Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier can start with an alphabet (capital or small) or an underscore (_) only. It may consist of alphabets, digits (0 to 9) or underscores. Following points are to be noted about identifiers in Python.

1. The identifiers are case sensitive.
2. Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
3. Starting an identifier with a single leading underscore indicates that the identifier is private. It is just by convention and python interpreter does not enforce this convention.
4. Starting an identifier with two leading underscores indicates a strongly private identifier. Python interpreter applies name mangling to these attributes.
5. If the identifier also ends with two trailing underscores, the identifier is a language-defined special name. You are recommended to never use this pattern for your own code.
6. A single underscore can be appended to an identifier to avoid it getting in conflict with Python language keywords. For example an identifier name like class is not acceptable whereas class_ will work.
7. A single lone underscore (_) points to the result of the last executed statement in an interactive interpreter session. Also used as a throwaway name (a certain name is assigned but not intended to be used) by convention.

   Read this article about the significance of underscores in Python.
   https://shahriar.svbtle.com/underscores-in-python

Blocks of Code and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Or all the statements at the same indentation level are part of the same block.

Statement Suites

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

```
if(input < 48):
      print('The input is less than 48')
      some_function(48)
elif(input >= 200):
      print('The input is more than 200')
      another_function(200)
```

Input and Output

The function `input()` reads and stores input from STDIN. The `input()` function can also be called with a text message. The message prompts the user for input.

```
var = input()
var = input('Enter your name')
```

The input is always read as a string. You need to use `int()` or `float()` to convert a string to an integer or float respectively. If the number is not convertible `NameError` will be thrown.

The `print`() function prints data to STDOUT.

```
output = 45
print('The output is =', output)
```

You can separate the output using the comma delimiter. By default, this adds a space between the output items. By default `print()` appends a newline at the end of the output. You can use `sep` and `end` arguments to change the separation and end character.

```
a = 'quick fox'
b = 23
c = 2.3
print(a, b, c)
print(a, b, c, sep=',')
print(a, b, c, sep=',', end='@\n')
```

```
'quick fox' 23 2.3
'quick fox',23,2.3
'quick fox',23,2.3@
```

Operators in Python

Python language supports the following types of operators: Arithmetic Operators, Comparison (Relational) Operators, Assignment Operators, Logical Operators, Bitwise Operators and Identity Operators. These are listed below.

Arithmetic Operators

+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b

%	Modulus. Divides left hand operand by right hand operand and returns remainder	a % b
**	Exponent	a ** b
//	Floor division. Divides left hand operand with right hand operand, rounds off the result to the closest but lower integer	9//4 will yield 2 and -9/4 will yield -3.

Note: Unlike other languages Python do not have ++ and -- operators.

Comparison Operators

==	Returns True if both operands are equal.	`a == b`
!=	Returns True if both operands are not equal.	`a != b`
>	Returns true if left operand is more than right operand.	`a > b`
<	Return true if right operand is more than left operand.	`a < b`
>=	Return true of left operand I more than or equal to right operand.	`a > = b`
<=	Returns true if left operand is less than or equal to right operand.	`a <= b`
`is`	Object comparison. Evaluates to true if the names on either side of the operator point to the same object and false otherwise.	`x is y` essentially checks the identity (as returned by the `id()` function) of the two objects – `x` and `y`.
`is not`	Opposite of above.	Opposite of above.

Comparisons can be chained arbitrarily. For example, `x < y <= z` is equivalent to `x < y and y <= z`. This is unlike other programming languages.

Assignment Operators

=	Assigns values from right side operands to left side operand	`c = a + b` assigns value of `a + b` into `c`
+=	It adds right operand to the left operand and assign the result to left operand	`c += a` is equivalent to `c = c + a`
-=	It subtracts right operand from the left operand and assign the result to left operand	`c -= a` is equivalent to `c = c - a`
*=	It multiplies right operand with the left operand and assign the result to left operand	`c *= a` is equivalent to `c = c * a`
/=	It divides left operand with the right operand and assign the result to left operand	`c /= a` is equivalent to `c = c/a`
%=	It takes modulus using two operands and assign the result to left operand	`c %= a` is equivalent to `c = c % a`
**=	Performs exponential (power) calculation on operators and assign value to the left operand	`c **= a` is equivalent to `c = c ** a`
//=	It performs floor division on operators and assign value to the left operand	`c //= a` is equivalent to `c = c // a`

Multiple Assignments

Single value can be assigned to multiple variables in a single statement. The first statement creates an object of type integer and binds it with three different names (a, b and c), whereas the second statement creates 3 different integer objects and binds them to 3 different names.

```
a = b = c = 48
a, b, c = 23, 45, 93
```

Bitwise Operators

In the following examples assume that a = 60 (00111100) and b = 13 (00001101)

& Binary AND	Operator copies a bit to the result if it exists in both operands	a & b = 12 (0b0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (0b0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (0b0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (-0b11110100)
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (0b1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (0b0000 1111)

Logical Operator

In the following examples assume that a = True and b = False.

and Logical AND	If both operands are true then condition becomes true.	(a and b) is False.
or Logical OR	If any of the two operands are true then condition becomes true.	(a or b) is True.
not Logical NOT	Used to reverse the logical state of its operand.	not (a and b) is True.

Membership Operator

Membership operators test for membership in a sequence, such as strings, lists, or tuples. Membership operators are also applicable to dictionaries and sets. If applied over a dictionary it checks for the existence of the given element in the keys of the dictionary.

in	Evaluates to True if it finds a variable in the specified sequence and False otherwise.

`not in`	Evaluates to `True` if it does not find a variable in the specified sequence and `False` otherwise.

Truth value of Objects

In Python any object can be tested to be `True` or `False` for use in an `if` or `while` or boolean operations. The following objects are considered `False`.

1. `None` (built in object denoting absence of information).
2. `False`.
3. Zero of any numeric type – `0, 0.0` and `0j`.
4. Any empty sequence – `''` (empty string), `()` (empty tuple), `[]` (empty list).
5. Empty mappings – `{}` (empty dictionary).

All other objects are considered to be `True`. Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. But the Boolean operations `or` and `and` always return one of their operands. In case of mixed operand types the result could be any one of the inputs.

Python Built in Data Types

The Python built in data types can be categorized in the following categories:

1. Numeric Types (integers, float and complex),
2. Sequence Types (String, List, Tuple and Range),
3. Mapping Types (Dictionary), and
4. Set Types (Sets and Frozensets).

Numeric Types

There are three distinct numeric types: integers, floating point numbers, and complex numbers. In addition, booleans are a subtype of integers.

In Python, **value of a numeric type is not restricted by the number of bits and can expand to the limit of the available memory**. Thus we never need any special arrangement for storing large numbers. See the code below.

```
a = 100**100
print(a)
```

```
10000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
```

Floating point numbers are implemented using double in C. Complex numbers have a real and imaginary part, which are each implemented using double in C.

Floating point numbers and integers support arithmetic, comparison, assignment and logical operations. Integers support bitwise operations too. See the list of operators for the syntax of these operations.

Python fully supports mixed arithmetic. **When a binary arithmetic operator has operands of different numeric types, the operand with the narrower type is widened to that of the other**, where integer is narrower than floating point, which is narrower than complex.

When the type of the input numbers are same the result will be a number of the same type. **Except the case of division**. In case of division, division of two integers will always result in a float. This is not what happens in other programming languages like Java or C.

```
result = 3/2
type(result)
```

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

Examples of Numeric Types

1. Integer: `48, -48, 0x260, -0x260, 0o41, -0o131, 0b111001000, -0b11100110`.
1. Float: `3.9, -45.6, 32.3E18, 23.19E-3`
2. Complex: `2j, 2.3j, 3 + 4j`

Sequence Types

Sequence is the generic term for an ordered set. There are six sequence types - **strings, byte sequence** (bytes objects), **byte array** (bytearray objects), **lists, tuples, and range**. Strings, lists, tuples and range are the most frequently used.

Strings

Strings are a special type of sequence that can only store characters. Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters. However, **Python does not have a character data type**, a single character is simply a string with a length of one. Strings are immutable (once created they cannot be modified). String objects can be created by string literals or by calling built-in functions like `str()`.

```
var = "quick brown fox"
var = 'jumps over the lazy dogs'
```

You can embed single quote within a double quoted string and double quote within a single quoted string. Backward slash can be used to escape quotes. Triple quotes can be used for strings spanning multiple lines.

```
var = "quick 'brown' fox"
```

```
print(var)
var = 'jumps "over" the lazy dogs'
print(var)
var = "quick \"brown\" fox"
print(var)
var = '''quick brown fox
jumps over the
dogs'''
print(var)
```

```
quick 'brown' fox
jumps "over" the lazy dogs
quick "brown" fox
quick brown fox
jumps over the
dogs
```

Lists

Lists are the most versatile sequence type, which can be written as a list of comma-separated values (items) between square brackets. Important thing about the list is that items in a list need not be of the same type. Lists are mutable (they can be changed). Lists can be constructed in many ways.

Using square brackets with each item separated by a comma. Empty square brackets can be used to create empty lists.

```
my_list = ['string', 1, 2.34]
empty_list = []
print(my_list)
print(my_list[1])
print(type(my_list[2]))
print(type(my_list[1]))
print(empty_list)
```

```
['string', 1, 2.34]
1
<class 'float'>
<class 'int'>
[]
```

Using the type constructor – `list(iterable)`. The constructor builds a list whose items are the same and in the same order as the items of `iterable`. The `iterable` may be either a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a list, a copy is made and returned. The constructor can be called with no arguments to create an empty list.

```
my_string = 'python'
my_list = list(my_string)
```

```
print(my_list)
my_list = list(range(5))
print(my_list)
```

```
['p', 'y', 't', 'h', 'o', 'n']
[0, 1, 2, 3, 4]
```

Using list comprehensions. List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition. Using list comprehension we can easily replace nested loops, lambda functions and map function. Note that you can write a for-loop for every list comprehension, but you cannot write list comprehensions for very complex for-loop.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more for or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. Given below is the basic syntax for list comprehension followed by examples. The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

```
new_list = [expression for_clause [multiple for or if clauses]]
```

```
# List of squares from 0 to 16
squares = [x**2 for x in range(5)]
print(squares)

# List of common numbers of two different lists.
list_a = [1, 2, 3, 9, 18]
list_b = [0, 2, 8, 9, 28]
common = [a for a in list_a for b in list_b if a == b]
print(common)

# Prints numbers from 11 to 19
my_list = [i for i in range(100) if i > 10 if i < 20]
print(my_list)

# Return numbers from the list which are not equal as a tuple
list_a = [1, 2, 3]
list_b = [0, 2, 8]
different = [(a, b) for a in list_a for b in list_b if a != b]
print(different)

# Apply a function to a list (similar to map())
my_list = ['quick', 'brown', 'fox', 'jumps']
modified = [str.upper() for str in my_list]
print(modified)
```

```
[0, 1, 4, 9, 16]
[2, 9]
```

```
[11, 12, 13, 14, 15, 16, 17, 18, 19]
[(1, 0), (1, 2), (1, 8), (2, 0), (2, 8), (3, 0), (3, 2), (3, 8)]
['QUICK', 'BROWN', 'FOX', 'JUMP']
```

Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The major differences between tuples and lists are - tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses.

```
tuple1 = 'name', 'address', 45, 23.6
tuple2 = ('name', 'address', 45, 23.6)
```

An empty tuple is written as two parentheses containing nothing and a single item tuple must have a trailing comma.

```
tuple3 = ()
tuple4 = (213.6,)
```

Why do we need Tuples, when we have Lists?

Immutable objects allow substantial optimization. For the same amount of data tuples take less space than lists. See example below

```
import sys
tuple1 = ('quick', 'brown', 'fox', 'jumps')
list1 = ['quick', 'brown', 'fox', 'jumps']
sys.getsizeof(tuple1)
sys.getsizeof(list)
```

```
44
52
```

Operations with tuples are much faster than operations with lists. The performance difference can be partially measured using the `timeit` library which allows you to time your Python code. The code below, which simply creates a tuple and a list consisting of exactly the same elements, is run 1 million times and its total time of execution is measured using the `timeit` library.

```
import timeit
timeit.timeit('x=(1,2,3,4,5,6,7,8,9,10,11,12)', number=1000000)
timeit.timeit('x=[1,2,3,4,5,6,7,8,9,10,11,12]', number=1000000)
```

```
0.02018076300737448
0.1307151880027959
```

Range

The range type is an immutable sequence which is commonly used for looping. The advantage of the range type is that a range object will always take the same amount of memory, no matter the size of the range it represents. Range objects have very little behavior – they only support indexing, iteration, and the `len()` function. And they do not support slicing, concatenation or repetition.

Objects of type range are created using the `range()` function. This is a versatile function to create iterables yielding arithmetic progressions. It is most often used in `for` loops. Following is the usage of `range()` function.

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

The arguments must be integers. The default values for `step` and `start` are `1` and `0` respectively. The full form returns an iterable of integers `[start, start + step, start + 2 * step, ...]`. If `step` is positive the last element is closest to `stop` but less than or equal to `stop`. If the `step` is negative the last element will be closest to `stop` but more than or equal to `stop`. `Step` must not be zero (or else `ValueError` is raised). See the examples below.

`range(4, 16, 3)`	Generates the sequence: 4, 7, 10, 13
`range(4, 16, -3)`	Makes no sense. Will not generate anything
`range(16, 4, -3)`	Generates the sequence: 16, 13, 10, 7
`range(4, 16, -3)`	Makes no sense. Will not generate anything.
`range(10)`	Generates 0, 1, 2, ... 8, 9
`range(-5)`	Makes no sense. Will not generate anything

Since range type of sequences support only those items that follow specific patterns, hence range sequences do not support slicing, concatenation or repetition. And using `in`, `not in`, `min()` or `max()` on them is very inefficient.

Operations Applicable to All Sequences

Following set of operations are applicable to all types of sequences (mutable or not).

Inclusion Check

The operator `in` returns `True` if the given item is part of the given sequence and vice-versa. The operator `not in` returns `True` if the given item is not part of the given sequence and vice-versa. If this is applied over a string this operation is equivalent to substring test.

```
string = 'healthy wealthy and wise'
my_list = ['wise', 48, 29.3]
substring = 'health'
print(substring in string)
print(substring not in string)
print(substring  in my_list)
```

```
True
```

```
False
False
```

Concatenation

The operators + and * concatenates given sequences. The asterisk operator creates a **shallow copies** of the sequence and appends them to the sequence.

```
str_one = 'hello'
str_two = 'python'
print(str_one + str_two)
print(str_one * 3)
```

```
hellopython
hellohellohello
```

Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. If we are dealing with strings, it is recommended to use `str.join(seq)` method instead, which assures consistent linear concatenation performance across versions and implementations. A `TypeError` will be raised if the argument is non string type.

```
str_one = 'hello'
str_two = 'python'
print(str_one.join(str_two))
```

```
phelloyhellothellohhelloohellon
```

Element Access

Using square brackets and 0 based indexing individual elements can be accessed.

```
tuple1 = (‘health’, 48, 23.9, ‘wealth’)
print(tuple1[2])
```

```
23.9
```

Slicing

Getting sub-sequences is called slicing. Subsequences from `start` to `end` (not included) with a step of `step` can be obtained by using square brackets and colon operator. The general format is `seq[start:end:step]`. The default value of `start`, `end` and `step` are `0`, `len(seq)` and `1` respectively. **Negative indices are counted from the end and are not 0 based. That is, -3 is actually the 3rd last character**. See examples below assuming the `string` is = `'healthy wealthy and wise'`. **It is to be noted that slicing returns an altogether new sequence.**

`string[3:6]`	`lth`	Substring from 3rd index to 5th index.

`string[3:]`	`lthy wealthy and wise`	Substring from 3rd index to the last.
`string[:6]`	`health`	Substring from start to the 5th index.
`string[6:-3]`	`y wealthy and w`	Substring from 6th index to the 4th last character.
`string[-1:6]`		Makes no sense. Will not print anything.
`string[-15:-3]`	`ealthy and w`	Substring from 15th last to 4th last character.
`string[2:10:3]`	`ahw`	Substring consisting of every 3rd character, starting from 2nd and closest to 10th character but never including it.
`string[-1:3:-3]`	`ewnylwh`	Starting from the last character moves towards the character at 3rd index, picking every 3rd character. Moves closest to the character at third index but does not include it (last character is not included)

Length, Minimum and Maximum

The methods to obtain length, minimum and maximum values from a sequence are `len(seq)`, `min(seq)` and `max(seq)`. The `min` and `max` methods are applicable only when the items in the sequence are homogeneous, otherwise `TypeError` is thrown. Characters and strings are compared lexicographically. Integers and floats are compared using their values. Arbitrary objects are compared using their `__eq__` method (if the class has implemented the method properly).

Element Location and Count

To look for the index of first occurrence of a given item we can use `seq.index(item)` method. This method returns the first occurrence of the `item` in the sequence. If the `item` is not found in the list the method throws `ValueError`. The method `seq.count(item)` returns the count of `item` in the given sequence.

Unpacking

Python provides for a powerful feature called unpacking, wherein the elements of any iterable can be assigned to a set of variables. Please note that this feature is available to all iterables (not just sequences). See the below code to understand. If the number of variables on the left hand side is not equal to the values to be unpacked `ValueError` is raised.

```
my_list = [24, 'quick fox', 24.9]
my_string = 'adt'

# 24, 'quick fox' and 24.9 will be assigned to x, y and z respectively.
x, y, z = my_list
print(x, y, z)

# 'a', 'd' and 't' will be assigned to x, y and z respectively.
x, y, z = my_string
print(x, y, z)
```

```
24 quick fox 24.9
a d t
```

Asterisk operator (*) can be used to unpack an iterable with an unspecified number of elements. See the example below.

```
my_list = (24, 'quick', 'fox', 24.9)
my_string = 'jumping jack'

# 24 and 'quick' will be assigned to x and y respectively
# and other elements will be assigned to z.
x, y, *z = my_tuple
print(x, y, z)

# 24 and 24.9 will be assigned to x and z respectively
# and other elements will be assigned to y.
x, *y, z = my_tuple
print(x, y, z)

# 'j' and 'k' will be assigned to a and z respectively
# and others will be assigned to y.
x, *y, z = my_string
print(x, y, z)
```

```
24 quick ['fox', 24.9]
24 ['quick', 'fox'] 24.9
j ['u', 'm', 'p', 'i', 'n', 'g', ' ', 'j', 'a', 'c'] k
```

Reference:

1. https://treyhunner.com/2018/03/tuple-unpacking-improves-python-code-readability/
2. https://www.geeksforgeeks.org/unpacking-a-tuple-in-python/

Operations Applicable to Mutable Sequences

List is a mutable type sequence, whereas string and tuples are immutable type. Following operations are applicable to mutable type sequences. In all the following examples positive indices are 0 based and are counted from the start, whereas negative indices are 1 based and are counted from the end.

`seq[i] = x`	Item at i^{th} index is replaced by `x`.
`del seq[i:j]` `seq[i:j] = []`	Removes the slice from `i` to `j`.
`del seq`	Removes the complete sequence.
`seq.append(x)`	Appends the element `x` to the `seq`.
`seq.extend(seq2)`	Adds all the items of `seq2` to `seq`.
`seq[i:j] = seq2`	Replaces the specified slice with the contents of `seq2`. It actually deletes the slice from `i` to `j` and then inserts the contents of `seq2` in the same position.
`seq[i:j:k] = seq2`	Replaces the specified slice with the contents of `seq2`. The length of `seq2` should be exactly equal to the length of the slice being replaced. With `k = 1`, the statement becomes

	equivalent to the previous one and hence the condition of equality is not required.
`seq.insert(i, x)`	Inserts the item `x` at the `ith` index. Out of bound values of `i` are replaced with `len(seq) - 1` or `0`, whichever is closest.
`seq.pop(i)`	Returns the `ith` item and deletes it from the sequence. If `i` is omitted it pops the last item.
`seq.remove(x)`	Deletes the `x` from the list. If `x` is not found it throws `ValueError`.
`seq.reverse()`	Reverses the items of `seq` in place.
`seq.sort(reverse=True|False, key=my_func)`	Sorts the given sequence. The default value of reverse is `False`. `my_func` is the custom comparison function. The default value of key does lexicographical comparison with strings and value comparison with integers and floats. If default value of key is used the sequence should be homogeneous otherwise `TypeError` will be thrown.

Other Functions Applicable to Strings Only

There are many useful functions applicable to strings only. For a complete list of all the string functions see this resource: https://docs.python.org/3.0/library/stdtypes.html#id4

`replace(old, new [, count])`:Replaces `count` occurrences of `old` string with the `new` string. If `count` not specified, replaces all occurrences.

`title(), capital(), lower(), islower() and isupper()`: Returns a new string in title case, capital case and lower case respectively. The last two methods return `True` if all the characters in the string are lower and smaller respectively.

`swapcase()`: Returns another string with all uppercase characters converted to lowercase and vice versa of the given string.

`isalpha()` and `isalnum()` and `isdigit()`: Returns `True` if the string has only alphabets (`abc..zABC..Z`). The second function returns `True` if the string has only alphabets (`abc..zABC..Z`) and/or digits (`12..0`), without decimal point. The third function returns `True` if the string has digits only, without a decimal point.

`strip([str])`, `lstrip([str])` and `rstrip([str])`: Strip `str` from both the ends, leading end or the trailing end respectively. If no argument is provided, whitespaces (including tabs and newlines) are stripped. See the examples below.

```
str = ' brown'
str.strip(' br')
str.strip('br')
str.strip()
```

```
'own'
' brown'
'brown'
```

Mappings Types - Dictionary

There is currently only one standard mapping type, the dictionary. Dictionaries are an unordered collection of mappings between hashable values and arbitrary objects. Dictionaries are mutable objects. Keys are unique within a dictionary while values may not be. The values of a dictionary can be any arbitrary object (built-in or user defined) but the keys must be hashable. All immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are.

Dictionaries can be created by placing a comma-separated list of `key:value` pairs within curly braces (known as mappings). See the example below.

```
# Keys should be unique and hashable (strings mostly). Look carefully
# the keys are given within quotes.
my_dict = {'name':'quick fox', 'age':25, 'weight':56}
```

Dictionaries can also be created by making use of built-in type constructor - `dict()`. It can take keyword arguments, iterables (an iterable of tuples or lists each having exactly two objects, the first object becomes the key and the second object becomes the value) or mappings. Following is the syntax of this function.

```
dict([**kwargs])
dict(mapping [, **kwargs])
dict(iterable [, **kwargs])
```

This function expects only one mapping (`mapping`) or one iterable (`iterable`). Extra key value pairs can be provided in the form of keyword arguments (`**kwargs`). Note that keyword arguments are optional. Below are some examples of the usage of this function.

```
# Using keyword arguments. The keywords become the keys of the
# dictionary. Note that the keywords are not given within quotes.
my_dict = dict(name = 'quick fox', age = 25, weight = 56)

# Using key value value pairs in the form of mapping. Note that the
# keywords are given in quotes.
my_dict = dict({'name':'quick fox', 'age':25, 'weight':56})

# Note that any immutable object can act as key. Usually it is strings,
but float, int and other immutables can also act as key.
my_dict = dict({2:'quick fox', 39:25, 'weight':56})

# Using key value pairs in the form of lists or tuples. Each tuple or
# list should have exactly 2 elements. The first one becomes the key and
# the second one becomes the value. Note that the keywords are given in
# quotes.
my_dict = dict([['name', 'quick fox'], ['age', 25], ['weight', 56]])
my_dict = dict([('name', 'quick fox'), ('age', 25), ('weight', 56)])

# Additional keyword arguments can be provided along with the mapping or
# iterable.
```

```
my_dict = dict({'name':'fox', 'age':25, 'weight':56}, height = 1.5)
my_dict = dict([['name', 'quick fox'], ['age', 25]], weight = 56)
```

Dictionaries support the following set of operations and built-in functions.

```
# Creating the dictionary object
my_dict = {‘name’: ‘Quick Fox’, ‘area’: 24, ‘age’: 23.9}

# Value can be accessed with a key and square brackets
print(my_dict[‘name’])
print(my_dict[‘area’])

# Specific values can be modified.
my_dict['name'] = 'Lazy Fox'
print(my_dict)

# Adding new key-value pairs.
my_dict['height'] = 1.2
print(my_dict)

# Checks for availability of a key in the dictionary
'name' in my_dict

# Checks for non-availability of a key in the dictionary
'name' not in my_dict

# Returns the number of elements in the dictionary
len(my_dict)

# Delete specific value
del my_dict['height']
print(my_dict)

# Delete the complete dictionary.
del my_dict
```

```
Quick Fox
24
{'name': 'Lazy Fox', 'area': 24, 'age': 23.9}
{'name': 'Lazy Fox', 'area': 24, 'age': 23.9, 'height': 1.2}
True
False
4
{'name': 'Lazy Fox', 'area': 24, 'age': 23.9}
```

Other important functions available with the dictionary class are listed below. For a complete list of functions and operations supported by dictionary please see the following resources:
https://docs.python.org/3.7/library/stdtypes.html#mapping-types-dict

1. `clear()` - Removes all the elements from the dictionary.
2. `items()`, `keys()` and `values()` - Return a new view object of dictionary's key and value (in the form of tuples), keys and values respectively. View objects provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

   But view objects are not lists. The return type of these methods are `dict_items`, `dict_keys` and `dict_values` respectively. All these types are iterable and support membership tests. If you want list you may use list type constructor. For example, `list(dict.keys())` will return a list of dictionary keys. You may also use `iter()` function to get an iterator over these objects. Otherwise these objects are pretty much useless except for providing a live view of the underlying dictionary data.

3. `pop(key [, default])` - If `key` is in the dictionary, remove it and return its value, else return `default`. If `default` is not given and `key` is not in the dictionary, a `KeyError` is raised.
4. `popitem()` - Remove and return a (key, value) tuple from the dictionary. Pairs are returned in LIFO order.
5. `update([**kwargs])` or `update(mapping [, **kwargs])` or `update(iterable [, **kwargs])` - Updates the dictionary with the supplied key-value pairs. If the key already exists, its value is updated, otherwise the key-value pair is added. The usage of this function is similar to the usage of `dict()` function.

Set Types - Set

A set object is an unordered collection of distinct hashable objects. This means only immutable types can become part of a set. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

Set objects can be created by passing a set of hashable objects within curly braces. See example below.

```
my_set = {‘quick fox’, 2, 2.4}
print(my_set)
```

```
{'quick fox', 2, 2.4}
```

Set objects can also be created by using the built in type constructor. The following is the syntax:

```
my_set = set(iterable)
```

List, tuple, string, range, dictionary and set are all examples of iterables. The function can accept any one of these and create set from it. See below for sample usage of this function.

```
my_set = set('quick fox')
print(my_set)
my_set = set(['quick fox', 2, 2.4])
print(my_set)
```

```
my_dict = {'name': 'Quick Fox', 'area': 24, 'age': 23.9}
my_set = set(my_dict)
print(my_set)
my_set = set(list(range(5)))
print(my_set)
```

```
{'f', 'k', 'c', 'u', 'q', 'i', ' ', 'o', 'x'}
{'quick fox', 2, 2.4}
{'name', 'age', 'area'}
{0, 1, 2, 3, 4}
```

Sets support the following operations:

1. Inclusion check: `obj in set` returns `True` if `obj` is part of the set.
2. Size of set: `len(set)` returns the number of elements in the set.

Following functions are available in the Set class:

1. `add(item)` adds single element (`item`) to the set.
2. `update(iterable)` adds all the elements of the `iterable`. Could be any iterable including list, tuple, string, range, dictionary, and another set or a set of objects within curly braces. Behaves exactly as set().
3. `remove(element)` removes `element` from the set. If the element is not found `KeyError` is raised.
4. `discard(element)` removes `element` from the set. Will not raise any error, if the element is not found.
5. `pop()` removes the last element. Though you will never know which is the last element (set is an unordered collection).
6. `clear()` removes all the elements of the set.
7. `difference(t)`, `intersection(t)` or `union(t)` returns another set which is the difference, intersection or union with the other set (`t`). Difference of 2 sets can also be obtained by - operator. For example `s - t` is equivalent to `s.difference(t)`.
8. `issubset(t)` and `isuperset(t)` returns `True` if the host set is a subset or superset of `t`. Subset or superset can also be checked by `<=` or `<` operator. For example `s <= t` returns `True` if `s` is a subset of `t` or equal to `t`.
9. Sets also support equality check by using `==` operator. This operator returns `True` if both operands have the same content. This operation compares the content not the id of the objects contained by the sets.
10. `isdisjoint(t)` returns `True` if the host set and the other set (`t`) has an intersection.

Looping

The `while` Looping Construct

As with most languages while looping mechanism is also available with Python. See below examples.

```
counter = 0
while counter < 5:
      print(counter)
```

```
    counter += 1
```

```
0
1
2
3
4
```

The `for` Loop Construct

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. The `for` statement iterates over the items of any sequence (list, string, range among others), in the order that they appear in the sequence.

Iteration over an iterable or iterator - A

Iterating over an iterable or iterator is equivalent to *for each* looping construct from Java. See below example.

```
my_list = ['quick', 2, 'fox', 98.0]
for data in my_list:
    print(data)
```

```
quick
2
fox
98.0
```

Iteration over an iterable or iterator - B

Sometimes we also need to access the index of the element while iterating over an iterable/iterator. The build in function - `enumerate` comes in handy. This function returns an enumerator. Enumerator is just like an iterator which returns a two element tuple whose first element is the index and the second element is the corresponding element. Following is the syntax of this function.

```
enumerate(iterable/iterator, index_offset = 0)
```

This function accepts an iterator or an iterator and an optional `index_offset`. The value of the `index_offset` will be added to all the indices returned by the function. See example below.

```
my_list = ['quick', 2, 'fox', 98.0]
for index, value in enumerate(my_list, 2):
    print(index, value)
```

```
2 quick
3 2
4 fox
5 98.0
```

Iteration over a sequence of numbers – the range() function

```
for i in range(5):
      print(i)
```

```
0
1
2
3
4
```

Iteration over a string

```
my_string = ‘quick fox’
for char in my_string:
      print(char)
```

```
q
u
i
c
k

f
o
x
```

`break` and `continue`

As with other languages `break` statements breaks from the innermost loop. The `continue` statement, skips the part of code after continue and continues with the next iteration of the loop.

`else` statement with loops

Python has a provision of putting an else block after the `while` and `for` loops, which gets executed only when the loop has terminated after completing all of its iterations. In other words the loop has terminated because of a break statement.

```
for n in range(2, 10):
      for x in range(2, n):
            if n % x == 0:
                  print(n, 'equals', x, '*', n//x)
                  break
      else:
            print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

The `pass` Statement

In Python 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. 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. See the below examples for better understanding.

```
for n in range(1, 10):
      pass

def dead_function(args):
      pass

class empty_class:
       pass
```

Decision Making

Similar to other programming languages Python uses `if`, `elif` and `else` keywords for decision making during the program flow. See below example.

```
if count >= 50:
      print('More than or equal to 50')
elif count <= 0:
      print('Less than or equal to 0')
else
      print('In between 0 and 50')
```

Functions

Function is a piece of code written to carry out a specified task. To carry out that specific task, the function might or might not need multiple inputs. When the task is carried out, the function can or cannot return one or more values.

Python has large collection of inbuilt functions. A user can also define their own custom functions. The user defined functions are of two types – named function and lambda or anonymous functions.

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 does not have this restriction – it just refers to a standalone function. This means that all methods are functions, but not all functions are methods. Functions are defined using the keyword – def. See the below example.

```
1    # Defining the function.
2    def sum_upto(n):
3          '''Returns the sum of integers up to n.
4          Expects an integer argument. The return
5          type is also an integer.'''
6          sum = 0
7          for i in range(n):
8                sum += i
9          return sum
10
11   # Calling the function
12   sum_upto(5)
```

```
     10
```

Docstring

The first statement of the function body can optionally be a string literal, called documentation string, or docstring (lines 3 to 5). Docstrings describe what your function does, such as the computations it performs or its return values. Function docstrings are placed in the immediate line after the function header and are placed in between triple quotation marks. There are tools which use docstrings to automatically produce documentation, or to let the user interactively browse through the code. It is a good practice to include docstrings in code that you write.

The `return` Statement

Every function returns a value either explicitly using the `return` statement or implicitly (line number 9). Even if the function does not appear to return a value using the `return` statement the function actually returns `None` value (`NoneType`).

```
def nothing_returns(message):
      print(message)

print(nothing_returns('hello'))
```

```
hello
None
```

Multiple values can be returned using a tuple or a list.

```
def get_data():
      return (‘Bheem’, 14, 32.5)

print(get_data())
```

```
(‘Bheem’, 14, 32.5)
```

Function Arguments and Parameters

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. There are four types of arguments that Python uses:

1. Required Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable Number of Arguments

For our purpose we will mostly be using required arguments. These are the regular arguments and should mandatorily be supplied and in exactly the same order as expected.

Variable Length Arguments

Variable length arguments can be passed to a function using two special operators - asterisk (*) for non keyword arguments and double asterisk (**) for keyword arguments.

In the function declaration we should use an asterisk * before the parameter name to enable the function to accept variable length non keyword arguments. The arguments are passed as a tuple with the same name as the parameter excluding the asterisk. As a convention we use `arg` as the parameter name. See below example.

```
def does_nothing(*args):
      print(type(args))
      print(len(args))

does_nothing(45, 2.56, 'quick')
does_nothing(290, 'fox')

def still_does_nothing(param, *args):
      print(type(args))
      print(len(args))
      print(param)

still_does_nothing([1, 2, 3], 290, 'fox')
```

```
<class 'tuple'>
3
<class 'tuple'>
```

```
2
<class 'tuple'>
2
[1, 2, 3]
```

In the function declaration we should use the double asterisk ** before the parameter name to enable the function to accept a variable number of keyword arguments. The arguments are passed as a dictionary with the keys matching with the keys supplied as arguments. As a convention we use `kwargs` as the parameter name. See the below example.

```
def does_nothing(**kwargs):
      print(type(kwargs))
      print(len(kwargs))
      for key in kwargs.keys():
            print(key)

does_nothing(name = 'quick fox', age = 12)
```

```
<class 'dict'>
2
name
age
```

Lambda Functions

While normal functions are defined using the `def` keyword, in Python anonymous functions are defined using the `lambda` keyword. The lambda function has the following syntax:

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required. For example see the below code samples.

```
mod = lambda x, y: x**2 + y**2
print(mod(5, 2))
```

```
29
```

We use lambda functions when we require a nameless function for a short period of time. For example, it is mostly used with higher order functions (functions which take other functions as its arguments). In the below examples lambda functions have been used with filter and map built in functions.

```
# Below line creates a list of the even numbers between 0 and 9.
my_list = list(filter(lambda x: x%2 == 0, range(10)))
```

```
print(my_list)

# Below line creates a list of squares of numbers from 0 to 9.
my_list = list(map(lambda x: x**2, range(10)))
print(my_list)
```

```
[0, 2, 4, 6, 8]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Variables

Everything in Python is an object!

Every integer, float, string, boolean, lists, functions, classes, object instances (obviously), modules (once they have been imported in the current namespace), and almost every other language construct is conceptually an object. Consider the following statement.

```
person = 'quick fox'
height = 2
```

The part at the right hand side of the assignment operator is actually creating objects of type `str` and `int` and binding `person` and `height` to those objects respectively. The objects thus created are proper objects with a bunch of built in attributes and members functions (specific to `str` and `int` types).

The function `dir(object)` returns a list of all the valid attributes in an object (and inherited attributes as well). If we call this method with `height` as argument or even 2 as argument we get the following result. The attributes could be variables, functions, modules and other things. Without arguments, this function returns the list of names in the current local scope.

```
dir(height) # Or dir(2)
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

There are no variables in Python!

In the classical sense variables can be visualized as container of values. When a variable is declared as `int` the compiler/runtime allocates a memory chunk just sufficient to hold integer values. And

hence we cannot put `float` or other types in that container. In Python, variables are more like labels attached to objects. This is why we do not need to define the type of a variable in Python. And this is why we can map any label (variable) to any object type. **Instead of variables (in the classic sense), Python has names and object bindings.** This crucial difference has serious implications. Consider the following two scenarios.

Consider the following piece of code in C language. The first line allocates 4 bytes for a and stores 45 at that location. The second line allocates 4 bytes for b and copies the value of a (which is 45) at that location. It is to be noted that a and b are names pointing to two different memory locations, but both the memory locations are carrying the same value. This can be verified by using the & operator to get the addresses of a and b.

```
int a = 45;
int b = a;
printf("%p %p", &a, &b);
```

```
0x7fff343f9ec0 0x7fff343f9ec4
```

Consider the following piece of similar code in Python. In the first line, the Python interpreter, creates an object of type `int` and binds it with the name a. The second line binds b also with the same object. After these two statements, we can say that both a and b are bound (pointing) to the same object. This can be verified by making use of the function `id()`. This function returns an integer representing the identity (the address of the object in the memory) of the object.

```
a = 45
b = a
id(a)
id(b)
```

```
1952081776
1952081776
```

More about Objects

Every object has an identity, a type and a value. Identity of an object (essentially its address in the memory) never changes once it has been created. The `is` operator compares the identity of two objects.

```
a = 45
b = 65
a is b
a = b
a is b
```

```
False
True
```

The type of an object determines the operations which the object supports and also defines the possible values for objects of that type. The `type()` function returns the type of the object (which is

an object itself). Like its identity, the type of an object is also unchangeable, once it has been created.

The value of some objects can change. Objects whose value can change are said to be mutable and immutable otherwise.

Mutable and Immutable Objects

The value of some objects can change. Objects whose value can change are said to be mutable and immutable otherwise. Mutable objects in Python include - `list`, `dictionary`, `set`, and `byte array`. Immutable objects include - `int`, `float`, `complex`, `tuple`, `string`, `frozen set`, `bytes`.

For example you can create a `list`, append some values, and the `list` is updated in place. A `string` is immutable. Once you create a `string`, you cannot change its value. When you try to change a `string`, you are actually rebinding it to a newly created `string` object. The original object remains unchanged, even though nothing may be referring to it anymore. Consider the following example of string concatenation.

```
one = 'brown'
two = 'fox'
id(one)
id(two)
one += two # Concatenating string two to string one.
print(one)
id(one)
```

```
53860608
53859776
brownfox
58234424
```

Note that, after concatenation, the id of the object bound to one has changed. See the below example for integer.

```
a = 45
b = 54
id(a)
id(b)
a += b
print(a)
id(a)
```

```
1952081120
1952081152
99
1952081216
```

When we attempt to create two immutable objects (with the same content) and bind them to two different names, the interpreter may choose to create only one instance and bind both the names to the same instance. This is done for the sake of optimization. See the example below.

```
# Attempting to bind 45 (an int type object) to a and b. Only one
# instance is created.
a = 45
b = 45
id(a)
id(b)

# Sometimes two different instance may also be created.
a = 'quick fox jumps'
b = 'quick fox jumps'
id(a)
id(b)
```

```
1965713264
1965713264
56928880
51371848
```

The value of an immutable container object that contains a reference to a mutable object can change when the latter's value has changed. However the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle. Mutability of an object is determined only by its type. In the below example, the tuple (immutable) contains a list object (mutable). The contents of the list can be modified despite being held by an immutable container.

```
tpl = ('brown fox', 24.0, [1, 2, 3])
print(tpl)
tpl[2][0] = 45
print(tpl)
```

```
('brown fox', 24.0, [1, 2, 3])
('brown fox', 24.0, [45, 2, 3])
```

Namespaces and Variable Scopes

Variables are names that map to objects in memory. Python keeps track of all these mappings with namespaces. A namespace is a dictionary of variable names (keys) and their corresponding objects (values). This allows access to objects by names you choose to assign to them. The tricky part is that multiple independent namespaces can exist simultaneously. And names from one namespace can be re-used in another namespace without raising an error.

A block is a piece of Python program text that is executed as a unit. The following are blocks – a module, a function body, and a class definition. Each command typed interactively is a block. A script

file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block. A script command (a command specified on the interpreter command line with the -c option) is a code block. The string argument passed to the built-in functions `eval()` and `exec()` is a code block. **Note that contrary to Java and C looping and decision constructs do not introduce their own namespaces.**

The scope (visibility) of a name is the innermost block in which it was created. The scope of a name also extends to all the nested blocks. When the Python interpreter starts to look for names it follows a certain hierarchical pattern. It will first search in the namespace of the block in which the name was referred to. Followed by containing block and then followed by the block containing the containing block (if it exists) and so on. Finally it moves on to the global namespace followed by the build in namespace. This rule is called the LEGB rule. Local – Enclosed – Global – Built-in rule. If the referred name is not found in the entire hierarchy the interpreter raises a `NameError`.

An important note. Contrary to Java and C the looping and decision constructs in Python do not create namespaces of their own. The names introduced by these constructs are stored in the namespace they are declared in. We need to be looking out for such situations to avoid inadvertent bugs.

The `global` and `nonlocal` keywords

Consider the following code snippet and its output. Line 1 binds the `var` with a value (20) in the global namespace. Line 4 tries to rebind the same name with another value (40) but inside the function namespace. The original binding of `var` (created in line 1) remains intact. The first print statement (line 7) prints the value of `var` from the global namespace, the second print statement (line 4) prints the value of `var` from the function namespace, whereas the third print statement (line 9) again prints value of `var` from the global namespace.

```
1   var = 20
2
3   def print_value():
4         var = 40
5         print(var)
6
7   print(var)
8   print_value()
9   print(var)
```

```
20
40
20
```

Every time we bind a name to a value it gets bound to the value but only in the current scope. If the current scope already has a binding for the name it is overwritten. But the bindings in the enclosing scope or global scope are not touched.

What if we intentionally want to modify the existing bindings from inside a local scope? In this case `global` and `nonlocal` keywords come to help. `global var` tells the interpreter to use the binding of the name `var` in the top-most (or global scope). Putting `global var` in a code block is a

way of saying, “copy the binding of this global variable, or if you don’t find it, create the name `var` in the global scope."

Similarly, the `nonlocal var` statement instructs the interpreter to use the binding of the name `var` defined in the nearest enclosing scope. This is a way to rebind a name not defined in either the local or global scope. Without `nonlocal`, we would only be able to alter bindings in the local scope or the global scope. Unlike `global var` however, if we use `nonlocal var` then `var` must already exist. It will not be created if it is not found.

```
1   var = 20
2
3   def outer_fun():
4         #Use the existing binding from global namespace
5         global var = 40
6         print(var)
7         another_var = 80
8         print(another_var)
9
10        def inner_fun():
11              # Use the existing binding from enclosing namespace.
12              nonlocal another_var = 160
13              print(another_var)
14
15        inner_fun()
16        print(another_var)
17
18  print(var)
19  outer_fun()
20  print(var)
```

```
20 # Output from line 18
40 # Output from line 6
80 # Output from line 8
160 # Output from line 13
160 # Output from line 16
40 # Output from line 20
```

Thus, without making use of the `global` and `nonlocal` keywords, global variables and variables of enclosing scopes cannot be directly assigned a value from within the local scope, although they may be referenced.

Consider one more example to understand this well. The following piece will raise an `UnboundLocalError`. Carefully look at line 3. The statement is without `global` keyword. The interpreter, instead of using the existing binding from global namespace (created in line 1) will try to create a new binding for `var` in the function scope. The right hand side of the expression (line 3) will be evaluated first, leading to an error, because `var` is not defined in this scope.

```
1   var = 20
2   def print_value():
3         var = var + 40
```

```
4         print(var)
5
6   print(var)
7   print_value()
8   print(var)
```

```
    Traceback (most recent call last):
      File "<pyshell#481>", line 1, in <module>
        print_value()
      File "<pyshell#479>", line 2, in print_value
        var = var + 40
    UnboundLocalError: local variable 'var' referenced before assignment
```

The `globals()` and `locals()` functions

These two built-in function return the namespace from global and local scopes. The returned values are dictionaries.

The `del` operator

This operator unbounds a name from its object and removes it from the concerned namespace. The object, if not bound to another name too, is a candidate for garbage collection.

Some useful Built-in functions

Conversion Functions

`float([number|string])`: Convert a string or a number to a floating point. The argument may also be '`[+|-]nan`' or '`[+|-]inf`'. The string can optionally be preceded by a + or – (with no space in between) and optionally surrounded by whitespace.

`bool([object])`: Converts the object to its corresponding truth value. See truth testing of the objects. Without argument it returns `False`.

`int([number|string)`: Convert a number or string to an integer. If no arguments are given, return 0. The argument may also be '`[+|-]nan`' or '`[+|-]inf`'. The string can optionally be preceded by a + or – (with no space in between) and optionally surrounded by whitespace.

`bin(x)`: Convert an integer number to a binary string.

`hex(x)`: Convert an integer number to a hexadecimal string.

`oct(x)`: Convert an integer number to an octal string.

`str(object)`: Converts the object into its string representation.

`list([object])`: Return a list whose items are the same and in the same order as the items of `object`. `object` may be a sequence, a container that supports iteration, or an iterator object. If `object` is already a list, a copy is made and returned. For instance, `list('abc')` returns `['a',`

`'b', 'c']` and `list((1, 2, 3))` returns `[1, 2, 3]`. If no argument is given, returns an empty list.

`tuple([iterable])`: Return a tuple whose items are the same and in the same order as the iterable's items. `iterable` may be a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a tuple, it is returned unchanged. For instance, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`. If no argument is given, returns an empty tuple.

`set([iterable])`: Return a new set, optionally with elements taken from the `iterable`. If no argument is given returns an empty set.

`dict([arg])`: Return a dictionary initialized with the optional argument. See dictionary section for details of usage of this function.

Other Functions

`abs(x)`: Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.

`all(iterable)` or `any(iterable)`: Return `True` if all elements of the iterable are true and the other function returns `True` if any element is true.

`divmod(a, b):` Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division.

`globals():` Return a dictionary representing the current global symbol table.

`id(object):` Returns the identity of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id.

`input([prompt]):` If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that.

`isinstance(object, classinfo)`: Return True if the `object` argument is an instance of the `classinfo` argument, or of a (direct or indirect) subclass thereof.

`issubclass(class, classinfo):` Return True if class is a subclass (direct or indirect) of classinfo.

`iter(iterable|sequence):` Return an iterator object. The argument must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at 0). If it does not support either of these protocols, `TypeError` is raised.

`len(object)`: Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

`locals():` Update and return a dictionary representing the current local symbol table.

`map(function, iterable, ...)`: Return an iterator after applying the given function to every item of iterable. If additional iterable arguments are passed, the function must take that many

arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted.

```
num_one = [1, 2, 3]
num_two = [6, 7, 8]
def addition(x, y):
      return x + y

print(list(map(addition, num_one, num_two)))
```

```
[7, 9, 11]
```

`filter(function, iterable)`: Returns an iterator were the items are filtered through a function to test if the item is accepted or not. In simple words, the `filter()` method filters the given iterable with the help of a function that tests each element in the iterable to be true or not.

```
# Below line creates a list of the even numbers between 0 and 9.
my_list = list(filter(lambda x: x%2 == 0, range(10)))
print(my_list)
```

```
[0, 2, 4, 6, 8]
```

`min(iterable)` or `max(iterable)` or `min(args...)` or `max(args...)`: Returns the minimum or maximum value from the given iterable or the minimum or maximum of all the arguments.

`pow(x, y [, z])`: Return x to the power y. If z is present, return x to the power y, modulo z (computed more efficiently than `pow(x, y) % z`). The two-argument form `pow(x, y)` is equivalent to using the power operator: x**y. If y is negative the returned value is float and z should not be specified.

`round(x [, n])`: Return the floating point value x rounded to n digits after the decimal point. If n is omitted, it defaults to zero. The return value is an integer if called with one argument, otherwise of the same type as x.

`sum(iterable [, start])`: Sums the items of the iterable from left to right starting from `start`. `start` defaults to 0. The items of the iterable are normally numbers, and are not allowed to be strings.

`zip(*iterables)`: Make an iterator that aggregates elements from each of the iterables. Returns an iterator of tuples, where the ith tuple contains the ith element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With no arguments, it returns an empty iterator.

`ord(char)`: Returns the unicode value of the given one character string. If the input string is lengthier than one, TypeError will be raised. Unicode is a superset of ASCII, and the numbers 0 to 127 have the same meaning in ASCII as they have in Unicode. For example, the number 65 stands for A in ASCII as well as unicode.

`chr(code)`: Returns the character (string of length one) corresponding to the given unicode value. This function is the opposite of `ord`.

`reversed(iterable)`: Returns an iterable that accesses the `iterable` in reverse order.

`sorted(iterable [, key] [, reverse])`: Returns a new sorted iterable. The sorting will be in descending order if `reverse` is set to `True`. A comparison function can be provided through the optional `key` parameter. Any built-in or custom functions can be specified for comparison, **as long as it takes one argument and returns one value**. The `sorted` function applies the key function only once to every element of the iterable and uses the returned value to sort the iterable.

```
my_list = ['quick', 'brown', 'fox', 'jumps']

# Sorts the list lexicographically.
sorted(my_list)

def third(st):
      return st[2]

# Sorts the list in descending order using the third character.
sorted(my_list, key = third, reverse = True)
```

```
['brown', 'fox', 'jumps', 'quick']
['fox', 'brown', 'jumps', 'quick']
```

Rather than doing in-place sorting, this function returns a new list. If this function is applied over a dictionary it returns a new sorted list of keys in the dictionary.

The `sort([, key] [, reverse])` method in the list class can also be used for sorting. The `key` and `reverse` parameters are optional and have the same significance as they have in the `sorted`() built-in function. It is to be noted that this method does sorting in-place. In other words instead of returning a new list it modifies the list in-place.

Other functions can be seen here: https://docs.python.org/3/library/functions.html

References

1. https://sebastianraschka.com/Articles/2014_python_scope_and_namespaces.html
2. https://people.cs.clemson.edu/~malloy/courses/pythonProg-2015/lessons/scope/paper.pdf
3. https://docs.python.org/3.3/reference/executionmodel.html
4. https://jeffknupp.com/blog/2013/02/14/drastically-improve-your-python-understanding-pythons-execution-model/
5. https://docs.python.org/3/reference/datamodel.html