PythonTip 01 - Functions (post-class-version)

January 27, 2025

1 Python Tip #1: Functions

Functions are separately defined code snippets that you can then use in your main code.

[1]:
```
def double(number): # arguments = input
    new_number = 2*number
    return new_number # output
```

[2]:
```
double(5)
```

[2]:
```
10
```

[3]:
```
double("hello")
```

[3]:
```
'hellohello'
```

[5]:
```
def print_hello():
    print("hello")
```

[6]:
```
print_hello()
```

```
hello
```

[7]:
```
print_hello()
print_hello()
print_hello()
```

```
hello
hello
hello
```

[8]:
```
message = print_hello()
```

```
hello
```

[10]:
```
print(message)
```

```
None
```

[11]:
```
def double(number=7): # number will default to 7 if you don't specify it
    new_number = 2*number
    return new_number
```

[12]:
```
double()
```

[12]: 14

[13]:
```
double(5)
```

[13]: 10

[14]:
```
double(number=5)
```

[14]: 10

[37]:
```
def add_up(a=1, b=2, c=3, d=4, e=5):
    return a + b + c + d + e
```

[40]:
```
add_up(c=100)
```

[40]: 112

[16]:
```
number = 9

def double(number): # arguments = input
    new_number = 2*number
    return new_number

print(number)
print(double(5))
```

```
9
10
```

"Lambda Functions" sound very fancy, but they are just a quicker way to define very simple functions.

```
double = lambda x : 2*x
```

```
[name] = lambda [inputs] : [outputs]
```

[17]:
```
new_double = lambda number : 2*number
new_double(5)
```

[17]: 10

[18]:
```
combine = lambda x, y: 2*x + 3 * y**2
```

[19]:
```
combine(5,2)
```

[19]: 22

[ ]:

They are often useful (as we'll see later) for extracting one component of a tuple or list.

[20]:
```
second_component = lambda r : r[1]
```

[21]:
```
second_component([5, -8, 1])
```

[21]: -8

This is totally equivalent to:

```
def second_component(r):
    return r[1]
```

This is mostly useful when you just want to use the function in one spot, and not define it forever.

When sorting a list, you can give it a "key" function to tell it what to sort by.

[22]:
```
L = [-5, 1, 0, 7, -10]
print(L)
L.sort()
print(L)
```

```
[-5, 1, 0, 7, -10]
[-10, -5, 0, 1, 7]
```

[23]:
```
L.sort(key=lambda x : abs(x))
print(L)
```

```
[0, 1, -5, 7, -10]
```

[34]:
```
strings = ["hello", "world", "A", "Z"]
strings.sort(key=lambda z : z.lower())
print(strings)
```

```
['A', 'hello', 'world', 'Z']
```

[ ]:

[35]:
```
def make_lower(z):
    return z.lower()

strings.sort(key=make_lower)
print(strings)
```

```
['A', 'hello', 'world', 'Z']
```

[33]:
```
"hELLo".lower()
```

[33]:
```
'hello'
```

[32]:
```
things = [[1, 2, 3], [1, 1, 1], [5]]
things.sort()
print(things)
```

```
[[1, 1, 1], [1, 2, 3], [5]]
```

[29]:
```
ord("a")
```

[29]:
```
97
```

[31]:
```
chr(92)
```

[31]:
```
'\\'
```

[41]:
```
L = [(0, 3), (-1, 7), (2, 5)]
```

[42]:
```
sorted(L)
```

[42]:
```
[(-1, 7), (0, 3), (2, 5)]
```

[43]:
```
L
```

[43]:
```
[(0, 3), (-1, 7), (2, 5)]
```

[44]:
```
sorted(L, key=lambda x : x[1])
```

[44]:
```
[(0, 3), (2, 5), (-1, 7)]
```

[49]:
```
sorted([(2, 5), (0, 5), (1, 7)], key=lambda x : x[1])
```

[49]:
```
[(2, 5), (0, 5), (1, 7)]
```

[ ]:
```
# If the key function is tied, then it leaves them in the original order
# what if we want it to look at the first component to break ties in the second␣
 ↪component?
```

[ ]:
```
def sly(x):
    return (x[1], x[0])
```

[46]:
```
sorted([(2, 5), (0, 5), (1, 7)], key=lambda x : (x[1], x[0]))
```

[46]:
```
[(0, 5), (2, 5), (1, 7)]
```

[ ]:
```

```