Python Hidden Treasure

(Draft Version: 0.1)

Table of Contents

Python Hidden Treasure

This ebook contains few lesser known Python gems. As usual, I will try to keep them updated and will continue to expand. If you wish to add any new, send them to me at (funmayank @ yahoo . co . in).

Variables

In-place value swapping

```
a = 10
b = "TEST"
a, b = b, a
print(a, b)
```

```
TEST 10
```

Unicode identifier

Python 3 allows to have unicode identifier's, which allows non-english speaking users to code.

```
हिन्दी = 10
print(हिन्दी)
```

```
10
```

Integer

Negative round

`round` is a function to round off the numbers and its normal usage is as follows

```
num = round(283746.32321, 1)
print(num)
```

```
283746.3
```

The second parameter defines the decimal number to which the number to rounded of. But if we provide a -ve number to it then it starts rounding of the number itself instead of decimal digit as shown in the below example

```
num = round(283746.32321, -2)
print(num)
num = round(283746.32321, -1)
print(num)
num = round(283746.32321, -4)
```

```
print(num)
```

```
283700.0
283750.0
280000.0
```

pow power - pow() can calculate (x ** y) % z

```
x, y, z = 1019292929191, 1029228322, 222224
pow(x, y, z)
```

```
115681
```

```
# Do not run this, please. it will take forever.
##### (x ** y) % z
```

String

Multi line strings

In python we can have multiple ways to achieve multi line strings.

- Using triple quotes

```
txt = """The Supreme Lord said: The indestructible, transcendental living
entity is called Brahman  and his eternal nature is called the
self. Action pertaining to the development of these  material
bodies is called karma, or fruitive activities."""
print(txt)
```

```
The Supreme Lord said: The indestructible, transcendental living
entity is called Brahman  and his eternal nature is called the
self. Action pertaining to the development of these  material
bodies is called karma, or fruitive activities.
```

- Using brackets "( )"

```
txt = ("The Supreme Lord said: The indestructible, transcendental living"
      "entity is called Brahman  and his eternal nature is called the "
      "self. Action pertaining to the development of these  material"
      "bodies is called karma, or fruitive activities.")
print(txt)
```

```
The Supreme Lord said: The indestructible, transcendental livingentity is called
Brahman  and his eternal nature is called the self. Action pertaining to the
development of these  materialbodies is called karma, or fruitive activities.
```

```
txt = "The Supreme Lord said: The indestructible, transcendental living " \
      "entity is called Brahman  and his eternal nature is called the"
print(txt)
```

```
The Supreme Lord said: The indestructible, transcendental living entity is called
Brahman  and his eternal nature is called the
```

Print String multiple times

using `string` multiply with `int` results in concatinating string that number of times. Lets print a line on console using `-`.

```
print("-" * 80)
```

```
--------------------------------------------------------------------------------
```

Search substring in string

```
print("ash" in "ashwini")
```

```
True
```

```
print("ash" is ['a', 's', 'h'])
```

```
False
```

```
print("ash" is 'ash')
```

```
True
```

```
### Implicit concatenation without "+" operator
name = "Mayank" " " "Johri"
print(name)
```

```
Mayank Johri
```

Join list of strings

```
list_cities = ["Bhopal", "New Delhi", "Agra", "Mumbai", "Aligarh", "Hyderabad"]


# Lets join the list of string in string using `join`
str_cities = ", ".join(list_cities)
print(str_cities)
```

```
Bhopal, New Delhi, Agra, Mumbai, Aligarh, Hyderabad
```

Reverse the string

There are few methods to reverse the string, but two are most common

- using slices

```
txt = "The Mother Earth"
print(txt[::-1])
```

```
htraE rehtoM ehT
```

```
txt = "The Mother Earth"
print("".join(list(reversed(txt))))
```

```
htraE rehtoM ehT
```

List / Tuple

tuple / list unpacking

```
a, b, *remaining = (1, 2, 3, 4, 5, "tst")
print(a, b)
print(remaining)
```

```
1 2
[3, 4, 5, 'tst']
```

```
a, b, *remaining = [1, 2, 3, 4, 5, "tst"]
print(a, b)
print(remaining)
```

```
1 2
[3, 4, 5, 'tst']
```

```
first,*middle,last = (1,2,3,4,5,6,7,8)

print(first, last)
print(middle)
```

```
1 8
[2, 3, 4, 5, 6, 7]
```

```
first,*middle,last = [1,2,3,4,5,6,7,8]

print(first, last)
print(middle)
```

```
1 8
[2, 3, 4, 5, 6, 7]
```

List/tuple multiplication ;)

similar to String we can literally multiply string and tuples with integer as shown below

```
lst = [1, 2, 3]
```

```
print(lst*3)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
print(lst*3)
```

```
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Array Transpose using `zip`

```
a = [(1,2), (3,4), (5,6)]
print(list(zip(a)))
print("*"*33)
print(list(zip(*a)))
```

```
[((1, 2),), ((3, 4),), ((5, 6),)]
*********************************
[(1, 3, 5), (2, 4, 6)]
```

enumerate with predefined starting index

```
lst = ["Ashwini", "Banti", "Bhaiya", "Mayank", "Shashank", "Rahul" ]
list(enumerate(lst))
```

```
[(0, 'Ashwini'),
 (1, 'Banti'),
 (2, 'Bhaiya'),
 (3, 'Mayank'),
 (4, 'Shashank'),
 (5, 'Rahul')]
```

Now, lets change the starting index to 10

```
print(list(enumerate(lst, 10)))
```

```
[(10, 'Ashwini'), (11, 'Banti'), (12, 'Bhaiya'), (13, 'Mayank'), (14, 'Shashank'),
(15, 'Rahul')]
```

Reverse the list

`built-in` keyword `reversed` allows the list to be reversed.

```
print(list(reversed([1, 2, 3, 4, 53])))
```

```
[53, 4, 3, 2, 1]
```

Flattening of list

```
l = [[1,2], [3], [4,5], [6], [7, 8, 9]]
l1 = [[1,2], 3, [4,5], [6], [7, 8, 9]]
```

- **Method 1:**

```
from itertools import chain
flattened_list = list(chain(*l))
print(flattened_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

NOTE: this method will fail if any of the element is non list item as shown in the below example

```
from itertools import chain
flattened_list = list(chain(*l1))
print(flattened_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-174-a93702599cff> in <module>()
      1 from itertools import chain
----> 2 flattened_list = list(chain(*l1))
      3 print(flattened_list)

TypeError: 'int' object is not iterable
```

- **Method 2:**

```
flattened_list = [y for x in l for y in x]
print(flattened_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

NOTE: this method will fail if any of the element is non list item as shown in the below example

```
flattened_list = [y for x in l1 for y in x]
print(flattened_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-190-7d7abc28b176> in <module>()
----> 1 flattened_list = [y for x in l1 for y in x]
      2 print(flattened_list)

<ipython-input-190-7d7abc28b176> in <listcomp>(.0)
----> 1 flattened_list = [y for x in l1 for y in x]
      2 print(flattened_list)

TypeError: 'int' object is not iterable
```

Lets update code to handle this situation

```
flattened_list = [si for i in l1 for si in (i if isinstance(i, list) else [i])]
print(flattened_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Method 3:**

```
flattened_list = sum(l, [])
print(flattened_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

NOTE: this method will fail if any of the element is non list item as shown in the below example

```
sum(l1, [])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-148-6c8ec61ef5b9> in <module>()
----> 1 sum(l1, [])

TypeError: can only concatenate list (not "int") to list
```

- **Method 4:**

```
flattened_list = []
for x in l:
    for y in x:
        flattened_list.append(y)
print(flattened_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

NOTE: this method will fail if any of the element is non list item as shown in the below example

```
flattened_list = []
for x in l1:
    for y in x:
        flattened_list.append(y)
print(flattened_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-166-4967a0a245fb> in <module>()
      1 flattened_list = []
      2 for x in l1:
----> 3     for y in x:
      4         flattened_list.append(y)
      5 print(flattened_list)

TypeError: 'int' object is not iterable
```

- **Method 5:**

```
from functools import reduce
flattened_list = reduce(lambda x, y: x + y, l)
print(flattened_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

NOTE: this method will fail if any of the element is non list item as shown in the below example

```
flattened_list = reduce(lambda x, y: x + y, l1)

print(flattened_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-197-78cb8e6490ba> in <module>()
----> 1 flattened_list = reduce(lambda x, y: x + y, l1)
      2 print(flattened_list)

<ipython-input-197-78cb8e6490ba> in <lambda>(x, y)
----> 1 flattened_list = reduce(lambda x, y: x + y, l1)
      2 print(flattened_list)

TypeError: can only concatenate list (not "int") to list
```

- **Method 6:**

```
import operator

flattened_list = reduce(operator.add, l)

print(flattened_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

NOTE: this method will fail if any of the element is non list item as shown in the below example

```
import operator

flattened_list = reduce(operator.add, l1)

print(flattened_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-187-2781850cb615> in <module>()
      1 import operator
----> 2 flattened_list = reduce(operator.add, l1)
      3 print(flattened_list)

TypeError: can only concatenate list (not "int") to list
```

Infinite Recursion

```
lst = [1, 2]

lst.append(lst)

print(lst)
```

```
[1, 2, [...]]
```

lets check if really we have infinite recursion, with the following code. We should get `RuntimeError: maximum recursion depth exceeded in comparison` error later in the execution.

```
def test(lst):

    for a in lst:
        if isinstance(a, list):
            print("A", a)
            test(a)
        print(a)

test(lst)
```

```
1
2
A [1, 2, [...]]
1
2
A [1, 2, [...]]
1
2
A [1, 2, [...]]
- - - - - - - - -
1
2
A [1, 2, [...]]
1
2
A [1, 2, [...]]
1
2
A [1, 2, [...]]
1

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-156-a9e88ac6beac> in <module>()
      6         print(a)
      7 
----> 8 test(lst)

<ipython-input-156-a9e88ac6beac> in test(lst)
      3         if isinstance(a, list):
      4             print("A", a)
----> 5             test(a)
      6         print(a)
      7 

... last 1 frames repeated, from the frame below ...

<ipython-input-156-a9e88ac6beac> in test(lst)
      3         if isinstance(a, list):
      4             print("A", a)
----> 5             test(a)
      6         print(a)
      7 

RuntimeError: maximum recursion depth exceeded in comparison
```

Copy a list

```
ori = [1, 2, 3, 4, 5, 6]
dup = ori
```

```
print(id(ori))
print(id(dup))
```

```
140397756982280
140397756982280
```

Both the variables are still pointing to same list, thus change in one will change another also.

```
dup.insert(0, 29)
print(ori)
print(dup)
```

```
[29, 1, 2, 3, 4, 5, 6]
[29, 1, 2, 3, 4, 5, 6]
```

Deepcopy a list

```
ori = [1, 2, 3, 4, 5, 6]
dup = ori[:]
print(id(ori))
print(id(dup))
```

```
140397756981960
140397756913096
```

```
dup.insert(0, 29)
print(ori)
print(dup)
```

```
[1, 2, 3, 4, 5, 6]
[29, 1, 2, 3, 4, 5, 6]
```

Dictionaries

Reverse the key values in unique dictionary

```
states_capitals = {'MP': 'Bhopal', 'UP': 'Lucknow', 'Rajasthan': 'Jaipur'}
```

- **Method 1:**

```
capitals_states = dict(zip(*list(zip(*states_capitals.items()))[::-1]))
print(capitals_states)
```

```
{'Jaipur': 'Rajasthan', 'Bhopal': 'MP', 'Lucknow': 'UP'}
```

- **Method 2:**

```
capitals_states = dict([v,k] for k,v in states_capitals.items())
print(capitals_states)
```

```
{'Jaipur': 'Rajasthan', 'Bhopal': 'MP', 'Lucknow': 'UP'}
```

- **Method 3:**

```
capitals_states = dict(zip(states_capitals.values(), states_capitals.keys()))
print(capitals_states)
```

```
{'Jaipur': 'Rajasthan', 'Bhopal': 'MP', 'Lucknow': 'UP'}
```

- **Method 4:**

```
capitals_states = {states_capitals[k] : k for k in states_capitals}
print(capitals_states)
```

```
{'Jaipur': 'Rajasthan', 'Bhopal': 'MP', 'Lucknow': 'UP'}
```

Creating dictionaries

Multiple methods can be used to create a dictionary. We are going to cover few of the cool ones.

- **Using two lists**

```
states = ["MP", "UP", "Rajasthan"]
capitals = ["Bhopal", "Lucknow", "Jaipur"]

states_capitals = dict(zip(states, capitals))
print(states_capitals)
```

```
{'MP': 'Bhopal', 'UP': 'Lucknow', 'Rajasthan': 'Jaipur'}
```

- **Using arguments**

```
states_capitals = dict(MP='Bhopal', Rajasthan='Jaipur', UP='Lucknow')
print(states_capitals)
```

```
{'MP': 'Bhopal', 'UP': 'Lucknow', 'Rajasthan': 'Jaipur'}
```

- **list of tuples**

```
states_capitals = dict([('MP', 'Bhopal'), ('UP', 'Lucknow'), ('Rajasthan', 'Jaipur')])
print(states_capitals)
```

```
{'MP': 'Bhopal', 'UP': 'Lucknow', 'Rajasthan': 'Jaipur'}
```

- **By adding two dictionary using copy and update**

```
a = {'MP': 'Bhopal', 'UP': 'Lucknow', 'Rajasthan': 'Jaipur'}
b = {'Jaipur': 'Rajasthan', 'Bhopal': 'MP', 'Lucknow': 'UP'}
c = a.copy()
c.update(b)
```

```
print(c)
```

```
{'MP': 'Bhopal', 'Jaipur': 'Rajasthan', 'Bhopal': 'MP', 'UP': 'Lucknow', 'Lucknow':
'UP', 'Rajasthan': 'Jaipur'}
```

```
# for Python >= 3.5: https://www.python.org/dev/peps/pep-0448
c = {**b, **a}
print(c)
```

- **Using dictionary comprehension**

```
def  double_bubble(x):
    yield x
    yield x*x

d = {k:v for k, v in double_bubble}
```

```
{chr(97+i)*2 : i for i in range(5)}
```

```
{'aa': 0, 'bb': 1, 'cc': 2, 'dd': 3, 'ee': 4}
```

if

Conditional Assignment

```
y = 10
x = 3 if (y == 1) else 2
print(x)
```

```
2
```

```
x = 3 if (y == 1) else 2 if (y == -1) else 1
print(x)
```

```
1
```

Functions

default arguments

Dangerous mutable default arguments

```
def foo(x=[]):
    x.append(1)
    print(x)


foo()
foo()
foo()
```

```
[1]
[1, 1]
[1, 1, 1]
```

```
# instead use:
def fun(x=None):
    if x is None:
        x = []
    x.append(1)
    print(x)


fun()
fun()
fun()
```

```
[1]
[1]
[1]
```

TODO: Add more examples

Function argument unpacking

```
def draw_point(x, y):
    """You can unpack a list or a dictionary as
    function arguments using * and **."""
    print(x, y)


point_foo = (3, 4)
point_bar = {'y': 3, 'x': 2}


draw_point(*point_foo)
draw_point(**point_bar)
```

```
3 4
2 3
```

Function arguments

```
def letsEcho():
    test = "Hello"
    print(test)

letsEcho.test = "Welcome"
print(letsEcho.test)
letsEcho()
```

```
Welcome
Hello
```

Finally returns the ultimate return

```
def dum_dum():
    try:
        return '`dum dum` returning from try'
    finally:
        return '`dum dum` returning from finally'

print(dum_dum())
```

```
`dum dum` returning from finally
```

OOPS

Attributes

Dynamically added attributes

```
class Test():
    def __getattribute__(self, name):
        f = lambda: " ".join([name, name[::-1]])
        return f

t = Test()
# New attribute created at runtime
t.rev()
```

```
'rev ver'
```

operators

Chaining comparison operators

```
x = 5
```

```
1 < x < 100
```

```
True
```

```
10 < x < 20
```

```
False
```

```
x < 10 < x*10 < 100
```

```
True
```

```
10 > x <= 9
```

```
True
```

```
5 == x > 4
```

```
True
```

enumerate

Wrap an iterable with enumerate and it will yield the item along with its index.

```
a = ['a', 'b', 'c', 'd', 'e']

for index, item in enumerate(a): print (index, item)
```

```
0 a
1 b
```

```
2 c
3 d
4 e
```

Generators

Sending values into generator functions

https://www.python.org/dev/peps/pep-0342/, also please reaad http://www.dabeaz.com/coroutines/

```
def mygen():
    """Yield 5 until something else is passed back via send()"""
    a = 5
    while True:
        f = (yield a) #yield a and possibly get f in return
        if f is not None:
            a = f  #store the new value

g = mygen()
print(next(g))
print(next(g))
g.send(7)
print(next(g))
print(next(g))
g.send(17)
print(next(g))
print(next(g))
```

```
5
5
7
7
17
17
```

Descriptor

http://users.rcn.com/python/download/Descriptor.htm

Iterators

iter() can take a callable argument

```
def seek_next_line(f):
    """
    The iter(callable, until_value) function repeatedly calls
    callable and yields its result until until_value is returned.
    """
    for c in iter(lambda: f.read(1),'\n'):
        pass
```

I/O

with

open multiple files in a single `with`.

```
try:
    with open('a', 'w') as a, open('b', 'w') as b:
        pass
except IOError as e:
    print ('Operation failed: %s' % e.strerror)
```

```
#### write file using `print`
```

```
with open("outfile.txt" , "w+") as outFile:
    print('Modern Standard Hindi is a standardised and sanskritised register of the Hindustani language.', file=outFile)
```

Exception

Re-raising exceptions:

```
# Python 2 syntax
try:
    some_operation()
except SomeError, e:
    if is_fatal(e):
        raise
    handle_nonfatal(e)
```

```
  File "<ipython-input-54-bfe95c85d6b3>", line 4
    except SomeError, e:
                    ^
SyntaxError: invalid syntax
```

```
def some_operation():
    raise Exception


def is_fatal(e):
    return True


# Python 3 syntax
try:
    some_operation()
except Exception as e:
    if is_fatal(e):
        raise
    handle_nonfatal(e)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-55-18cc95aa93e6> in <module>()
      7 # Python 3 syntax
      8 try:
----> 9     some_operation()
     10 except Exception as e:
     11     if is_fatal(e):

<ipython-input-55-18cc95aa93e6> in some_operation()
      1 def some_operation():
----> 2     raise Exception
      3
      4 def is_fatal(e):
      5     return True

Exception:
```

!!! Easter Eggs !!!

```
from __future__ import braces
```

```
  File "<ipython-input-56-6d5c5b2f0daf>", line 1
    from __future__ import braces
SyntaxError: not a chance
```

```
import __hello__
```

```
Hello world!
```

Lets encrypt our code using `cot13`

```
import codecs
s   = 'The Zen of Python, by Tim Peters'
enc = codecs.getencoder( "rot-13" )
dec = codecs.getdecoder("rot-13")
os  = enc( s )[0]
print(os)
print(dec(os)[0])
```

```
Gur Mra bs Clguba, ol Gvz Crgref
The Zen of Python, by Tim Peters
```

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```