if True:

print "True"

else:

print "False"

if expression :

suite

elif expression :

suite

else :

suite

## Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

total = item\_one + \

item\_two + \

item\_three

## 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."""

# First comment

**Python** has five standard data types −

* Numbers
* String
* List
* Tuple
* Dictionary

**Strings**

str = 'Hello World!'

print str # Prints complete string

print str[0] # Prints first character of the string

print str[2:5] # Prints characters starting from 3rd to 5th

print str[2:] # Prints string starting from 3rd character

print str \* 2 # Prints string two times

print str + "TEST" # Prints concatenated string

## Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']

print list # Prints complete list

print list[0] # Prints first element of the list

print list[1:3] # Prints elements starting from 2nd till 3rd

print list[2:] # Prints elements starting from 3rd element

print tinylist \* 2 # Prints list two times

print list + tinylist # Prints concatenated lists

## 

## 

## 

## 

## 

## Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as read-only lists.

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

tinytuple = (123, 'john')

print tuple # Prints complete list

print tuple[0] # Prints first element of the list

print tuple[1:3] # Prints elements starting from 2nd till 3rd

print tuple[2:] # Prints elements starting from 3rd element

print tinytuple \* 2 # Prints list two times

print tuple + tinytuple # Prints concatenated lists

## Python Dictionary

Key value pairs

dict = {}

dict['one'] = "This is one"

dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one'] # Prints value for 'one' key

print dict[2] # Prints value for 2 key

print tinydict # Prints complete dictionary

print tinydict.keys() # Prints all the keys

print tinydict.values() # Prints all the values

## Python Membership Operators

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | Evaluates to true if it finds a variable in the specified sequence and false otherwise. | x in y, here in results in a 1 if x is a member of sequence y. |
| not in | Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. | x not in y, here not in results in a 1 if x is not a member of sequence y. |

## Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below −

|  |  |  |
| --- | --- | --- |
| **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). |

## Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example −

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'

## Built-in String Methods

## Python includes the following built-in methods to manipulate strings −

## 

<https://www.tutorialspoint.com/python/python_strings.htm>

## Basic List Operations

Lists respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| len([1, 2, 3]) | 3 | Length |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | Concatenation |
| ['Hi!'] \* 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition |
| 3 in [1, 2, 3] | True | Membership |
| for x in [1, 2, 3]: print x, | 1 2 3 | Iteration |

**[list(seq)](https://www.tutorialspoint.com/python/list_list.htm)**

Converts a tuple into list.

## Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

#!/usr/bin/python

tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');

# Following action is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2;

print tup3;

**Dictionary**

del dict['Name']; # remove entry with key 'Name'

dict.clear(); # remove all entries in dict

del dict ; # delete entire dictionary

(b) 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.fromkeys

**dict.fromkeys(seq[, value])**

seq = ('name', 'age', 'sex')

dict = dict.fromkeys(seq)

print "New Dictionary : %s" % str(dict)

dict = dict.fromkeys(seq, 10)

print "New Dictionary : %s" % str(dict)

When we run above program, it produces following result −

New Dictionary : {'age': None, 'name': None, 'sex': None}

New Dictionary : {'age': 10, 'name': 10, 'sex': 10}

Python includes following dictionary methods −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods with Description** |
| 1 | **[dict.clear()](https://www.tutorialspoint.com/python/dictionary_clear.htm)**  Removes all elements of dictionary *dict* |
| 2 | **[dict.copy()](https://www.tutorialspoint.com/python/dictionary_copy.htm)**  Returns a shallow copy of dictionary *dict* |
| 3 | **[dict.fromkeys()](https://www.tutorialspoint.com/python/dictionary_fromkeys.htm)**  Create a new dictionary with keys from seq and values *set* to *value*. |
| 4 | **[dict.get(key, default=None)](https://www.tutorialspoint.com/python/dictionary_get.htm)**  For *key* key, returns value or default if key not in dictionary |
| 5 | **[dict.has\_key(key)](https://www.tutorialspoint.com/python/dictionary_has_key.htm)**  Returns *true* if key in dictionary *dict*, *false* otherwise |
| 6 | **[dict.items()](https://www.tutorialspoint.com/python/dictionary_items.htm)**  Returns a list of *dict*'s (key, value) tuple pairs |
| 7 | **[dict.keys()](https://www.tutorialspoint.com/python/dictionary_keys.htm)**  Returns list of dictionary dict's keys |
| 8 | **[dict.setdefault(key, default=None)](https://www.tutorialspoint.com/python/dictionary_setdefault.htm)**  Similar to get(), but will set dict[key]=default if *key* is not already in dict |
| 9 | **[dict.update(dict2)](https://www.tutorialspoint.com/python/dictionary_update.htm)**  Adds dictionary *dict2*'s key-values pairs to *dict* |
| 10 | **[dict.values()](https://www.tutorialspoint.com/python/dictionary_values.htm)**  Returns list of dictionary *dict*'s values |

FUNCTIONS

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

## Function Arguments

You can call a function by using the following types of formal arguments −

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments

## Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme()

When the above code is executed, it produces the following result −

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

## Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme( str = "My string")

When the above code is executed, it produces the following result −

My string

The following example gives more clear picture. Note that the order of parameters does not matter.

#!/usr/bin/python

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

## Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

#!/usr/bin/python

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Name: miki

Age 35

## Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length*arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

#!/usr/bin/python

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print "Output is: "

print arg1

for var in vartuple:

print var

return;

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, it produces the following result −

Output is:

10

Output is:

70

60

50

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )

## The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax −

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib, use the following statement −

from fib import fibonacci

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

## The *PYTHONPATH* Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system −

set PYTHONPATH = c:\python20\lib;

And here is a typical PYTHONPATH from a UNIX system −

set PYTHONPATH = /usr/local/lib/python

File I/O

# Open a file

fo = open("foo.txt", "wb")

print "Name of the file: ", fo.name

print "Closed or not : ", fo.closed

print "Opening mode : ", fo.mode

print "Softspace flag : ", fo.softspace

<https://www.tutorialspoint.com/python/python_files_io.htm>

* [File Object Methods](https://www.tutorialspoint.com/python/file_methods.htm): The *file* object provides functions to manipulate files.
* [OS Object Methods](https://www.tutorialspoint.com/python/os_file_methods.htm): This provides methods to process files as well as directories.

Data frame slicing, loc, iloc

<https://datacarpentry.org/python-ecology-lesson/03-index-slice-subset/index.html>

<https://medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c>