:

Python input() Function:

The input() function allows user input.Syntax is:
input(prompt)
Example Use the prompt parameter to write a message before the input:

Table 2.16: Parameter Values for input()

Parameter	Description
prompt	A String, representing a default message before the input.

x = input('Enter your name:')
print('Hello, ' + x)
Output:
Hello Name

Output formatting : The Pythonic Way: The string method "format"

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

The template (or format string) is a string which contains one or more format codes (fields to be replaced) embedded in constant text. The "fields to be replaced" are surrounded by curly braces {}. The curly braces and the "code" inside will be substituted with a formatted value from one of the arguments, according to the rules which we will specify soon. Anything else, which is not contained in curly braces will be literally printed, i.e. without any changes. If a brace character has to be printed, it has to be escaped by doubling it: {{ and }}.

There are two kinds of arguments for the *.format()* method. The list of arguments starts with zero or more positional arguments (p0, p1, ...), it may be followed by zero or more keyword arguments of the form name=value.

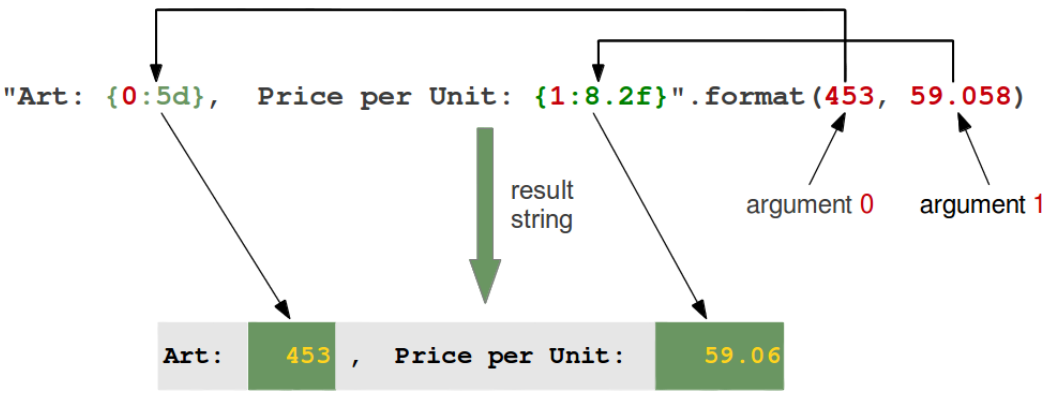


Figure 2.1: Output formatting.

A positional parameter of the format method can be accessed by placing the index of the parameter after the opening brace, e.g. {0} accesses the first parameter, {1} the second one and so on. The index inside of the curly braces can be followed by a colon and a format string, which is similar to the notation of the string modulo, which we had discussed in the beginning of the chapter of our tutorial, e.g. {0:5d} . If the positional parameters are used in the order in which they are written, the positional argument specifiers inside of the braces can be omitted, so '{} {} {}' corresponds to '{0} {1} {2}'. But they are needed, if you want to access them in different orders: '{2} {1} {0}'.

Examples of positional parameters:

⟩⟩⟩ "First argument: {0}, second one: {1}".format(47,11)
'First argument: 47, second one: 11'
⟩⟩⟩ "Second argument: {1}, first one: {0}".format(47,11)
'Second argument: 11, first one: 47'
⟩⟩⟩ "Second argument: {1:3d}, first one: {0:7.2f}".format(47.42,11)
'Second argument: 11, first one: 47.42'
⟩⟩⟩ "First argument: {}, second one: {}".format(47,11)
'First argument: 47, second one: 11'
⟩⟩⟩ # arguments can be used more than once:
...
⟩⟩⟩ "various precisions: {0:6.2f} or {0:6.3f}".format(1.4148)
'various precisions: 1.41 or 1.415'
⟩⟩⟩ Some More Examples:
⟩⟩⟩ x = 5; y = 10
⟩⟩⟩ print('The value of x is {} and y is {}'.format(x,y))
The value of x is 5 and y is 10

Here the curly braces {} are used as placeholders. We can specify the order in

which it is printed by using numbers (tuple index).

Option	Meaning
'<'	The field will be left-aligned within the available space. This is usually the default for strings.
'>'	The field will be right-aligned within the available space. This is the default for numbers.
'0'	If the width field is preceded by a zero ('0') character, sign-aware zero-padding for numeric types will be enabled. ⟩⟩⟩ x = 378 ⟩⟩⟩ print("The value is :06d".format(x)) The value is 000378 ⟩⟩⟩ x = -378 ⟩⟩⟩ print("The value is :06d".format(x)) The value is -00378
','	This option signals the use of a comma for a thousands separator. ⟩⟩⟩ print("The value is :,".format(x)) The value is 78,962,324,245 ⟩⟩⟩ print("The value is 0:6,d".format(x)) The value is 5,897,653,423 ⟩⟩⟩ x = 5897653423.89676 ⟩⟩⟩ print("The value is 0:12,.3f".format(x)) The value is 5,897,653,423.897
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form "+000000120". This alignment option is only valid for numeric types.
'∧'	Forces the field to be cantered within the available space.

Chapter 3

Python Program Flow Control

Decision making is required when we want to execute a code only if a certain condition is satisfied.

3.1 If Statement

3.1.1

Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True. If the text expression is False, the statement(s) is not executed. In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first unindented line marks the end. Python interprets non-zero values as True. None and 0 are interpreted as False.

if test expression:
⟩⟩⟩ statement(s)

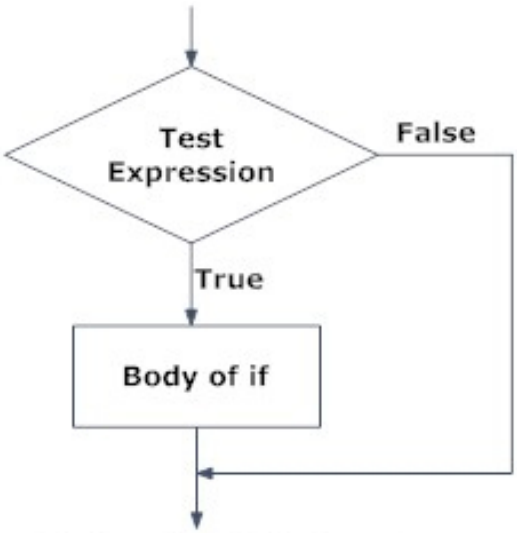


Fig: Operation of if statement

Figure 3.1: If...else control flow.

Example: Python if Statement

```
# If the number is positive, we print an appropriate message
num = 3
if num > 0:
⟩⟩⟩print(num, ”is a positive number.”)
print(”This is always printed.”)
num = -1 if num > 0:
⟩⟩⟩print(num, ”is a positive number.”)
print(”This is also always printed.”)
```

When you run the program, the output will be:

```
3 is a positive number
This is always printed
This is also always printed.
```

In the above example, num > 0 is the test expression. The body of if is executed only if this evaluates to True.

When variable num is equal to 3, test expression is true and body inside body of if is executed.

If variable num is equal to -1, test expression is false and body inside body of if is

skipped.
The print() statement falls outside of the if block (unindented). Hence, it is executed regardless of the test expression.

3.2 if...else Statement

The if..else statement evaluates test expression and will execute body of if only when test condition is True. If the condition is False, body of else is executed. Indentation is used to separate the blocks.

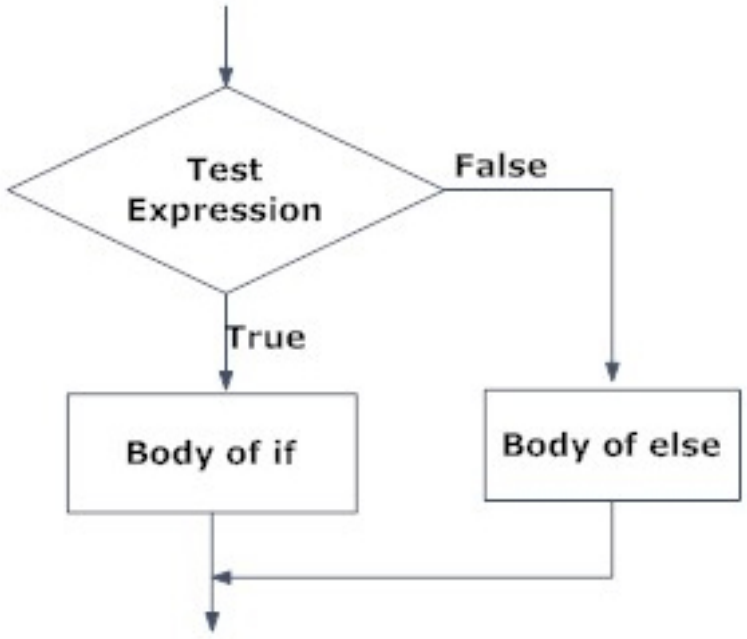


Fig: Operation of if...else statement

Figure 3.2: If...else control flow.

Example of if...else

```
# Program checks if the number is positive or negative
# And displays an appropriate message

    num = 3
```

```
    # Try these two variations as well.
# num = -5
# num = 0


    if num >= 0:


    print(”Positive or Zero”)


    else:


    print(”Negative number”)
```

In the above example, when num is equal to 3, the test expression is true and body of if is executed and body of else is skipped.
If num is equal to -5, the test expression is false and body of else is executed and body of if is skipped.
If num is equal to 0, the test expression is true and body of if is executed and body of else is skipped.

3.2.1 if...elif...else Statement

Syntax of if...elif...else
if test expression: Body of if elif test expression: Body of elif else: Body of else
The elif is short for else if. It allows us to check for multiple expressions.
If the condition for if is False, it checks the condition of the next elif block and so on.
If all the conditions are False, body of else is executed.
Only one block among the several if...elif...else blocks is executed according to the condition.
The if block can have only one else block. But it can have multiple elif blocks.

Flowchart of if...elif...else

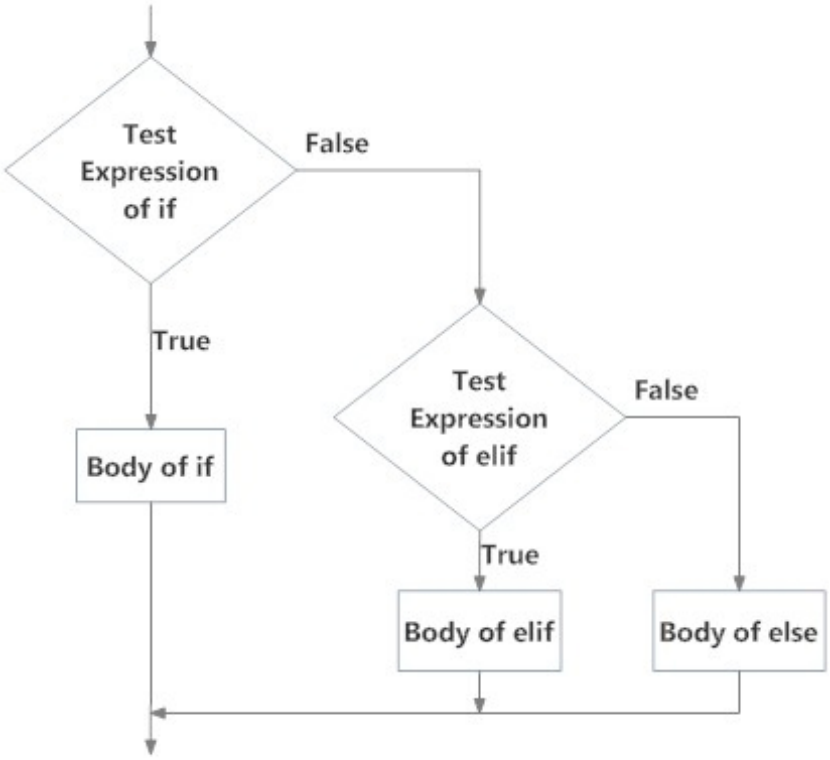


Fig: Operation of if...elif...else statement

Figure 3.3: If...elif...else control flow.

Example of if...elif...else

```
# In this program,
# we check if the number is positive or
# negative or zero and
# display an appropriate message

    num = 3.4

    # Try these two variations as well:
# num = 0
# num = -4.5

    if num > 0:
print(”Positive number”)
elif num == 0:
print(”Zero”)
```

else:
print(”Negative number”)
When variable num is positive, Positive number is printed.
If num is equal to 0, Zero is printed.
If num is negative, Negative number is printed

3.3 Python Nested if statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming. Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

Python Nested if Example # In this program, we input a number # check if the number is positive or # negative or zero and display # an appropriate message # This time we use nested if

num = float(input(”Enter a number: ”)) if num >= 0:

if num == 0:
print(”Zero”)
else:

print(”Positive number”)
else:

print(”Negative number”)

Output 1 Enter a number: 5

Positive number
Output 2
Enter a number: -1
Negative number
Output 3
Enter a number: 0
Zero

3.4 Python for Loop

3.4.1 What is for loop in Python?

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

for val in sequence:

Body of for

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flowchart of for Loop

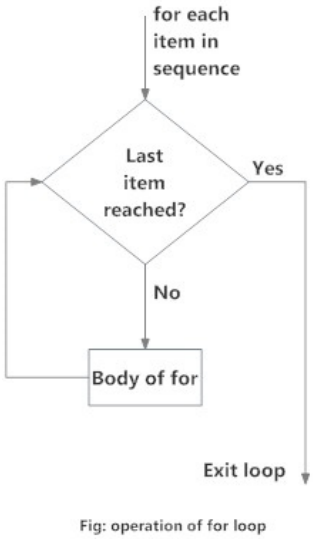


Fig: operation of for loop

Figure 3.4: Flowchart of for Loop.

Example: Python for Loop

```
# Program to find the sum of all numbers stored in a list

    # List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
```

```
    # variable to store the sum
sum = 0

    # iterate over the list
for val in numbers:

                sum = sum+val
print("The sum is", sum)

    # Output: The sum is 48
```

when you run the program, the output will be:
The sum is 48

3.5 The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided. This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go. To force this function to output all the items, we can use the function list(). The following example will clarify this.

```
    # Output: range(0, 10)
print(range(10))

    # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(range(10)))
    # Output: [2, 3, 4, 5, 6, 7]
print(list(range(2, 8)))

    # Output: [2, 5, 8, 11, 14, 17]
print(list(range(2, 20, 3)))
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing. Here is an example. # Program to iterate through a list using indexing

```
    genre = ['pop', 'rock', 'jazz']
```

```
    # iterate over the list using index
for i in range(len(genre)):

                print(”I like”, genre[i])
```

When you run the program, the output will be:

```
I like pop
I like rock
I like jazz
```

3.6 for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts. break statement can be used to stop a for loop. In such case, the else part is ignored. Hence, a for loop’s else part runs if no break occurs.
Here is an example to illustrate this.

```
digits = [0, 1, 5]

    for i in digits:

                print(i)

    else:

                print(”No items left.”)
```

When you run the program, the output will be:

```
0
1
5
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints No items left.
Python while Loop

3.7 Python Wile loop

3.7.1 What is while loop in Python?

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax of while Loop in Python

while test_expression:

Body of while

In while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False. In Python, the body of the while loop is determined through indentation.

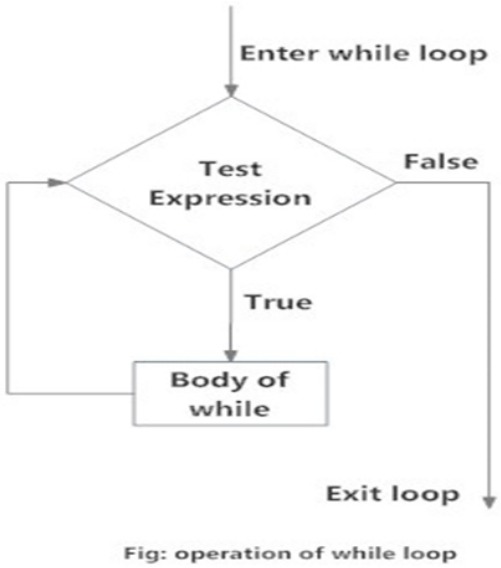


Fig: operation of while loop

Figure 3.5: Flowchart of while Loop.

Example: Python while Loop

```
# Program to add natural
# numbers upto
# sum = 1+2+3+...+n

    # To take input from the user,
```

```
# n = int(input("Enter n: "))

    n = 10

    # initialize sum and counter
sum = 0
i = 1

    while i <= n:

                sum = sum + i

                i = i+1 # update counter

    # print the sum
print("The sum is", sum)
```

When you run the program, the output will be:

```
Enter n: 10
The sum is 55
```

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (10 in our program). We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop).Finally the result is displayed.

3.7.2 while loop with else

Same as that of for loop, we can have an optional else block with while loop as well.The else part is executed if the condition in the while loop evaluates to False.

The while loop can be terminated with a break statement. In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

Here is an example to illustrate this. # Example to illustrate

```
# the use of else statement
# with the while loop
counter = 0
while counter < 3:
print("Inside loop")
counter = counter + 1
```

else:
print(”Inside else”)

3.8 Python break and continue

What is the use of break and continue in Python?
In Python, break and continue statements can alter the flow of a normal loop. Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The break and continue statements are used in these cases.

3.8.1 Python break statement

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Syntax of break
break
Flowchart of break

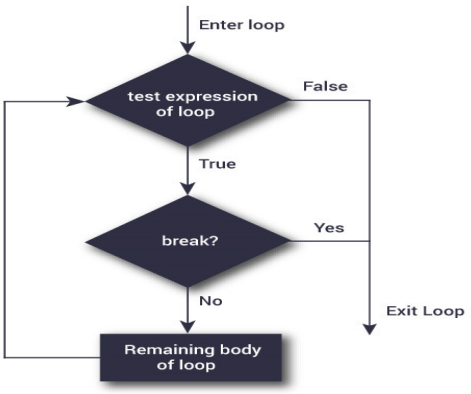


Figure 3.6: Flowchart of break.

The working of break statement in for loop and while loop is shown below.

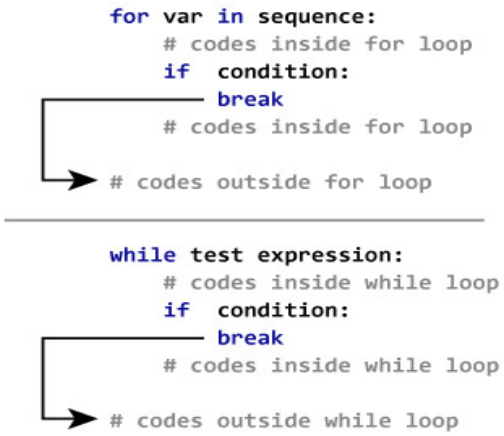


Figure 3.7: Working of break statement.

Example: Python break # Use of break statement inside loop

```
for val in ”string”:
        if val == ”i”:
        break
        print(val)
print(”The end”)
```

Output

```
s
t
r
The end
```

In this program, we iterate through the ”string” sequence. We check if the letter is ”i”, upon which we break from the loop. Hence, we see in our output that all the letters up till ”i” gets printed. After that, the loop terminates.

3.8.2 Python continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Syntax of Continue

continue

Flowchart of continue

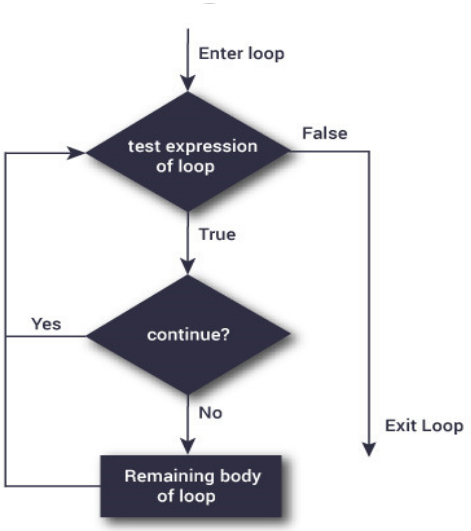


Figure 3.8: Flowchart of continue.

The working of continue statement in for and while loop is shown below.

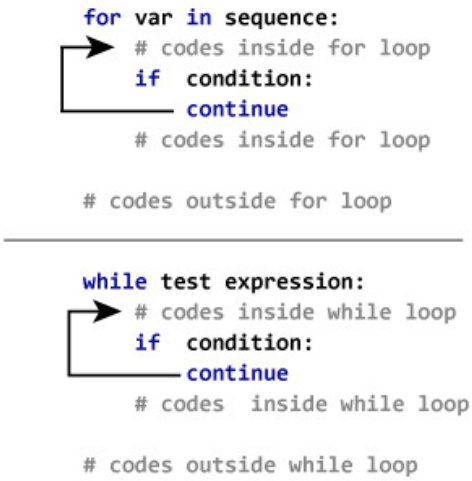


Figure 3.9: Working of continue.

Example: Python continue

```
for val in ”string”:
        if val == ”i”:
        continue
        print(val)
print(”The end”)
```

Output

```
s
t
r
n
g
The end
```

This program is same as the above example except the break statement has been replaced with continue. We continue with the loop, if the string is ”i”, not executing the rest of the block. Hence, we see in our output that all the letters except ”i” gets printed.

3.9 Python pass statement

What is pass statement in Python?

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when pass is executed. It results into no operation (NOP).

Syntax of pass

pass

We generally use it as a placeholder. Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

Example: pass Statement

```
# pass is just a placeholder for.
# functionality to be added later.

sequence = 'p', 'a', 's', 's'

for val in sequence:

        pass
```

Chapter 4

Python Functions

Functions are an essential part of the Python programming language: you might have already encountered and used some of the many fantastic functions that are built-in in the Python language or that come with its library ecosystem. However, as a Data Scientist, you'll constantly need to write your own functions to solve problems that your data poses to you.

How Function works in Python?

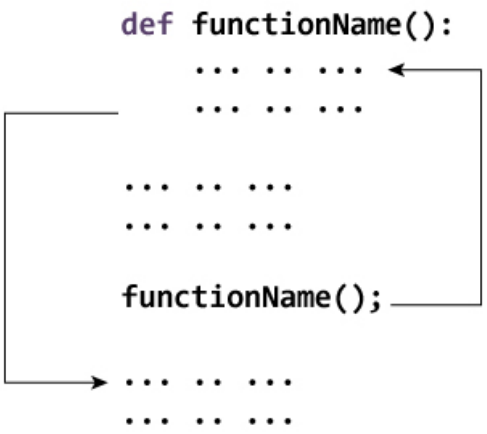


Figure 4.1: Working of Function.

There are three types of functions in Python:

1. Built-in functions, such as help() to ask for help, min() to get the minimum

value, print() to print an object to the terminal,... You can find an overview with more of these functions here.

2. User-Defined Functions (UDFs), which are functions that users create to help them out; And

3. Anonymous functions, which are also called lambda functions because they are not declared with the standard def keyword.

4.1 Functions vs Methods

A method refers to a function which is part of a class. You access it with an instance or object of the class. A function doesn't have this restriction: it just refers to a standalone function. This means that all methods are functions, but not all functions are methods.

Consider this example, where you first define a function plus() and then a Summation class with a sum() method: # Define a function 'plus()'

```
def plus(a,b):

return a + b
# Create a 'Summation' class

class Summation(object):

        def sum(self, a, b):

        self.contents = a + b

        return self.contents
```

If you now want to call the sum() method that is part of the Summation class, you first need to define an instance or object of that class. So, let's define such an object:

```
# Instantiate 'Summation' class to call 'sum()'

sumInstance = Summation()

sumInstance.sum(1,2)
```

Remember that this instantiation not necessary for when you want to call the function plus(). You would be able to execute plus(1,2).

4.2 Parameters vs Arguments

Parameters are the names used when defining a function or a method, and into which arguments will be mapped. In other words, arguments are the things which are supplied to any function or method call, while the function or method code refers to the arguments by their parameter names.

Consider the following example and look back to the above code you pass two arguments to the sum() method of the Summation class, even though you previously defined three parameters, namely, self, a and b.

What happened to self?

The first argument of every class method is always a reference to the current instance of the class, which in this case is Summation. By convention, this argument is called self.

This all means that you don't pass the reference to self in this case because self is the parameter name for an implicitly passed argument that refers to the instance through which a method is being invoked. It gets inserted implicitly into the argument list.

4.3 How To Define A Function: User-Defined Functions (UDFs)

The four steps to defining a function in Python are the following:

1. Use the keyword def to declare the function and follow this up with the function name.

2. Add parameters to the function: they should be within the parentheses of the function. End your line with a colon.

3. Add statements that the functions should execute.

4. End your function with a return statement if the function should output something. Without the return statement, your function will return an object None.

```
def hello():
name = str(input("Enter your name: "))
if name:
```

```
print (”Hello ” + str(name))
else:
print(”Hello World”)
return hello()
```