MSOE UNIVERSITY

Methods and Functions in Python (Part 2)

Instructor: Yiyang (Ian) Wang

Lambda Expressions

Lambda Expression

- Lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using *def*.
- Function objects returned by running lambda expressions work exactly the same as those created and assigned by *defs*. There is key difference that makes lambda useful in specialized roles:

  lambda's body is a single expression, not a block of statements.

A Lambda Expression Example

```
def square(num):
    result = num**2
    return result
```

```
square(2)
```

```
4
```

```
def square(num):
    return num**2
```

```
square(2)
```

```
4
```

```
def square(num): return num**2
```

```
square(2)
```

```
4
```

```
square = lambda num: num **2
```

```
square(2)
```

```
4
```

Why Lambda Expressions?

- Many function calls need a function passed in, such as map and filter.
- Often you only need to use the function you are passing in once, so instead of formally defining it, you just use the lambda expression.

```
list(map(lambda num: num ** 2, my_nums))
```

```
[1, 4, 9, 16, 25]
```

```
list(filter(lambda n: n % 2 == 0,nums))
```

```
[0, 2, 4, 6, 8, 10]
```

Lambda Expression – More Examples

```
lambda s: s[0]
```

```
<function __main__.<lambda>>
```

```
lambda s: s[::-1]
```

```
<function __main__.<lambda>>
```

```
lambda x,y : x + y
```

```
<function __main__.<lambda>>
```

Exercise 1: You are given a list of integers. Write a Lambda expression to filter out all the even numbers from the list.

numbers = [1,2,3,4,5,6,7,8,9,10]

Exercise 2: Given a list of tuples representing names and ages, use a lambda expression to sort the list based on age in ascending order.

```
people = [("Alice", 30), ("Bob", 25), ("Charlie", 35)]
```

Hint: using Sorted() function

Nested Statements and Scope

What is the output of *print(x)* and *print(printer())*?

```
x = 25

def printer():
    x = 50
    return x
```

LEGB Rule

- **L: Local** – Names assigned in any way within a function (*def* or *lambda*), and not declared global in that function
- **E: Enclosing function locals** – Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer
- **Global (module)** – Names assigned at the top-level of a module file, or declared global in a def within the file
- **Built-in (Python)** – Names preassigned in the built-in names module: open, range, SyntaxError,...

Local

```
# x is local here:
f = lambda x:x**2
```

Enclosing Function Locals

```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'

    def hello():
        print('Hello '+name)

    hello()

greet()
```

Global

```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'

    def hello():
        print('Hello '+name)

    hello()

greet()
```

```
print(name)
```

```
This is a global name
```

Built-in

These are the built-in function names in Python (don't overwrite these!)

```
len
```

```
<function len>
```

Exercise: What will be the output?

```
x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

Exercise: What will be the output?

```
x = 50

def func():
    global x
    print('This function is now using the global x!')
    print('Because of global x is: ', x)
    x = 2
    print('Ran func(), changed global x to', x)

print('Before calling func(), x is: ', x)
func()
print('Value of x (outside of func()) is: ', x)
```

You can use the *globals()* and *locals()* functions to check what are your current local and global variables.

*args and **kwargs

*args

When a function parameter starts with an asterisk, it allows for ***an arbitrary number of arguments***, and the function takes them in as a tuple of values.

```
1 def myfunc(a=0,b=0,c=0,d=0,e=0):
2     return sum((a,b,c,d,e))*.05
3
4 myfunc(40,60,20)
```

```
6.0
```

```
1 def myfunc(*args):
2     return sum(args)*.05
3
4 myfunc(40,60,20)
```

```
6.0
```

**kwargs

Similarly, Python offers a way to handle arbitrary numbers of keyworded arguments. Instead of creating a tuple of values, **kwargs builds ***a dictionary of key/value pairs.*** For example:

```
1 def myfunc(**kwargs):
2     if 'fruit' in kwargs:
3         print(f"My favorite fruit is {kwargs['fruit']}")
4     else:
5         print("I don't like fruit")
6
7 myfunc(fruit='pineapple')
```

```
My favorite fruit is pineapple
```

```
1 myfunc()
```

```
I don't like fruit
```

*args and **kwargs combined

You can pass *args and **kwargs into the same function, **but *args have to appear before **kwargs**

```
1 def myfunc(*args, **kwargs):
2     if 'fruit' and 'juice' in kwargs:
3         print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fruit']}")
4         print(f"May I have some {kwargs['juice']} juice?")
5     else:
6         pass
7
8 myfunc('eggs','spam',fruit='cherries',juice='orange')
```

```
I like eggs and spam and my favorite fruit is cherries
May I have some orange juice?
```

Placing keyworded arguments ahead of positional arguments raises an exception:

```
1 myfunc(fruit='cherries',juice='orange','eggs','spam')
```

```
  File "<ipython-input-8-fc6ff65addcc>", line 1
    myfunc(fruit='cherries',juice='orange','eggs','spam')
                                                       ^
SyntaxError: positional argument follows keyword argument
```

Exercise 3: Write a Python function called *sum_values* that accepts a variable number of arguments and returns the sum of all the numeric values passed as arguments. The function should ignore any non-numeric values.

Hint: isinstance(arg, (int, float))

```
result = sum_values(1, 2, 3, "four", 5, 6, "seven")
print(result)  # Output: 17 (1 + 2 + 3 + 5 + 6)
```

```
17
```

Exercise 4: What is the output?

```python
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"Key: {key}, Value: {value}")

print_info(name="Alice", age=30, city="New York", country="USA")
```

Top Secret :)

Errors and Exception Handling

try and except

```
try:
    f = open('testfile','w')
    f.write('Test write this')
except IOError:
    # This will only check for an IOError exception and then execute this
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

We could have also just said *except:* if we weren't sure what exception would occur.

```
try:
    f = open('testfile','r')
    f.write('Test write this')
except:
    # This will check for any exception and then execute this print state
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

finally: ensure that certain code is always executed, regardless of whether an exception occurred or not during the execution of the try block.

```
try:
    f = open("testfile", "w")
    f.write("Test write statement")
    f.close()
finally:
    print("Always execute finally code blocks")
```

It's often used for cleanup tasks, such as closing files, releasing resources, or disconnecting from a database, where you want to ensure that the resource is properly released regardless of any errors.

Exercise 5: Write a function that asks for an integer and prints the square of it. Use a while loop with a try, except, else block to account for incorrect inputs.