01ex_Fundamentals

January 18, 2025

0. Implement a function (whatever you want) and save it to a file (e.g. `function.py`). Import that file and use that function in this notebook.

[13]:
```
from lab01exfunction import square

print(square(8))
```

```
64
```

1. Write the following as a list comprehension

[3]:
```
# 1
ans = []
for i in range(3):
    for j in range(4):
        ans.append((i, j))
print (ans, end = '\n\n')

# 2
ans = map(lambda x: x*x, filter(lambda x: x%2 == 0, range(5)))
print (list(ans))
```

```
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1),
(2, 2), (2, 3)]

[0, 4, 16]
```

[35]:
```
# 1
# nested list comprehension
ans = [(i, j) for j in range(4) for i in range(3)]
print(ans, end= '\n\n')

#2
ans = [x**2 for x in range(5) if x%2 ==0 ]
print(ans)
```

```
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2), (0, 3),
(1, 3), (2, 3)]

[0, 4, 16]
```

2. Convert the following function into a pure function with no global variables or side effects

this is to remember that the interpreter treats immutable type function arguments as copies but mutable types arguments (such as list!) as references

[17]:
```
x = 5
def f(alist):
    for i in range(x):
         alist.append(i)
    return alist

alist = [1,2,3]
print(alist)
ans = f(alist)
print (ans)
print (alist) # alist has been changed!
```

```
[1, 2, 3]
[1, 2, 3, 0, 1, 2, 3, 4]
[1, 2, 3, 0, 1, 2, 3, 4]
```

[18]:
```
import copy
def f(alist):
    new_list = copy.copy(alist)
    for i in range(5):
        new_list.append(i)
    return new_list

alist = [1,2,3]
print(alist)
ans = f(alist)
print (ans)
print (alist)
```

```
[1, 2, 3]
[1, 2, 3, 0, 1, 2, 3, 4]
[1, 2, 3]
```

3. Write a `decorator` hello that makes every wrapped function print "Hello!", i.e. something like:

```
@hello
def square(x):
    return x*x
```

`*args`, `**kwargs` stands for arguments and keyword arguments respectively

`*args` are positional, unnamed arguments e.g. `square(8)`, 8 is a positional argument positional arguments, as the name suggest, are interpreted as their position says. e.g.

```
def power(n,m):
    return n**m
```

here `power(2,3)` is 2^3 while `power(3,2)` is 3^2. they are different and one should remember how the function was defined and pass the positional arguments in the correct order

if one calls the function as `power(n=2, m=3)`, then the values (2,3) are passed as keyword arguments *kwargs are named arguments

[29]:
```python
#here i specify that func can contain an arbitrary number of both
#named and unnamed arguments

def my_decorator(func):
    def wrapped_func(*args, **kwargs):
        print("Hello")
        return func(*args, **kwargs)
    return wrapped_func

@my_decorator
def square(n):
    return n**2

m = square(n=8)
print(m)
```

```
Hello
64
```

4. Write the factorial function so that it a) does and b) does not use recursion.

[34]:
```python
def factorial(n):
    f = 1
    for i in range(1, n+1):
        f *= i
    return f


print(factorial(4))
```

```
24
```

5. Use HOFs (zip in particular) to compute the weight of a circle, a disk and a sphere, assuming different radii and different densities:

```python
densities = {"Al":[0.5,1,2],"Fe":[3,4,5],"Pb": [15,20,30]}
radii = [1,2,3]
```

where the entries of the dictionary's values are the linear, superficial and volumetric densities of the materials respectively.

In particular define a list of three lambda functions using a comprehension that computes the circumference, the area and the volume for a given radius.

[47]:
```python
from math import pi
d_volumes = [lambda x: 2*pi*x, lambda x: pi* (x**2), lambda x: (4./3.
 ↪)*pi*(x**3)]

densities = {"Al":[0.5,1,2],"Fe":[3,4,5],"Pb": [15,20,30]}
radii = [1,2,3]

# I define a new dictiionary for weights, where the labels are materials and␣
 ↪the values is a list of lists
# where the inner list contains weights for a fixed radius
weights = {}

for material in densities.keys():
    outer_list = []
    for radius in radii:
        #inner_list = [d_volumes[d](radius) * densities[material][d] for d in␣
 ↪range(len(d_volumes))]
        #but this below is better!
        inner_list = [volume(radius) * density for volume,density in␣
 ↪zip(d_volumes, densities[material])]
        outer_list.append(inner_list)
    weights[material] = outer_list

for key, values in zip(weights.keys(), weights.values()):
    print(key, values)
```

```
Al [[3.141592653589793, 3.141592653589793, 8.377580409572781],
[6.283185307179586, 12.566370614359172, 67.02064327658225], [9.42477796076938,
28.274333882308138, 226.19467105846508]]
Fe [[18.84955592153876, 12.566370614359172, 20.94395102393195],
[37.69911184307752, 50.26548245743669, 167.5516081914556], [56.548667764616276,
113.09733552923255, 565.4866776461627]]
Pb [[94.24777960769379, 62.83185307179586, 125.66370614359171],
[188.49555921538757, 251.32741228718345, 1005.3096491487337],
[282.7433388230814, 565.4866776461628, 3392.920065876976]]
```

6. Edit the class defintion to add an instance attribute of is_hungry = True to the Dog class. Then add a method called eat() which changes the value of is_hungry to False when called. Figure out the best way to feed each dog and then output "My dogs are hungry." if all are hungry or "My dogs are not hungry." if all are not hungry. The final output should look like this:

```
I have 3 dogs.  Tom is 6.  Fletcher is 7.  Larry is 9.  And they're all mammals,
of course.  My dogs are not hungry.
```

```python
# Parent class
class Dog:

    # Class attribute
    species = 'mammal'
```

```python
    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)

# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)

# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

[68]:

```python
class Dog:
    # Class attribute
    #these are not changeable
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.is_hungry = True

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)
    def eats(self):
        self.is_hungry = False
        return
```

```
my_dogs = [Dog(name= 'Tom', age= 6), Dog(name= 'Fletcher', age= 7),␣
 ↪Dog(name='Larry',age= 9)]



print(f"I have {len(my_dogs)} dogs", end=". ")
for i, dog in enumerate(my_dogs): print(dog.description(), end = ". " if␣
 ↪i==len(my_dogs) -1 else ", ")
print(f"And they are all {my_dogs[0].species}, of course.")


for dog in my_dogs: dog.eats()
#my_dogs[0].eats()
#my_dogs[1].eats()
#my_dogs[2].eats()

from functools import reduce
all_hungry_check = reduce(lambda x,y: x*y , [dog.is_hungry for dog in my_dogs])
all_not_hungry_check = reduce(lambda x,y: x*y , [not(dog.is_hungry) for dog in␣
 ↪my_dogs])

if all_hungry_check == True:
    print("My dogs are hungry.")
elif all_not_hungry_check == True:
     print("My dogs are not hungry.")
```

```
I have 3 dogs. Tom is 6 years old, Fletcher is 7 years old, Larry is 9 years
old. And they are all mammal, of course.
My dogs are not hungry.
```