intro_to_fp

April 9, 2019

1 Functional Programming (FP) in Python

Programming paradigms supported by Python - Object Oriented - Procedural - Imperative - Functional

1.1 What is it and why do I care?

Wikipedia - In computer science, functional programming is a programming paradigm — a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and *avoids changing-state and mutable data*.

- Easier to make concurrent
- More testable

In [44]:
```
i = 0
def foo(x):
    return i + x

def bar(x):
    global i
    i = 10
    return i + x

foo(1)
bar(1)
foo(1)
```

Out[44]:
```
11
```

2 One example, four paradigms

In [25]:
```
# Object Oriented
my_data = [1, 2, 3, 4, 5]

class MyList:
    def __init__(self, data):
        self.data = data
```

```
            self.sum = 0

        def calculate_summation(self):
            self.sum = sum(self.data)

    # Driver code
    my_list = MyList(my_data)
    my_list.calculate_summation()
    my_list.sum
```

Out[25]: 15

In [12]:
```
# Procedural
my_data = [1, 2, 3, 4, 5]

def summation(nums):
    total = 0
    for num in nums:
        total += num
    return total

summation(my_data)
```

Out[12]: 15

In [13]:
```
# Imperative
nums = [1, 2, 3, 4, 5]
total = 0

for num in nums:
    total += num

total
```

Out[13]: 15

In [12]:
```
# Functional
import functools

my_data = [1, 2, 3, 4, 5]

def summation(nums):
    return functools.reduce(lambda x, y: x + y, nums)

summation(my_data)
```

Out[12]: 15

2.1 FP Concepts in Python

- Recursion
- Functions as first class citizens!
- closures
- list and dictionary comprehensions
- lazy vs eager evaluation
- lambda (annonymous functions)

3 Functions as first class citizens (Higher Order Functions)

```
In [1]: # Passing function as argument
        def square(x):
            """Calculates the square of a given number."""
            return x ** 2

        def apply(f, iterable):
            """Applies a function f to every element in the provided iterable."""
            result = []
            for it in iterable:
                result.append(f(it))
            return result

        apply(square, [1, 2, 3, 4, 5])

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

```
In [2]: # Function that returns a function
        def create_greeting(greeting):
            def greet(name):
                return "{} {}".format(greeting, name)
            return greet

        say_hello = create_greeting("Hello")

        say_hello("World!")

Out[2]: 'Hello World!'
```

4 Closure

Theory: A record storing a function with the mapping associating each free variable with its value or reference.

English: A function and the variables in its scope.

```
In [1]: """Contrived Example."""
        def adder(x):
            def _add(y):
```

```
            return x + y
        return _add

    add_five = adder(5)
    add_five(10)
```

Out[1]: 15

In [33]:
```
"""Real World Example"""
import logging
import queue
import time
from concurrent.futures import ThreadPoolExecutor

def done(logger, queue):
    # Our closure keeps the value of our logger
    # and queue inside of the _done function below
    def _done(func):
        if func.cancelled():
            logger.info('future canceled')
        elif func.done():
            error = func.exception()
            if error:
                logger.info('future error: {}'.format(error))
            else:
                queue.put(func.result())
    return _done


def bamboozle(duration):
    print("Muhahaha. You have been bamboozled for {}!".format(duration))
    time.sleep(duration)
    return duration


durations = [1, 2, 3, 2, 1, 5]
results_queue = queue.Queue(maxsize=len(durations))
executor = ThreadPoolExecutor(max_workers=5)
queue_logger = logging.getLogger("example")
done_callback = done(queue_logger, results_queue)

for duration in durations:
    future = executor.submit(bamboozle, duration)
    future.add_done_callback(done_callback)

while True:
    if results_queue.full():
        print('Finished bamboozling')
```

```
                break

        executor.shutdown()
```

```
Muhahaha. You have been bamboozled for 1!
Muhahaha. You have been bamboozled for 2!Muhahaha. You have been bamboozled for 3!

Muhahaha. You have been bamboozled for 2!
Muhahaha. You have been bamboozled for 1!
Muhahaha. You have been bamboozled for 5!
Finished bamboozling
```

```
In [22]: """Decorators Demystified! They are just closures!"""
         import functools
         import time

         def timeit(function):
             """Records the time a function takes to execute."""
             def wrapper():
                 start = time.time()
                 result = function()
                 end = time.time()
                 diff = end - start
                 print("{} took {} seconds".format(function.__name__, diff))
                 return result
             return wrapper

         def sleep_two():
             time.sleep(2)


         timed_sleep = timeit(sleep_two)
         timed_sleep()
```

```
sleep_two took 2.0027852058410645 seconds
```

```
In [ ]: # It can get very verbose doing this for every function we want to decorate.
        # Here's some syntatic sugar for cleaner code.
        def timeit(function):
            """Records the time a function takes to execute."""
            def wrapper():
                start = time.time()
                return function()
                end = time.time()
                diff = end - start
                print("{} took {} seconds".format(function.__name__, diff))
```

```
    return wrapper

@timeit
def sleep_one():
    time.sleep(1)


sleep_one()
```

In [26]:
```
import functools
import time

# https://docs.python.org/3/library/functools.html#functools.update_wrapper
# https://docs.python.org/3/library/functools.html#functools.wraps
def timeit(function):
    """Records the time a function takes to execute."""
    @functools.wraps(function)
    def wrapper():
        start = time.time()
        result = function()
        end = time.time()
        diff = end - start
        print("{} took {} seconds".format(function.__name__, diff))
        return result
    return wrapper

@timeit
def sleep_two():
    time.sleep(2)


sleep_two
```

Out[26]:
```
<function __main__.sleep_two()>
```

In [35]:
```
# This is nice and all but what if we want to pass arguments to our decorated functio
# Since our decorator returns a function, we can have that function expect a variable
# of positional arguments and keyword arguments i.e. give it the function defintion w
# *args and **kwargs.


def timeany(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        start = time.time()
        return function(*args, **kwargs)
        end = time.time()
        diff = end - start
```

```
            print("{} took {} seconds".format(function.__name__, diff))
        return wrapper


    def my_decorator(f):
        @functools.wraps(f)
        def wrapper(*args, **kwargs):
            print('calling decorated function')
            return f(*args, **kwargs)
        return wrapper

    @my_decorator
    def example(x, y, greeting='Salutations'):
        print(greeting)
        return x + y

    example(1, 2, greeting='Hello World!')
```

```
calling decorated function
Hello World!
```

Out[35]: 3

5 Lazy vs eager evaluation and list/dictionary comprehensions

```
def lazy_numbers(n):
    num = 0
    while num < n:
        yield num
        num += 1

def eager_numbers(n):
    nums = []
    while num < n:
        nums.append(n)
        n += 1
    return nums
```

5.1 Questions

- What is this yield keyword?
- What is the difference between the lazy and eager example here?
- What happens if n is REALLY big?
- What if each element in the list is REALLY big?

In [20]:
```
# Comprehensions
```

```
# List
def nats_list(n):
    return [i for i in range(n)]

# Generator
def nats_gen(n):
    return (i for i in range(n))

# Dictionary
def nats_to_square(n):
    """Maps first n integers (0 indexed) to their square."""
    return {i: i ** 2 for i in range(n)}


print(type(nats_list(5)))
print(type(nats_gen(5)))
print(type(nats_to_square(5)))
```

```
<class 'list'>
<class 'generator'>
<class 'dict'>
```

In [21]:
```
# Using If-Condition
def evens_list(n):
    return [i for i in range(n) if i % 2 == 0]


def events_gen(n):
    return (i for i in range(n) if i % 2 == 0)
```

In [13]:
```
"""lazy_numbers is an example of a generator in Python.

`yield` keyword suspends function execution and returns the current value back to the
Python saves the enough information i.e. the stack frame so it can pick up where it l

All of eager_numbers is being written into memory which for large n or large elements
means it can consume more memory that you would like.
"""
```

Out[13]:
```
'lazy_numbers is an example of a generator in Python.\n\n`yield` keyword suspends func
```

In [29]:
```
# Annonymous Functions
users = [{
  "id": 1,
  "first_name": "Sharity",
  "last_name": "Latus",
  "email": "slatus0@ft.com",
  "ip_address": "166.36.139.73"
```

```
}, {
  "id": 2,
  "first_name": "Polly",
  "last_name": "Fautly",
  "email": "pfautly1@deviantart.com",
  "ip_address": "223.169.73.214"
}, {
  "id": 3,
  "first_name": "Elliott",
  "last_name": "Geratt",
  "email": "egeratt2@dailymail.co.uk",
  "ip_address": "63.101.181.87"
}, {
  "id": 4,
  "first_name": "Darcee",
  "last_name": "Stenton",
  "email": "dstenton3@bandcamp.com",
  "ip_address": "129.82.4.219"
}, {
  "id": 5,
  "first_name": "Dorey",
  "last_name": "Bastistini",
  "email": "dbastistini4@pen.io",
  "ip_address": "255.220.99.177"
}]

sorted(users, key=lambda user: user['first_name'])
```

```
[{'id': 4, 'last_name': 'Stenton', 'email': 'dstenton3@bandcamp.com', 'first_name': 'Darcee',
```

In [30]:
```
sorted(users, key=lambda user: user['ip_address'])
```

Out[30]:
```
[{'email': 'dstenton3@bandcamp.com',
  'first_name': 'Darcee',
  'id': 4,
  'ip_address': '129.82.4.219',
  'last_name': 'Stenton'},
 {'email': 'slatus0@ft.com',
  'first_name': 'Sharity',
  'id': 1,
  'ip_address': '166.36.139.73',
  'last_name': 'Latus'},
 {'email': 'pfautly1@deviantart.com',
  'first_name': 'Polly',
  'id': 2,
  'ip_address': '223.169.73.214',
  'last_name': 'Fautly'},
```

```
          {'email': 'dbastistini4@pen.io',
           'first_name': 'Dorey',
           'id': 5,
           'ip_address': '255.220.99.177',
           'last_name': 'Bastistini'},
          {'email': 'egeratt2@dailymail.co.uk',
           'first_name': 'Elliott',
           'id': 3,
           'ip_address': '63.101.181.87',
           'last_name': 'Geratt'}]
```

```
In [32]: list(filter(lambda user: user['id'] < 3, users))
```

```
Out[32]: [{'email': 'slatus0@ft.com',
           'first_name': 'Sharity',
           'id': 1,
           'ip_address': '166.36.139.73',
           'last_name': 'Latus'},
          {'email': 'pfautly1@deviantart.com',
           'first_name': 'Polly',
           'id': 2,
           'ip_address': '223.169.73.214',
           'last_name': 'Fautly'}]
```