Python refresher

June 13, 2020

1 This is a Python refresher notebook.

1.1 Core Data Types

1.1.1 Numbers

[1]:

```
n1 = 1234
n2 = 12.34
print(n1, n2)
```

```
1234 12.34
```

1.1.2 Strings

[2]:

```
s1 = 'bob'
s2 = "Bob's"
print(s1, s2)
```

```
bob Bob's
```

1.1.3 Tuples

[3]:

```
t1 = (1,2,3)
t2 = (1,'spam',4,'U')
t3 = tuple('spam')
print(t1, t2, t3)
```

```
(1, 2, 3) (1, 'spam', 4, 'U') ('s', 'p', 'a', 'm')
```

1.1.4 Lists

[4]:

```
l1 = [1,[2,["three", "four"]],5, 6,[7,8]]
l2 = list(range(10))
print(l1, l2)
```

```
[1, [2, ['three', 'four']], 5, 6, [7, 8]] [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1.1.5 Sets

[5]:

```
st1 = {'3','2','1'}
st2 = set('abc')
print(st1, st2)
```

```
{'1', '3', '2'} {'a', 'b', 'c'}
```

1.1.6 Dictionaries

[6]:

```
d1 = {'food':'spam', 'taste':'yum'}
d2 = dict(hours=10)
print (d1, d2)
```

```
{'food': 'spam', 'taste': 'yum'} {'hours': 10}
```

1.2 Mutability

1.2.1 Immutable: You can never overwrite the values of the immutable objects. Example: Numbers, Strings, Tuples

[7]:

```
# s1[1] ='c'
# will result in type error
```

[8]:

```
# t2[1] = 'remove'
# will result in type error
```

1.2.2 Mutable

[9]:

```
l1[1] = 0
print(l1)
```

```
[1, 0, 5, 6, [7, 8]]
```

[10]:

```
d1['food'] = 'egg'
print(d1)
```

```
{'food': 'egg', 'taste': 'yum'}
```

1.3 Sequence Operation

Sequence is the generic term for an ordered set. There are several types of sequences in Python, the following three are the most important: lists, tuples, strings.

Common sequence operations are : indexing, slicing, concatenation

[11]:

```
print(l1[0], s1[2], t1[1]) #indexing
```

```
1 b 2
```

[12]:

```
print(l1[-1], s1[-1], t1[-2]) ## backward indexing
```

```
[7, 8] b 2
```

[13]:

```
print(l1[2:4], s1[1:], t1[:-1]) # slicing
```

```
[5, 6] ob (1, 2)
```

[14]:

```
print(l1+l1, s2+' Mill', t1*2) # concatenation
```

```
[1, 0, 5, 6, [7, 8], 1, 0, 5, 6, [7, 8]] Bob's Mill (1, 2, 3, 1, 2, 3)
```

1.4 Comprehensions [expression + looping construct]

List comprehension: Build new lists by running an expression on each items of a sequence

[15]:

```
print(l2)
lc1 = [x*x for x in l2] # List comprehension
print(lc1)
lc2 = {x:x*x for x in l2} # Dict comprehension
print(lc2)
lc3 = (x**3 for x in l2) # Generator object
print (lc3)
print(next(lc3))
print(next(lc3))
print(next(lc3))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
<generator object <genexpr> at 0x7fa98ae18af0>
0
1
8
```

[16]:

```
M = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
col0 = [x[0] for x in M]
print (col0)
diag = [M[i][i] for i in range(len(M))]
print(diag)
```

```
[1, 4, 7]
[1, 5, 9]
```

[17]:

```
G = (row for row in M) # generator object
rowsums = list(map(sum,G))
print(rowsums)
```

```
[6, 15, 24]
```

1.5 Type Specific Operations

1.5.1 Numbers

[18]:

```
print(1+2, 2*3, 2**(3), 2**(0.3))
```

```
3 6 8 1.2311444133449163
```

1.5.2 Strings

Common string operations are len(), find(), replace(), split(), upper(), lower(), isalpha(), format() etc

[19]:

```
s3 = 'Random Stuff'
print(len(s3))
print(s3.find('uf'))
print(s3.replace('Stuff','words'))
## strings are immutable, so a new string will be printed
## but there is no change in s3
print(s3.split(' '))
```

```
12
9
Random words
['Random', 'Stuff']
```

1.5.3 Lists

Common string operations are append(), pop(), sort(), reverse()

[20]:

```
l1.append([2,4])
print(l1)
```

```
[1, 0, 5, 6, [7, 8], [2, 4]]
```

[21]:

```
out = l1.pop()
print(l1)
print(out)
```

```
[1, 0, 5, 6, [7, 8]]
[2, 4]
```

[22]:

```
l3 = [2,4,1,2,42]
l3.sort()
print(l3)
l4 = ['dd','as','d']
l4.sort()
print(l4)
```

```
[1, 2, 2, 4, 42]
['as', 'd', 'dd']
```

[23]:

```
l3.reverse()
print(l3) # note that l3 is mutated.
```

```
[42, 4, 2, 2, 1]
```

1.6 Random Numbers

[24]:

```
import random
print(random.random())
print(random.choice([1,2,3,4]))
print(random.choice(list(range(100))))
```

```
0.1520151593322211
4
53
```

1.7 Pattern Matching

[25]:

```
import re
match = re.match('do', s3)
print(match)
match = re.match('Ra*', s3)
print(match)
match = re.match('ab(.*)AB(.*)end','abcdABCDend')
print(match.groups())
match = re.match('[/](.*)[/](.*)[/](.*)','/usr/home/etc/mydoc')
# note how the matching is resolved
print(match.groups())
```

```
None
<_sre.SRE_Match object; span=(0, 2), match='Ra'>
('cd', 'CD')
('usr/home', 'etc', 'mydoc')
```

1.8 Lambda Functions

[26]:

```
m = lambda x,y : x*y
```

[27]:

```
print(m(4,7))
```

```
28
```

[28]:

```
nums = [1,2,3,4]
sqr = map(lambda x: x*x, nums)
print(list(sqr))
```

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

[29]:

```
fltr = filter(lambda x:x%2, nums) # filter out even number
print(list(fltr))
```

```
[1, 3]
```

[30]:

```
# Map-filter can be used for list manipulations
examples = [3,2,5,1,2,5]
map_result = list(map(lambda x: x**2+5*x+6, examples))
print(map_result)
reduce_result = filter(lambda x:x in range(20), map_result)
print(list(reduce_result))
```

```
[30, 20, 56, 12, 20, 56]
[12]
```

1.9 Function Arguments

[31]:

```
def g(*args, **kwargs):
    print(args)
    print(kwargs)
g(1,2,3)
g(4,5,sum=15)
```

```
(1, 2, 3)
{}
(4, 5)
{'sum': 15}
```

[32]:

```
def f(w,x,y,z):
    print(w,x,y,z)
kw ={'y' : 3, 'z':2}
aw = (4,5)
f(*aw, **kw)
```

```
4 5 3 2
```

[33]:

```
def doubleit(func, *args, **kwargs):
    func(*args, **kwargs)
    return func(*args, **kwargs)

doubleit(print, 42, 'Arthur Dent', sep=' - ')
```

```
42 - Arthur Dent
42 - Arthur Dent
```

1.10 Generators

[34]:

```
def unique(iterable, key=lambda x: x):
    seen = set()
    for elem, ekey in ((e, key(e)) for e in iterable):
        if ekey not in seen:
                yield elem
                seen.add(ekey)
repeated_list = [1,3,3,2,3,1,4]
```

```
y = unique(repeated_list)
print(next(y))
print(next(y))
print(next(y))
print(next(y))
```

```
1
3
2
4
```

1.11 Decorators

[35]:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_whee():
    print("Whee!")
say_whee()
```

```
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

[36]:

```
# Boiler plate code for decorators

import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

[37]:

```
# Example : Time count
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)
        end_time = time.perf_counter()       # 2
        run_time = end_time - start_time     # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
waste_some_time(10)
waste_some_time(20)
```

```
Finished 'waste_some_time' in 0.0740 secs
Finished 'waste_some_time' in 0.1017 secs
```

[38]:

```
# Example: Code debug
import functools

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        args_repr = [repr(a) for a in args]                           # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]  # 2
        signature = ", ".join(args_repr + kwargs_repr)            # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")            # 4
        return value
    return wrapper_debug



@debug
def two_sum(a,b):
    return a-b;

two_sum(1,2)
```

```
Calling two_sum(1, 2)
'two_sum' returned -1
```

[38]: -1

[41]:

```
# caching and memoization
class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
```

```python
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__!r}")
        return self.func(*args, **kwargs)
def cache(func):
    """Keep a cache of previous function calls"""
    @functools.wraps(func)
    def wrapper_cache(*args, **kwargs):
        cache_key = args + tuple(kwargs.items())
        if cache_key not in wrapper_cache.cache:
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        print("Current cache is",wrapper_cache.cache)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = dict()
    return wrapper_cache

@cache
@CountCalls
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
fibonacci(4)
print('Now note that no more computation is done for the next calls')
print("fibonacci(2) is:", fibonacci(2))
print("fibonacci(3) is:", fibonacci(3))
print('See how the cached value is used')
print("fibonacci(7) is:", fibonacci(7))
```

```
Call 1 of 'fibonacci'
Call 2 of 'fibonacci'
Call 3 of 'fibonacci'
Call 4 of 'fibonacci'
Current cache is {(1,): 1}
Call 5 of 'fibonacci'
Current cache is {(1,): 1, (0,): 0}
Current cache is {(1,): 1, (0,): 0, (2,): 1}
Current cache is {(1,): 1, (0,): 0, (2,): 1}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3}
Now note that no more computation is done for the next calls
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3}
fibonacci(2) is: 1
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3}
fibonacci(3) is: 2
See how the cached value is used
Call 6 of 'fibonacci'
Call 7 of 'fibonacci'
Call 8 of 'fibonacci'
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3, (5,): 5}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3, (5,): 5}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3, (5,): 5, (6,): 8}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3, (5,): 5, (6,): 8}
Current cache is {(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3, (5,): 5, (6,): 8,
(7,): 13}
fibonacci(7) is: 13
```

[40]:

```python
# singletons
import functools

def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if not wrapper_singleton.instance:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton

@singleton
class TheOne:
    pass
first_one = TheOne()
another_one = TheOne()
id(first_one)
id(another_one)
first_one is another_one
```

[40]:

```
True
```