Python Programming

Lecture 9 Closures, Decorators, File

9.1 Closures & Decorators

Closure Function

```
def print_msg(msg):
    def printer():
        print(msg)
    return printer

another = print_msg("zen of python")
print(another())
```

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

When to use closures?

- Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

Outer function

Inner function

Example

```
def lazy_sum(*args):
    def calc_sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return calc_sum

f = lazy_sum(1, 3, 5, 7, 9)
print(f())
print(type(f))
```

```
25
<class 'function'>
```

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier

times3 = make_multiplier_of(3)
times5 = make_multiplier_of(5)

print(times3(9))
print(times5(3))
print(times5(times3(2)))
```

Decorator

- Decorators are "wrappers", which means that they let you execute code before and after the function they decorate without modifying the function itself.

```
def new_decorator(a_function_to_decorate):
    def the_wrapper():
        print("Before the function runs")
        a_function_to_decorate()
        print("After the function runs")
    return the_wrapper

def a_function():
    print("I am a stand alone function.")
```

```
a_function_decorated = new_decorator(a_function)
a_function_decorated()
```

```
Before the function runs
I am a stand alone function.
After the function runs
```

```
@new_decorator
def another_stand_alone_function():
    print("Leave me alone")

another_stand_alone_function()
```

```
Before the function runs
Leave me alone
After the function runs
```

```
def bread(func):
    def wrapper():
        print("<!--''''''\-->")
        func()
        print("<\______/>")
    return wrapper

def ingredients(func):
    def wrapper():
        print("#tomatoes#")
        func()
        print("~salad~")
    return wrapper

def sandwich(food="--ham--"):
    print(food)
```

```
sandwich()
```

```
--ham--
```

```python
@bread
@ingredients # The order matters here.
def sandwich(food="--ham--"):
    print(food)

sandwich()
```

```
<!--''''''\-->
 #tomatoes#
 --ham--
 ~salad~
<\_____/>
```

- Taking decorators to the next level
- Passing arguments to the decorated function

```python
def passing_arguments(function_to_decorate):
    def wrapper(arg1, arg2):
        print(f"I got args! Look: {arg1}, {arg2}")
        function_to_decorate(arg1, arg2)
    return wrapper

@passing_arguments
def print_name(first_name, last_name):
    print(f"My name is {first_name}, {last_name}")

print_name("Peter", "Venkman")
```

```
I got args! Look: Peter, Venkman
My name is Peter, Venkman
```

- If you're making general-purpose decorator--one you'll apply to any function or method, no matter its arguments--then just use *args, **kwargs:

```
def passing_arbitrary_arguments(function_to_decorate):
    def wrapper(*args, **kwargs):
        print("Do I have args?:")
        print(args)
        print(kwargs)
        function_to_decorate(*args, **kwargs)
    return wrapper

@passing_arbitrary_arguments
def function_with_no_argument():
    print("Python is cool, no argument here.")

function_with_no_argument()
```

```
Do I have args?:
()
{}
Python is cool, no argument here.
```

```
@passing_arbitrary_arguments
def function_with_arguments(a, b, c, d):
    print(a, b, c, d)

function_with_arguments(1,2,3,4)
```

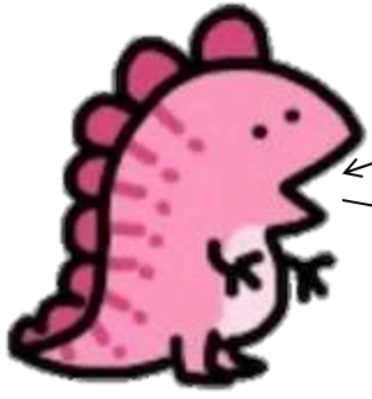
```
Do I have args?:
(1, 2, 3, 4)
{}
1 2 3 4
```

```
function_with_arguments(1,2,3,d = "banana")
```

```
Do I have args?:
(1, 2, 3)
{'d': 'banana'}
1 2 3 banana
```

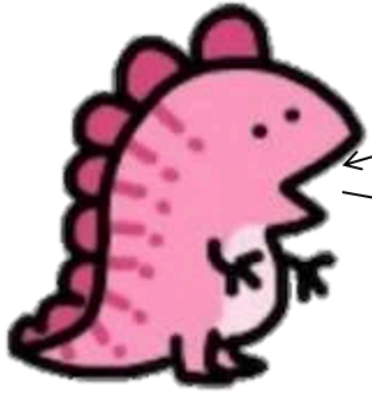
- Decorator Basics

Outter Function Inner Function



msg

msg="zen of python"



x*n

x*3

Closure function

Decorator @ Function

function

Wow!

function

Amazing!

Decorator

9.2 File

Reading from a File

- pi_digits.txt

```
3.1415926535
8979323846
2643383279
```

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()
contents = contents.rstrip()
print(contents)
```

```
3.1415926535
8979323846
2643383279
```

File Path

- relative path

```
path = Path('text_files/filename.txt')
```

- absolute path

```
path = Path('/home/eric/data_files/text_files/filename.txt')

path = Path('c:/text_files/filename.txt')
path = Path(r'c:\text_files\filename.txt')
path = Path('c:\\text_files\\filename.txt')

path = Path('c:\text_files\filename.txt')  #error
```

- Making a List of Lines from a File

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
print(lines)
```

- Working with a File's Contents

```
pi_string = ''
for line in lines:
    pi_string = pi_string + line
print(pi_string)
print(len(pi_string))
```

```
['3.1415926535', '8979323846', '2643383279']
3.141592653589793238462643383279 # string
32
```

Writing to a File

```
1 from pathlib import Path
2
3 path = Path('programming.txt')
4
5 path.write_text("I love programming.")
```

- Python can only write strings to a text file. If you want to store numerical data in a text file, you'll have to convert the data to string format first using the str() function.

```
1 from pathlib import Path
2
3
4 contents = "I love programming.\n"
5 contents += "I love creating new games.\n"
6 contents += "I also love working with data.\n"
7
8 path = Path('programming.txt')
9 path.write_text(contents)
```

```
I love programming.
I love creating new games.
I also love working with data.
```

- Handling the FileNotFoundError Exception

```
1 from pathlib import Path
2
3 path = Path('alice.txt')
4 contents = path.read_text(encoding='utf-8') #Error
```

```
1 from pathlib import Path
2
3 path = Path('alice.txt')
4 try:
5     contents = path.read_text(encoding='utf-8')
6 except FileNotFoundError:
7     print(f"Sorry, the file {path} does not exist.")
```

```
Sorry, the file alice.txt does not exist.
```

- ZeroDivisionError

- try-except-else

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
else:
    # Count the approximate number of words in the file:
    words = contents.split()
    num_words = len(words)
    print(f"The file {path} has about {num_words} words.")
```

JSON

- JSON (JavaScript Object Notation, pronounced /ˈdʒeɪsən/; also /ˈdʒeɪˌsɒn/) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.
- JSON is a language-independent data format. It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data. JSON filenames use the extension .json.

- json.dumps() and json.loads()

```
from pathlib import Path
import json


numbers = [2, 3, 5, 7, 11, 13]

path = Path('numbers.json')
contents = json.dumps(numbers)
path.write_text(contents)
```

```
from pathlib import Path
import json

path = Path('numbers.json')
contents = path.read_text()
numbers = json.loads(contents)

print(numbers)
```

```
[2, 3, 5, 7, 11, 13]
```

- Saving and Reading User-Generated Data

```python
from pathlib import Path
import json

username = input("What is your name? ")

path = Path('username.json')
contents = json.dumps(username)
path.write_text(contents)

print(f"We'll remember you when you come back, {username}!")
```

```python
from pathlib import Path
import json

path = Path('username.json')
contents = path.read_text()
username = json.loads(contents)

print(f"Welcome back, {username}!")
```

```python
from pathlib import Path
import json


def get_stored_username(path):
    """Get stored username if available."""
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        return username
    else:
        return None
```

```python
def get_new_username(path):
    """Prompt for a new username."""
    username = input("What is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    return username
```

```
def greet_user():
    """Greet the user by name."""
    path = Path('username.json')
    username = get_stored_username(path)
    if username:
        print(f"Welcome back, {username}!")
    else:
        username = get_new_username(path)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

```
What is your name? Eric #First time
We'll remember you when you come back, Eric!
```

```
Welcome back, Eric!
```

9.3 Data Reading Examples

Reading Multiple Lines in JSON file

- Yelp: complete Dataset
- Yelp: sample Dataset

```
from pathlib import Path
import json

path = Path('yelp_sample.json')
contents = path.read_text()
lines = contents.splitlines()
yelp = []
for line in lines:
    yelp.append(json.loads(line))

print(len(yelp))
print(yelp[100])
```

Reading Multiple text file

- Marvel Cinematic Universe Dialogue Dataset

```
import os
from pathlib import Path
from textblob import TextBlob #pip install

folder_path = "marvel/"
summary = []
```

```
for filename in os.listdir(folder_path):  #列出所有文件
    # 组成相对路径
    path = Path(os.path.join(folder_path, filename))
    contents = path.read_text(errors="ignore")
    lines = contents.splitlines()
    scores = []

    for line in lines:
        line = line.strip()
        if line:
            # 看看这一行话的情绪是正面、负面，还是中性（+1到-1）
            polarity = TextBlob(line).sentiment.polarity
            scores.append(polarity)
    if scores:
        average = sum(scores) / len(scores)
        summary.append((average, filename))
```

```
summary.sort()
for score, movie in summary:
    print(movie, score)
```

Summary

- File
  - Reading: Python Crash Course, Chapter 10