Data Analysis Using Python

Samatrix Consulting Pvt Ltd

Samatrix.io

Data Structures, Functions, Exceptions, Files

Python Data Structures

- The Python data structures such as `tuples`, `lists`, `dicts`, and `sets`, are simple but powerful.
- Any python programmer should master these data structures

Tuple

A fixed length and immutable sequence of Python objects is called Tuple.

You can create a tuple with a comma-separated sequence of values

```
In [1]: tup = 2, 3, 4

In [2]: tup
Out[2]: (2, 3, 4)
```

Tuple

If you want to define a complicated expression of tuple, you can enclose the values in a parenthesis.

Given below is the example of creating a tuple of tuples

```
In [3]: nested_tuple = (2, 3, 4), (5, 6)

In [4]: nested_tuple
Out[4]: ((2, 3, 4), (5, 6))
```

Samatrix.io

Tuple

Any sequence or iterator can also be converted into the tuple by invoking tuple;

```
In [5]: tuple([7, 6, 3])
Out[5]: (7, 6, 3)

In [6]: tupl = tuple('Python')

In [7]: tupl
Out[7]: ('P', 'y', 't', 'h', 'o', 'n')
```

Samatrix.io

Tuple

You can access the elements of the tuple using the square brackets.

In python the sequences are 0-indexed.

```
In [8]: tupl[0]
Out[8]: ‘P’
```

Even though the objects stored in a tuple may be mutable, the tuple are immutable.

In other words, we can say, we cannot modify the objects stored in each slot.

Samatrix.io

Tuple

```
In [9]: tupl = tuple(['abc', [3, 4], False])

In [10]: tupl[2] = True
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-42f621c38a64> in <module>
----> 1 tupl[2] = True

TypeError: 'tuple' object does not support item assignment
```

Samatrix.io

Tuple

If the object inside the tuple is mutable, if can be modified in-place

```
In [11]: tupl[1].append(5)

In [12]: tupl
Out[12]: ('abc', [3, 4, 5], False)
```

Tuples can be concatenated using the + operator. We get a longer tuple as a result.

```
In [13]: (5, 'def', None) + (1, 2, 3) + ('abc',)
Out[13]: (5, 'def', None, 1, 2, 3, 'abc')
```

Tuple

If you multiply the tuple by an integer, that many copies of the tuple will concatenate together.

In this case, the objects are not copied.

The reference to the objects are copied.

```
In [14]: ('abc', 'def') * 4
Out[14]: ('abc', 'def', 'abc', 'def', 'abc', 'def', 'abc', 'def')
```

Samatrix.io

Unpacking Tuple

In Python, when we assign the values to a tule, we call it packing the tuple.

When we extract the variables back into the tuple, this is called unpacking a tuple.

The variables should have tuple-like expressions

```
In [15]: tupl = (5, 6, 7)

In [16]: a, b, c = tupl

In [17]: c
Out[17]: 7
```

Samatrix.io

Unpacking Tuple

We can even unpack the nested tuple

```
In [18]: tupl = 1, 2, (3, 4)

In [19]: p, q, (r, s) = tupl

In [20]: s
Out[20]: 4
```

Samatrix.io

Unpacking Tuple

One of the use-case of tuples is the swapping of the variable names.

In many programming languages this can be achieved using the following sequence of steps

```
temp = a
a = b
b = temp
```

In Python the swapping can be done more easily

Unpacking Tuple

```
In [21]: a, b = 1, 2
In [22]: a
Out[22]: 1
In [23]: b
Out[23]: 2
In [24]: b, a = a, b
In [25]: a
Out[25]: 2
In [26]: b
Out[26]: 1
```

Samatrix.io

Unpacking Tuple

Another use case of variable unpacking is iterating over sequence of tuples or lists:

```
In [27]: a_seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: for a, b, c in a_seq:
    ...:     print('a={0}, b={1}, c={2}'.format(a, b, c))
    ...:
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

Unpacking Tuple

For a tuple the number of variables must match the number of values in the tuple.

If they do not match, we should use an asterisk to collect the remaining values as a list.

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list.

In the following example, we assign the remaining values to a list called rest.

```
In [29]: val = 1, 2, 3, 4, 5

In [30]: a, b, *rest = val

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```

Samatrix.io

Unpacking Tuple

Many programmers prefer underscore (_) for such list of variables

```
In [33]: a, b, *_ = val
```

Tuple Methods

Since the tuples are immutable,

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

`count()` returns the number of occurrences of a value.

Whereas `index()` searches the tuple for a specified value and return the position where the value was found.

```
In [34]: tupl = 1, 2, 2, 2, 3, 2, 4

In [35]: tupl.count(2)
Out[35]: 4

In [36]: tupl.index(3)
Out[36]: 4
```

Samatrix.io

List

We use python lists to store multiple items in a single variable. We can create the lists using square brackets `[ ]` or `list()` function.

The list items have following properties

Ordered: The items have a defined order and that order will not change. If you add a new item to a list, the item will be added at the end of the list. Even though using some list methods, you can change the order but in general, the order of the items does not change.

Mutable: The list items are mutable. That means that we can change, add, or remove the items from the list after the list has been created

List

Allow duplicates: The lists are indexed that allows the list items to have the same value.

List Length: You can determine the number of items in the list using `len()` function.

List item data type: The list can contain items of any data type

Indexed: The list items are indexed. The first item of the list has index `[0]`. The second item has index `[1]`

List

We can create the list using square bracket `[ ]`

```
In [37]: a_list = [3, 6, 9, None]
```

We can create the list using `list()` function

```
In [38]: tupl = ('abc', 'def', 'ghi')

In [39]: b_list = list(tupl)

In [40]: b_list
Out[40]: ['abc', 'def', 'ghi']
```

Samatrix.io

List

We can update a list item. In this case second list item has been updated to 'xyz'

```
In [41]: b_list[1] = 'xyz'

In [42]: b_list
Out[42]: ['abc', 'xyz', 'ghi’]
```

List

We also use the lists in data processing as an iterator

```
In [43]: gen = range(10)

In [44]: gen
Out[44]: range(0, 10)

In [45]: list(gen)
Out[45]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Samatrix.io

Adding and Removing Elements

Use `append` method to append an element at the end of the list

```
In [46]: b_list.append('jkl')

In [47]: b_list
Out[47]: ['abc', 'xyz', 'ghi', 'jkl']
```

Adding and Removing Elements

Use `insert` method to insert an element at a specific location in the list. However, the index number should be between 0 and length of the list inclusive.

```
In [48]: b_list.insert(1, 'def')

In [49]: b_list
Out[49]: ['abc', 'def', 'xyz', 'ghi', 'jkl']
```

Please note that `insert` method is computationally expensive when compared with `append`.

In the case of `insert`, the reference to the subsequent elements has to be shifted internally so that a room can be created for the new element

Adding and Removing Elements

Use `pop` to remove and return an element at a particular index item. `pop` is a reverse operation of `insert`.

```
In [50]: b_list.pop(2)
Out[50]: 'xyz'

In [51]: b_list
Out[51]: ['abc', 'def', 'ghi', 'jkl']
```

Samatrix.io

Adding and Removing Elements

We can also remove the elements using `remove`. The `remove` method locates the first instance of the value and remove the same from the list.

```
In [52]: b_list.append('abc')

In [53]: b_list
Out[53]: ['abc', 'def', 'ghi', 'jkl', 'abc’]

In [54]: b_list.remove('abc')

In [55]: b_list
Out[55]: ['def', 'ghi', 'jkl', 'abc']
```

Samatrix.io

Adding and Removing Elements

In order to search whether the list contains a specific value, you can use `in` keyword.

```
In [56]: 'ghi' in b_list
Out[56]: True
```

In order to search whether the list does not contains a specific value, you can use `not in` keyword.

```
In [57]: 'ghi' not in b_list
Out[57]: False
```

Samatrix.io

Concatenating and Combining List

If you want to concatenate or add two lists together, you can use + operator

```
In [58]: [5, 'abc', None] + [1, 3, (6, 9)]
Out[58]: [5, 'abc', None, 1, 3, (6, 9)]
```

But if the list is predefined, you can use `extend` method to add multiple elements to it

```
In [57]: ls = [5, 'abc', None]

In [58]: ls.extend([1, 3, (6, 9)])

In [59]: ls
Out[59]: [5, 'abc', None, 1, 3, (6, 9)]
```

Samatrix.io

Concatenating and Combining List

- Please note: If we concatenate the list by addition, a new list is created and the objects are copied over.
- Hence it is an expensive operation. If we are building a large list, we prefer extend method.

Sorting

You can use `sort` method to sort a list without creating a new object.

```
In [60]: a_ls = [8, 4, 9, 1, 0, 7]

In [61]: a_ls.sort()

In [62]: a_ls
Out[62]: [0, 1, 4, 7, 8, 9]
```

Sorting

You can sort a collection of strings by their length. This is possible because `sort` method has the ability to pass a secondary sort key.

```
In [63]: b_ls = ['fghi','ab','jklmn','cde','opqrst']

In [64]: b_ls.sort(key=len)

In [65]: b_ls
Out[65]: ['ab', 'cde', 'fghi', 'jklmn', 'opqrst']
```

Slicing

We can use slicing notation to select the section of most sequence types.

We can slice the list by passing the form `start:stop` to the index operator `[ ]`.

```
In [66]: seq = [1, 4, 6, 5, 3, 2, 7, 9, 8]

In [67]: seq[1:5]
Out[67]: [4, 6, 5, 3]
```

Please note that while the element at the `start` index is included in the slice, the `stop` index is not included.

Hence the number of elements in the results are `stop – start`.

Samatrix.io

Slicing

You can also assign a sequence to a slice

```
In [68]: seq[3:5] = [10, 11]

In [69]: seq
Out[69]: [1, 4, 6, 10, 11, 3, 2, 7, 9, 8]
```

We can either omit `start` or `stop` from the index operator. If we omit `start`, the default value which is `start` of the sequence is taken.

```
In [70]: seq[ : 4]
Out[70]: [1, 4, 6, 10]
```

Samatrix.io

Slicing

If we omit `stop`, the default value which is end of the sequence is taken

```
In [71]: seq[5 : ]
Out[71]: [3, 2, 7, 9, 8]
```

We can also use negative indexing.

The negative indexing means start from the end.

For example, `-1` refers to the last item. `-2` refers to the second last item.

Samatrix.io

Slicing

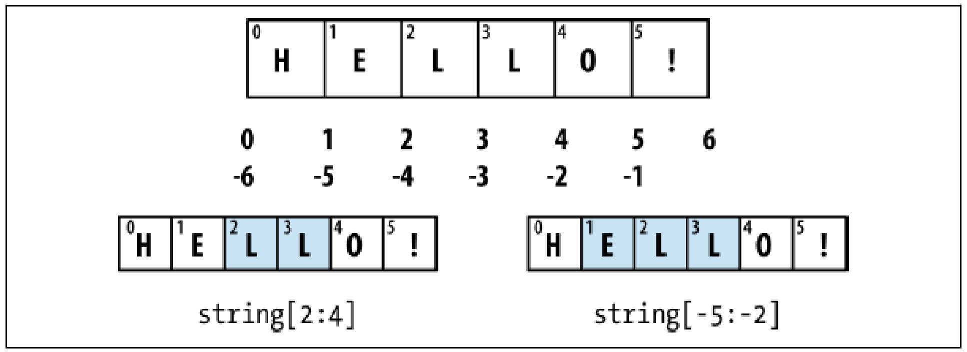
The following example searches the fourth element from the end (right side) till the end of the list.

```
In [72]: seq[-4:]
Out[72]: [2, 7, 9, 8]
```

We can also provide a range. For example, we can search sixth element from the end till second element (non-inclusive) from the end.

```
In [73]: seq[-6:-2]
Out[73]: [11, 3, 2, 7]
```

Slicing



If we want to skip the elements while slicing, we can provide a step after the second colon. For example, the following code takes every other element.

```
In [74]: seq[::2]
Out[74]: [1, 6, 11, 2, 9]
```

If we use the -1 value for the step, the list or tuple will be reversed.

```
In [75]: seq[::-1]
Out[75]: [8, 9, 7, 2, 3, 11, 10, 6, 4, 1]
```

Samatrix.io

Built in sequence function

- In this section, we will cover some of the very useful built-in sequence functions for python.

enumerate

When we work with iterators, we need to keep a track of the iterators. For example, view the following code

```
i = 0
for value in collection:
    #some statements
    i += 1
```

This is a common requirement for many programs.

To make the programmer’s task easy, python has a built-in function enumerate which returns a sequence of (`i, value`) tuple:

enumerate

```
for i, value in enumerate(collection):
    #some statements
```

In the example below, you can map the values in a `dict` to the value of the sequence

```
In [76]: a_list = ["abc", "def", "ghi"]

In [77]: mapping = {}

In [78]: for i, v in enumerate(a_list):
    ...:     mapping[v] = i
    ...:

In [79]: mapping
Out[79]: {'abc': 0, 'def': 1, 'ghi': 2}
```

Samatrix.io

sorted

To get new sorted list from the elements of any sequence, we used `sorted` method

```
In [80]: sorted([3, 6, 7, 1, 9, 2, 4, 8, 5])
Out[80]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [81]: sorted('learn python')
Out[81]: [' ', 'a', 'e', 'h', 'l', 'n', 'n', 'o', 'p', 'r', 't', 'y']
```

Samatrix.io

zip

Using `zip` method, you can pair up the elements of a number of `lists`, `tuples`, or other sequence. Output of the `zip` method is a list of `tuples`

```
In [82]: seq1 = ['abc', 'def' ,'ghi']

In [83]: seq2 = ['rst', 'uvw', 'xyz']

In [84]: zipped = zip(seq1, seq2)

In [85]: list(zipped)
Out[85]: [('abc', 'rst'), ('def', 'uvw'), ('ghi', 'xyz')]
```

Samatrix.io

zip

We can `zip` any number of sequences.

The number of elements it produces depends on the shortest sequences.

The `zip` returns the number of elements that equals the number of the elements in the shortest sequence.

```
In [86]: seq3 = [True, False]

In [87]: list(zip(seq1, seq2, seq3))
Out[87]: [('abc', 'rst', True), ('def', 'uvw', False)]
```

zip

A common use case of `zip` is to iterate over multiple sequences and possibly along with `enumerate`:

```
In [88]: for i, (a, b) in enumerate(zip(seq1, seq2)):
    ...:     print('{0}: {1}, {2}'.format(i, a, b))
    ...:
0: abc, rst
1: def, uvw
2: ghi, xyz
```

Samatrix.io

zip

You can also unzip the list of tuples if you want to separate the element of each tuple in independent sequence.

For this, you can use zip() along with unpacking operator *:

```
In [89]: pairs = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

In [90]: numbers, letters = zip(*pairs)

In [91]: numbers
Out[91]: (1, 2, 3, 4)

In [92]: letters
Out[92]: ('a', 'b', 'c', 'd')
```

Samatrix.io

reversed

The `reversed()` function returns the reversed iterator of the given sequence.

```
In [93]: list(reversed("Python"))
Out[93]: ['n', 'o', 'h', 't', 'y', 'P’]
```

You can also use `reversed()` in a for loop

```
In [94]: a_list = range(5)
In [95]: ra_list = reversed(a_list)
In [96]: for x in ra_list:
    ...:     print(x)
4
3
2
1
0
```

Please note that reversed is a generator. It can create the reversed sequence with list or a for loop.

dict

- Dictionaries are used to store data values in `key:value` pairs whereas `key` and `value` are python objects. The `dict` use curly braces `{ }` and colons to separate keys and values and they can be referred to by using the `key` name. The `dict` is a collection which is ordered, changeable and does not allow duplicates.
- Properties of `dict` are as follows
  - **Ordered**: As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered. When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change. Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

dict

- **Changeable:** Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- **Duplicates not allowed:** Dictionaries cannot have two items with the same key.
- **Dictionary length:** To determine how many items a dictionary has, use the `len()` function.
- **Dictionary Items - Data Types**: The values in dictionary items can be of any data type.

dict

To define `dict`, we can use curly braces `{ }` and colons to separate the `key` and `value`.

```
In [97]: d1 = {'a' : 'some string', 'b' : [1, 2, 3, 4]}

In [98]: d1
Out[98]: {'a': 'some string', 'b': [1, 2, 3, 4]
```

Samatrix.io

dict

To access, insert, or set elements in a dictionary, you can use the same syntax as for accessing the elements of a `list` or `tuple`

```
In [99]: d1[7] = 'an integer'

In [100]: d1
Out[100]: {'a': 'some string', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [101]: d1['b']
Out[101]: [1, 2, 3, 4]
```

Samatrix.io

dict

You can check if the dict contains a key using in keyword.

```
In [102]: 'b' in d1
Out[102]: True
```

In order to delete values, you can either use del keyword or pop method. pop method deletes the key and returns the value.

```
In [103]: d1[5] = 'Some Number'

In [104]: d1
Out[104]: {'a': 'some string', 'b': [1, 2, 3, 4], 7: 'an integer',
5: 'Some Number'}
```

Samatrix.io

dict

```
In [105]: d1['dummy'] = 'another number’

In [106]: d1
Out[106]:
{'a': 'some string',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'Some Number',
 'dummy': 'another number’}

In [107]: del d1[5]
```

Samatrix.io

dict

```
In [108]: d1
Out[108]:
{'a': 'some string',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another number'}

In [109]: ret = d1.pop('dummy')

In [110]: ret
Out[110]: 'another number'

In [111]: d1
Out[111]: {'a': 'some string', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

dict

You can use keys and values methods to get the iterators of dict’s keys and values. The keys and values are returned in the same order.

```
In [112]: list(d1.keys())
Out[112]: ['a', 'b', 7]

In [113]: list(d1.values())
Out[113]: ['some string', [1, 2, 3, 4], 'an integer']
```

dict

To merge one `dict` with another, you can use `update` method.

```
In [114]: d1.update({'b' : 'abc', 'c' : 7})

In [115]: d1
Out[115]: {'a': 'some string', 'b': 'abc', 7: 'an integer', 'c': 7}
```

If the key given in update method already exists, the value is updated else the key values would be inserted.

Samatrix.io

dict

The `dict` is a collection of 2-tuples, the `dict` function accepts a list of `2-tuples`.

```
In [116]: mapping = dict(zip(range(5), reversed(range(5))))

In [117]: mapping
Out[117]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Loop through dictionary

You can loop through a dictionary using a `for` loop.

When you loop through a dictionary, the return values are they `keys`.

```
In [118]: a_dict = {"brand" : "Kia","model" : "Seltos", "year" : 2019}

In [119]: for x in a_dict:
     ...:     print(x)
     ...:
brand
model
year
```

Samatrix.io

Loop through dictionary

You can also print the values, one by one

```
In [120]: for x in a_dict:
     ...:     print(a_dict[x])
     ...:
Kia
Seltos
2019
```

Samatrix.io

Loop through dictionary

You can also use the `values()` method to print the values, one by one

```
In [121]: for x in a_dict.values():
     ...:     print(x)
     ...:
Kia
Seltos
2019
```

Loop through dictionary

Use the `keys` method to return the keys

```
In [122]: for x in a_dict.keys():
     ...:     print(x)
     ...:
brand
model
year
```

Samatrix.io

Loop through dictionary

Use the items method to return both the `keys` and `values`

```
In [124]: for x, y in a_dict.items():
     ...:     print(x, y)
     ...:
brand Kia
model Seltos
year 2019
```

Samatrix.io

Set

A `set` is an unordered and immutable collection of unique elements.

They are like `dict` with only keys and without values.

Unordered means that the items in a `set` do not have a defined order.

`Set` items can appear in a different order every time you use them, and cannot be referred to by index or key.

You can create a `set` using two-methods:

- Using `set` functions
- Using `set` literal with curly braces

Set

```
In [124]: set([2, 2, 2, 1, 3, 3])
Out[124]: {1, 2, 3}

In [125]: {2, 2, 2, 1, 3, 3}
Out[125]: {1, 2, 3}
```

Using the sets, you can perform mathematical set operations such as `union`, `intersection`, `difference`, and `symmetric` difference.

Set

You can use `union` method to return a new set with all the items from both the sets. The `union` method will exclude any duplicate items

```
In [126]: a = {1, 2, 3, 4, 5}

In [127]: b = {3, 4, 5, 6, 7, 8}

In [128]: a.union(b)
Out[128]: {1, 2, 3, 4, 5, 6, 7, 8}
```

You can also use binary operator `|` for union

```
In [129]: a | b
Out[129]: {1, 2, 3, 4, 5, 6, 7, 8}
```

Samatrix.io

Set

You can also use `update` method that inserts all the items from one set into another.
Like `union` method, `update` method will also exclude duplicate values.
You can either use `update` method or `|=` operator

```
In [130]: a.update(b)

In [131]: a
Out[131]: {1, 2, 3, 4, 5, 6, 7, 8}

In [132]: a |= b

In [133]: a
Out[133]: {1, 2, 3, 4, 5, 6, 7, 8}
```

Set

The `intersection` method will return a new set, that only contains the items that are present in both sets. You can either use `intersection` method or `&` operator

```
In [134]: a.intersection(b)
Out[134]: {3, 4, 5}

In [135]: a & b
Out[135]: {3, 4, 5}
```

The `intersection_update()` method will keep only the items that are present in both set. You can also use `&=` operator in place of `intersection_update` method.

```
In [136]: a.intersection_update(b)

In [137]: a
Out[137]: {3, 4, 5}
```

Set

The `difference()` method will return a new set, that contains the elements in a that are not in b.

```
In [138]: a.difference(b)
Out[138]: {1, 2}
```

The `symmetric_difference()` method will return a new set, that contains only the elements that are NOT present in both sets.

```
In [139]: a.symmetric_difference(b)
Out[139]: {1, 2, 6, 7, 8}
```

Samatrix.io

Set

You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

```
In [140]: {1, 2, 3}.issubset(a_set)
Out[140]: True

In [141]: a_set.issuperset({1, 2, 3})
Out[141]: True
```

Sets are equal if and only if their contents are equal:

```
In [142]: {1, 2, 3} == {3, 2, 1}
Out[142]: True
```

Samatrix.io

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Without list comprehension you will have to write a `for` statement with a conditional test inside.

```
In [143]: animals = ["cow","dog","cat", "horse", "ox"]

In [144]: newlist = []

In [145]: for x in animals:
     ...:     if "o" in x:
     ...:         newlist.append(x)
     ...:

In [146]: print(newlist)
['cow', 'dog', 'horse', 'ox']
```

List Comprehension

With list comprehension you can do all that with only one line of code:

```
In [147]: animals = ["cow","dog","cat", "horse", "ox"]

In [148]: newlist = [x for x in animals if "o" in x]

In [149]: print(newlist)
['cow', 'dog', 'horse', 'ox']
```

The syntax of list comprehension becomes

```
newlist = [expression for item in iterable if condition == True]
```

Samatrix.io

List Comprehension

The `return` value is a new list that leaves the old list unchanged. In the following example, we have filtered the string with length 2 or more and converted them to uppercase

```
In [150]: strings = ['a', 'am', 'boy', 'girl', 'people']

In [151]: [x.upper() for x in strings if len(x) > 2]
Out[151]: ['BOY', 'GIRL', 'PEOPLE']
```

The `if` condition can also be omitted

```
In [152]: [x.upper() for x in strings]
Out[152]: ['A', 'AM', 'BOY', 'GIRL', 'PEOPLE']
```

Samatrix.io

List Comprehension

You can also set the outcome to whatever you want.

```
In [153]: ['hello' for x in strings]
Out[153]: ['hello', 'hello', 'hello', 'hello', 'hello']
```

You can also use expressions that are not filter like conditions. But they help you manipulate the outcome

```
In [154]: [x if x != 'am' else 'are' for x in strings]
Out[154]: ['a', 'are', 'boy', 'girl', 'people']
```

The expression in the above example says:
“Return the item if it is not am, if it is am return are”

Samatrix.io

Dict and set Comprehension

A `dict` comprehension looks like this

```
dict_comp = {key-expr : value-expr for value in collection if
condition}
```

The set comprehension looks like the list comprehension. Instead of square brackets, we use curly braces.

```
set_comp = {expr for value in collection if condition}
```

Samatrix.io

Dict and set Comprehension

In the following example, we have used `set` comprehension to get a set that contains the lengths of the strings contained in the collection

```
In [155]: uniqueLength = {len(x) for x in strings}

In [156]: uniqueLength
Out[156]: {1, 2, 3, 4, 6}
```

Dict and set Comprehension

The simple dict comprehension example would be to create a lookup map of the strings to their index in the list

```
In [157]: loc_mapping = {val : index for index, val in
enumerate(strings)}

In [158]: loc_mapping
Out[158]: {'a': 0, 'am': 1, 'boy': 2, 'girl': 3, 'people': 4}
```

Nested List Comprehension

Nested List Comprehensions are nothing but a list comprehension within another list comprehension which is quite similar to nested for loops.

Suppose we have a list of lists that contains English and Spanish names:

```
In [159]: name_data = [['John','Emily','Michael','Mary','Steven'],
     ...: ['Maria','Juan','Javier','Natalia','Pilar']]
```

Samatrix.io

Nested List Comprehension

Suppose we want a single list containing all names with two or more e’s in them. This could be achieved using simple `for` loop

```
names_2e = []
for names in name_data:
    two_es = [name for name in names if name.count('e') >= 2]
    names_2e.extend(two_es)
```

Samatrix.io

Nested List Comprehension

You can use nested list comprehensions in this case

```
In [160]: names2e = [name for names in name_data for name in names
if name.count(
    ...: 'e') >= 2]

In [161]: names2e
Out[161]: ['Steven']
```

The `for` parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before.

Samatrix.io

Nested List Comprehension

In the following example we have flatten a list of tuples of integers into a simple list of integers

```
In [162]: a_tuple = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [163]: flat = [x for tup in a_tuple for x in tup]

In [164]: flat
Out[164]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Samatrix.io

Functions

Functions

Functions are used for code organization and reuse in Python.

If there is a need to repeat same or very similar code more than one, it is recommended to write a reusable function.

Functions help you give a name to a group of Python statements.

Hence it makes your code more readable.

You can declare function using `def` function and return using `return` keyword. If Python interpreter does not encounter return till end of the function, None is returned.

Arguments

```
def a_function(a, b, c=2.0):
    if c > 1:
        return c * (a + b)
    else:
        return c / (a + b)
```

You can pass an information or data into functions using arguments.

You can specify the arguments after the function name and inside the parentheses.

You can add as many arguments as per your requirement.

You have to separate each argument with a comma.

Positional / Keyword Argument

Each function can either have positional arguments or keyword arguments.

We use keyword arguments to specify the default value or optional argument.

In the example above, `a` and `b` are positional argument whereas `c` is a keyword argument.

The keyword argument follows the `key = value` syntax

We can call a function by using the function name followed by parenthesis. Hence the above defined function can be called in any of these ways:

```
a_function(2, 3, c=1.5)
a_function(5, 6, c =0.8)
a_function(2.4, 1.6)
```

Samatrix.io

Positional / Keyword Argument

There are few restrictions on function arguments.

The keyword argument must follow the positional arguments (if any).

While calling the function, the positional argument (if not specified with keywords) should follow the same order as followed in function definition.

The keyword argument can follow any order.

If you use keywords for passing the positional arguments, the positional arguments can also follow any order

```
a_function(a = 2, b =3, c = 1)
a_function(b =3, a = 2, c = 1)
```

Arbitrary Argument

Arbitrary argument: If you are not sure of number of arguments that would be passed into the function, you can add a * before the parameter name in the function definition.

```
In [165]: def a_function(*fruit):
     ...:     print("My favourite fruit is " + fruit[2])
     ...:

In [166]: a_function("Banana", "Mango", "Orange")
My favourite fruit is Orange
```

Samatrix.io

Arbitrary Keyword Arguments

If you do not know how many keyword arguments that you can pass into function, you can add two `*` before parameter name in function definition.

This way the function will receive a dictionary of arguments and it can access the items accordingly

```
In [167]: def a_function(**fruit):
     ...:     print("My favourite fruit recipie is " + 
fruit["recipie"])
     ...:


In [168]: a_function(name = "Orange", recipie = "Juice")
My favourite fruit recipie is Juice
```

Please note that Arbitrary Arguments are often shortened to `*args` whereas Arbitrary Kword Arguments are often shortened to `**kwargs` in Python documentations

Samatrix.io

Namespace

- We have studied that for the assignment statements such as `x = 2`, create an integer object 2 and associate a symbolic name `x` with it.
- While working on any program, you will create several such names and point each one to a specific object.
- Now, we can define the **namespace** as a collection of names.
- You can think of namespace as a dictionary in which the keys are the object names and the values are the objects themselves.

Samatrix.io

Namespace

- In a Python program, generally there are three types of namespaces
  - **Built-in namespace:** contains the names of all of Python’s built-in objects. These are available as long as the Python is running. Example: `print(), len()`
  - **Global namespace:** contains any name that id defined at the level of the main program. These are created when the main program body starts and remains in existence until the interpreter terminates. The interpreter also creates a global namespace for any module that your program loads with the import statement.
  - **Local namespace:** These are created whenever a function is executed. The local namespace is local to the function and remains in existence until the function terminates.
- Each namespace has different lifetime. As Python executes a program, a namespace is created. It is deleted when it is no longer needed

Scope

- Due to the existence of multiple, distinct namespaces, several different instances of a particular name can exist simultaneously when a Python program runs.
- These namespaces are maintained separately and they do not interfere with each other.
- Suppose you have several name `x` in your codes whereas `x` exists in several namespace. How Python decides, which one it means.
- This could be understood using the concepts of **scope**.
- Scope of a name is the region of a program in which that name has meaning. Functions can access variables in two different scopes: global and local

Local Scope

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

```
In [169]: def myfunc():
     ...:     a = 50
     ...:     print (a)
     ...:

In [170]: myfunc()
50
```

When the function `myfunc()` is called, object `50` is created and assigned to name `a`. Then `a` is destroyed when the function exits.

Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope and can be used by anyone. Global variables are available from within any scope, global and local.

```
In [171]: a = 50

In [172]: def myfunc():
     ...:     print(a)
     ...:

In [173]: myfunc()
50

In [174]: print(a)
50
```

Samatrix.io

Global Scope

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function)

```
In [175]: a = 50
In [176]: def myfunc():
     ...:     a = 30
     ...:     print(a)
     ...:

In [177]: myfunc()
30
In [178]: print(a)
50
```

Global Variable

If you use the global keyword, the variable belongs to the global scope

```
In [179]: a = 50

In [180]: def myfunc():
     ...:     global a
     ...:     a = 30
     ...:     print(a)
     ...:

In [181]: myfunc()
30

In [182]: print(a)
30
```

Samatrix.io

Returning Multiple Variable

One of the functionalities of Python is its ability to return multiple values from a function.

```
In [183]: def x():
     ...:     a = 5
     ...:     b = 5
     ...:     c = 7
     ...:     return a, b, c
     ...:

In [184]: d, e, f = x()

In [185]: d
Out[185]: 5
```

This functionality of Python is very useful for data analysis. Actually, the function is returning only one object, tuple, which has been unpacked into the result variable.

Returning Dict

If we assign the function to a variable called `return_variable(),` the `return_variable()` would be a `tuple` with three elements.

You may also return a `dict`

```
In [186]: def x():
     ...:     a = 5
     ...:     b = 6
     ...:     c = 7
     ...:     return{'a' : a, 'b' : b, 'c' : c}
```

Samatrix.io

Functions are Objects

In Python functions are first class objects. It means that the function in Python has following properties

- Functions have types
- Functions can be sent as arguments to another function
- Functions can be used in expression
- Function can be a part of various data structures such as list or dictionaries

This gives you the capability to do some things in Python that are difficult-to-impossible to carry out in many other languages.

Functions are Objects

To test this, let's define a simple function called `answer()` that doesn't have any arguments; it just prints the number `42`:

```
In [187]: def answer():
     ...:     print(42)
     ...:
```

If you run this function, you know that you will get the following

```
In [188]: answer()
42
```

Samatrix.io

Functions are Objects

Let’s define another function named `do_something()`. It has only any arguments called `func`, a function to run. It just calls the function:

```
In [189]: def do_something(func):
     ...:     func()
     ...:
```

If we pass answer to `do_something()`, we are using a function as data, just as with anything else:

```
In [190]: do_something(answer)
42
```

Samatrix.io

Functions are Objects

Notice that we passed `answer`, not `answer().`

In Python, the parentheses mean call the function.

Without parentheses, Python treats the function like any other object.

The reason is that everything in Python is an object.

We can try running a function with arguments.

Let's define a function `add_args().` It prints the sum of two numeric arguments `arg1` and `arg2`.

```
In [191]: def add_args(arg1, arg2):
     ...:     print(arg1 + arg2)
     ...:
```

Functions are Objects

We define another function `do_something_args()`. This function takes three argumemts

```
In [192]: def do_something_args(func, arg1, arg2):
     ...:     func(arg1, arg2)
     ...:
```

When you call `do_something_args()`, the function passed by the caller is assigned to the func parameter, whereas `arg1` and `arg2` get the values that follow in the argument list.

Samatrix.io

Functions are Objects

Then, running `func(arg1, arg2)` executes that function with those arguments because the parentheses told Python to do so.

Let's test it by passing the function name `add_args` and the arguments 5 and 9 to `do_something_args()`

```
In [193]: do_something_args(add_args, 5, 9)
14
```

Anonymous (lambda) function

The keyword `lambda` is used in Python to define anonymous functions, that is; functions without a name and described by a single expression.

You might just want to perform an operation on a function that can be expressed by a simple expression without naming this function and without defining this function by a lengthy `def` block.

The syntax of `lambda` function is

```
lambda arguments : expression
```

Anonymous (lambda) function

The expression is executed and the results are returned

For example, if we want to write a function to add 10 to the argument a and return the result

```
In [194]: x = lambda a : a + 10

In [195]: print(x(5))
15
```

Anonymous (lambda) function

Lambda functions can take any number of arguments.

For example, if we have to multiply two arguments and return the result

```
In [196]: x = lambda a, b : a * b

In [197]: print(x(3, 4))
12
```

Use of lambda function in Python

- Lambda functions are used when we need a nameless function for a short period of time.
- Generally, the lambda functions are used as an argument to a higher-order function (a function that takes in other functions as arguments).
- We can use the lambda function with built-in functions such as `map()` and `filter()`.

Use of lambda function with filter()

The `filter()` function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Here is an example use of `filter()` function to filter out only even numbers from a list.

```
In [198]: my_list = [1, 5, 4, 6, 7, 8, 11, 12]

In [199]: new_list = list(filter(lambda x : (x%2 ==0), my_list))

In [200]: print(new_list)
[4, 6, 8, 12]
```

Samatrix.io

Use of lambda function with map()

The `map()` function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of `map()` function to double all the items in a list.

```
In [201]: my_list = [1, 5, 4, 6, 7, 8, 11, 12]

In [202]: new_list = list(map(lambda x: x * 2, my_list))

In [203]: print(new_list)
[2, 10, 8, 12, 14, 16, 22, 24]
```

Samatrix.io

Error and Exception Handling

Exceptions

- An “exception” is usually defined as “something that does not conform to the norm,” and is therefore somewhat rare.
- There is nothing rare about exceptions in Python. They are everywhere.
- Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.
- To building robust program, you should handle the Python errors and exceptions gracefully.
- In data analysis, many functions are capable of working on certain kind of input only.
- For example, Python’s float function is used to cast a string to a floating-point number but if we input a string to float function, it will fail with ValueError.

Exceptions

```
In [204]: float('1.2345')
Out[204]: 1.2345

In [205]: float('number')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-205-4a5bbf889db2> in <module>
----> 1 float('number')

ValueError: could not convert string to float: 'number'
```

Samatrix.io

Try Except Block

Suppose if want that the program should return float if the casting is possible else return the input argument itself.

We can do so by using a try/except block.

```
In [206]: def try_float(x):
     ...:     try:
     ...:         return float(x)
     ...:     except:
     ...:         return x
```

Try Except Block

The code in except part of the block will be executed if `float(x)` raises an exception.

```
In [207]: try_float('1.2345')
Out[207]: 1.2345

In [208]: try_float('number')
Out[208]: 'number'
```

Try Except Block

In the previous example, the float raised `ValueError` exception. But other exceptions than `ValueError` are also possible.

```
In [209]: float((1,2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-209-a101b3a3d6cd> in <module>
----> 1 float((1,2))

TypeError: float() argument must be a string or a number, not 'tuple'
```

Samatrix.io

Try Except Block

In your program, the TypeError, may represent a legitimate bug, hence you may want to suppress the ValueError only. In that case, you can write exception type after except.

```
In [210]: def try_float(x):
     ...:     try:
     ...:         return float(x)
     ...:     except ValueError:
     ...:         return x
     ...:

In [211]: float((1,2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-211-a101b3a3d6cd> in <module>
----> 1 float((1,2))

TypeError: float() argument must be a string or a number, not 'tuple'
```

Samatrix.io

Try Except Block

You may catch multiple exception types by writing a tuple of exception types in parentheses.

```
In [212]: def try_float(x):
     ...:     try:
     ...:         return float(x)
     ...:     except (TypeError, ValueError):
     ...:         return x
     ...:

In [213]: try_float((1,2))
Out[213]: (1, 2)
```

Samatrix.io

Else Block

The try except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
In [214]: def divide(x, y):
     ...:     try:
     ...:         result = x / y
     ...:     except ZeroDivisionError:
     ...:         print ("Cannot divide a number by zero")
     ...:     else:
     ...:         print("Division Successful! Answer is: ",result)
In [215]: divide(5, 2)
Division Successful! Answer is:  2.5

In [216]: divide(5, 0)
Cannot divide a number by zero
```

Finally Block

Python provides a keyword finally, which is always executed after try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception.

```
In [217]: def divide(x, y):
     ...:     try:
     ...:         result = x / y
     ...:     except ZeroDivisionError:
     ...:         print ("Cannot divide a number by zero")
     ...:     else:
     ...:         print("Division Successful! Answer is: ",result)
     ...:     finally:
     ...:         print("Division Operation Complete")
```

Samatrix.io

Finally Block

```
In [218]: divide(5, 2)
Division Successful! Answer is:  2.5
Division Operation Complete

In [219]: divide(5, 0)
Cannot divide a number by zero
Division Operation Complete
```

Samatrix.io

Files and Operating System

Samatrix.io

File Handling

File handling is an important part of any data analytics application.

There are several functions for creating, reading, updating, and deleting the files in Python.

The primary function to work with file is `open()` function.

The `open()` function takes two parameters: `filename` and `mode`.

File Handling

The four different modes (methods) of opening a file are:

`"r"` – Read – Default value. It opens the file for reading. Gives error if file does not exist

`"a"` – Append – Opens the file for appending. It creates the file if the file does not exist

`"w"` – Write – Opens a file for writing. It creates the file if the file does not exist

`"x"` – Create – It creates the specified file. It returns an error if the file exists

In addition, you may specify the following options

`"t"` – Text – Default value, For Text mode

`"b"` – Binary – Binary mode for images

File Handling - Reading

To open a file for reading, you can specify the following syntax

```
In [220]: f = open("filetest.txt")
```

This code is same as

```
In [221]: f = open("filetest.txt", "rt")
```

Because the `“r”` for read and `“t”` for text are the default values.

Samatrix.io

File Handling - Reading

The `open()` function returns a file object that has a `read()` method for reading the content of the file

```
In [222]: f = open("filetest.txt", "r")

In [223]: print(f.read())
An “exception” is usually defined as “something that does not
conform to the norm,” and is therefore somewhat rare.
There is nothing rare about exceptions in Python.
They are everywhere.
```

Samatrix.io

File Handling - Reading

By default, the `read()` method returns the whole text.
But you can specify the characters that you want to return.
For example, if you want to return first 15 characters of the file.

```
In [224]: f = open("filetest.txt")

In [225]: print(f.read(15))
An “exception”
```

File Handling - Reading

You can return one line by `readline()` method.

```
In [226]: f = open("filetest.txt", "r")

In [227]: print(f.readline())
An "exception" is usually defined as "something that does not
conform to the norm," and is therefore somewhat rare.
```

File Handling - Reading

You can use the following to read two lines of the file

```
In [228]: f = open("filetest.txt", "r")

In [229]: print(f.readline())
An “exception” is usually defined as “something that does not
conform to the norm,” and is therefore somewhat rare.

In [230]: print(f.readline())
There is nothing rare about exceptions in Python.
```

Samatrix.io

File Handling - Reading

You can use `for` loop to read the whole file

```
In [231]: f = open("filetest.txt", "r")

In [232]: for x in f:
     ...:     print(x)
     ...:
An “exception” is usually defined as “something that does not conform to the norm,” and is
therefore somewhat rare.
There is nothing rare about exceptions in Python.
They are everywhere.
```

You should close the file when you are done with it

```
In [233]: f.close()
```

Samatrix.io

File Handling - Writing

If you want to write to an existing file, you may use the following parameter

`"a"` – Append – Opens the file for appending. It creates the file if the file does not exist

`"w"` – Write – Opens a file for writing. It creates the file if the file does not exist

To append to a file, you can open a file and append the content to the file by passing `"a"` as second parameter

```
In [234]: f = open("filetest.txt", "a")

In [235]: f.write("Now we have added more content!")
Out[235]: 31

In [236]: f.close()
```

File Handling - Writing

To append to a file, you can open a file and append the content to the file by passing "a" as second parameter

```
In [234]: f = open("filetest.txt", "a")


In [235]: f.write("Now we have added more content!")
Out[235]: 31


In [236]: f.close()


In [237]: f = open("filetest.txt")


In [238]: print(f.read())
An “exception” is usually defined as “something that does not conform to the norm,” and is
therefore somewhat rare.
There is nothing rare about exceptions in Python.
They are everywhere.Now we have added more content!
```

Samatrix.io

File Handling - Writing

You can also open the file and overwrite its content

```
In [239]: f = open("filetest.txt", "w")

In [240]: f.write("Sorry, the content has been deleted")
Out[240]: 35

In [241]: f.close()

In [242]: f = open("filetest.txt")

In [243]: print(f.read())
Sorry, the content has been deleted
```

Samatrix.io

File Handling - Creating

To create a file called `myfile.txt`

```
In [244]: f = open("myfile.txt",'x')
```

To create a new file called `myfile.txt` if it does not exist

```
In [245]: f = open("myfile1.txt",'w')
```

Samatrix.io

Delete File and OS Commands

To delete a file, you must import the OS module, and run its `os.remove()` function:

To remove the file `myfile.txt`

```
In [246]: import os

In [247]: os.remove("myfile.txt")
```

Delete File and OS Commands

Before deleting the file, you may want to check if the file exists so that you can avoid error

```
In [248]: import os

In [249]: if os.path.exists("myfile1.txt"):
     ...:     os.remove("myfile1.txt")
     ...: else:
     ...:     print("The file does not exist")
```

Samatrix.io

Thanks

Samatrix Consulting Pvt Ltd

Samatrix.io