01 Built-in Fuctions

April 14, 2019

0.1 Built-in Functions

0.1.1 abs

Return the absolute value of an int or a float

```
In [2]: abs(-12)
```

```
Out[2]: 12
```

0.1.2 divmod

return a pair of integers consisting of their integer quotient and remainder

```
In [3]: divmod (7, 3)
```

```
Out[3]: (2, 1)
```

0.1.3 min / max

Min / Max of non-empty sequence (list, string, tuple)

```
In [4]: min([1, 3, 5, 7])
```

```
Out[4]: 1
```

```
In [5]: max('KPMG')
```

```
Out[5]: 'S'
```

0.1.4 round

Round to n digits after the decimal point

```
In [6]: round(1.6666666666, 2)
```

```
Out[6]: 1.67
```

Be aware of float precision!

```
In [7]: c = 2.675
        print(c)
        print(round(c, 2))
```

```
2.675
2.67
```

```
In [8]: 2.675 == 2.67499999999999982236431605997495353221893310546875
```

```
Out[8]: True
```

0.1.5 len

Size of an interable.

```
In [9]: len("Hello World")
```

```
Out[9]: 11
```

```
In [10]: len([1,2,3])
```

```
Out[10]: 3
```

0.1.6 dir

Return a list of valid attributes for that object.

```
In [11]: a_string = "Hello World"
         dir(a_string)
```

```
Out[11]: ['__add__',
          '__class__',
          '__contains__',
          '__delattr__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getitem__',
          '__getnewargs__',
          '__getslice__',
          '__gt__',
          '__hash__',
          '__init__',
          '__le__',
          '__len__',
```

```
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'_formatter_field_name_split',
'_formatter_parser',
'capitalize',
'center',
'count',
'decode',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'index',
'isalnum',
'isalpha',
'isdigit',
'islower',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'partition',
'replace',
'rfind',
'rindex',
'rjust',
'rpartition',
'rsplit',
'rstrip',
'split',
'splitlines',
'startswith',
```

```
          'strip',
          'swapcase',
          'title',
          'translate',
          'upper',
          'zfill']
```

```
In [12]: a_string.__len__()
```

```
Out[12]: 11
```

```
In [13]: a_string.__hash__()
```

```
Out[13]: 5386787323570539423
```

0.1.7 filter(function, iterable)

Return a list of all elements of iterable where the fuction returns **True**

```
In [14]: def my_filter_function(param):
             if isinstance(param, (int, float)):
                 return True
```

```
In [15]: worklist = [1, "2", "drei", 4, 5.5, 6.66]
```

```
In [16]: filter(my_filter_function, worklist)
```

```
Out[16]: [1, 4, 5.5, 6.66]
```

```
In [17]: filter(lambda x: isinstance(x, (int, float)) , worklist)
```

```
Out[17]: [1, 4, 5.5, 6.66]
```

0.1.8 map(function, iterable)

Apply function to each item of a list

```
In [18]: def my_map_function(param):
             if isinstance(param, (int, float)):
                 return param ** 2
             return param
```

```
In [19]: map(my_map_function, worklist)
```

```
Out[19]: [1, '2', 'drei', 16, 30.25, 44.3556]
```

```
In [20]: map(lambda x: x ** 2 if isinstance(x, (int, float)) else x, worklist)
```

```
Out[20]: [1, '2', 'drei', 16, 30.25, 44.3556]
```

0.1.9 reduce(function, iterable)

Apply function of two arguments cumulatively to the items of iterable, from left to right.
->> TODO Only applies to conseqcutive.

```
In [21]: worklist = [1, 16, 30.25, 44.3556]

In [22]: reduce(lambda x, y: x + y, worklist)

Out[22]: 91.60560000000001

In [23]: reduce(lambda x, y: x + y, worklist, 100.)

Out[23]: 191.6056
```

0.1.10 id(object)

The address of the object in memory

```
In [24]: a = True
         b = True
         c = False
         d = 10
         e = .125

In [25]: for _ in a, b, c, d, e:
             print(_, '\t', id(_))

True          4564413072
True          4564413072
False          4564413096
10          140606168938320
0.125          140606171305264
```

Note: a,b point to the same address (id) in memory, they both *refer* to 'True'

```
In [26]: b = c

         for _ in a, b, c, d, e:
             print(_, '\t', id(_))

True          4564413072
False          4564413096
False          4564413096
10          140606168938320
0.125          140606171305264
```

Note: c = point to the same address (id) in memory, they both *refer* to False.

```
In [27]: c = .125
         a = 10

         for _ in a, b, c, d, e:
             print(_, '\t', id(_))
```

```
10          140606168938320
False          4564413096
0.125          140606171305144
10          140606168938320
0.125          140606171305264
```

```
In [28]: id(a) == id(d)
```

```
Out[28]: True
```

```
In [29]: id(c) == id(e)
```

```
Out[29]: False
```

Note: a and d both refer to the same **10** in memory. Though c and d not not refer to the same **0.125** in memory. Though the value of the tow is exactly the same.

```
In [30]: c == e
```

```
Out[30]: True
```

```
In [31]: f = False
         id(b) == id(f)
```

```
Out[31]: True
```

```
In [32]: g = 10
         id(a) == id(d) == id(g)
```

```
Out[32]: True
```

Explaination:

- Python uses references, so there will always be only one **True** and **False** to be referred to in memory.
- **10** is handled exactly the same way, but **0.125** isn't. The reason is in the design: To inprove performance Python adds low integers to memory on startup, they will always remain regardless if referenced to.

0.1.11 isinstance(object, classinfo)

Return if object is an instance of the classinfo argument

```
In [33]: check_my_type = "Hello World!"

         type(check_my_type)

Out[33]: str

In [34]: isinstance(check_my_type, str)

Out[34]: True

In [35]: isinstance(check_my_type, int)

Out[35]: False

In [36]: isinstance(check_my_type, (int, str))

Out[36]: True

In [37]: import datetime

         check_my_type = datetime.datetime.utcnow()

In [38]: isinstance(check_my_type, (int, str))

Out[38]: False

In [39]: isinstance(check_my_type, datetime.datetime)

Out[39]: True
```

0.1.12 callable

```
In [40]: def my_function():
             return true

In [41]: callable(my_function)

Out[41]: True

In [42]: a_string = "Hello World"
         callable(a_string)

Out[42]: False
```

0.1.13 Enumerate

```
In [43]: a_list = list("ABCDEFGHIJK")
         a_list
```

```
Out[43]: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K']
```

First: some really bad ways to do it:

```
In [44]: # worst
         i = -1
         for item in a_list:
             i += 1
             print(i, a_list[i])
```

```
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
8 I
9 J
10 K
```

```
In [45]: # bad
         i = 0
         for item in a_list:
             print(i, a_list[i])
             i += 1
```

```
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
8 I
9 J
10 K
```

```
In [46]: # acceptable
         i = 0
```

```
        for item in a_list:
            i += 1
            print(i, item)
```

```
1 A
2 B
3 C
4 D
5 E
6 F
7 G
8 H
9 I
10 J
11 K
```

The best way:

```
In [47]: for i, item in enumerate(a_list):
             print(i, item)
```

```
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
8 I
9 J
10 K
```

- enumerate also accepts a start value

```
In [48]: for i, item in enumerate(a_list, start=11):
             print(i, item)
```

```
11 A
12 B
13 C
14 D
15 E
16 F
17 G
18 H
19 I
```

```
20 J
21 K
```

- enumerate also accepts generators

```
In [49]: for i, item in enumerate(xrange(10), start=11):
             print(i, item)
```

```
11 0
12 1
13 2
14 3
15 4
16 5
17 6
18 7
19 8
20 9
```

```
In [50]: def a_generator(n):
             i = 0
             while i < n:
                 yield i
                 i += 1

         for i, item in enumerate(a_generator(10)):
             print(i, item)
```

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