LECTURER: TAI LE QUY

OBJECT ORIENTED AND FUNCTIONAL PROGRAMMING WITH PYTHON

Thanks Prof. Dr. Max Pumperla for his contribution

TOPIC OUTLINE

Object oriented programming	1 + 2
Functional programming	3
Projects and testing in Python	4
Working with Database in Python	5
Documenting a project	6

UNIT 3

FUNCTIONAL PROGRAMMING

STUDY GOALS

- Functional aspects of Python
- Higher-order functions
- More functional aspects of Python
  - Function composition
  - Currying
  - Generators
  - Comprehensions
  - Monad
- Examples

- FP has a long tradition
  - rooted in logic (lambda calculus)
- Pure functions
  - no side effects
  - deterministic
  - referential transparency
  - inputs, some processing, outputs
- Data are immutable
- Functions are composable

```python
def add(a, b):
    return a+b

def add_and_print(a, b):
    #or write to files
    print("Adding!")
    return a+b
```

- Mutable:
  - lists, sets, dictionaries, etc.
- Immutable:
  - Tuples, frozensets, strings, numbers

```
#Tuples are immutable
person = ("Paul", 29, "USA")

#Sets are mutable
nunber_set = set([1,2,3])

#Frozen sets are immutable
immutable_numbers = frozenset([1,2,3,4])

#lists are mutable
number_list = [1,2,3,4]
```

- Functions operating on functions
- Examples
  - **map**: applies a function to all the items in an input_list
  - **reduce**: performing some computation on a list and returning the result
  - **filter**: creates a list of elements for which a function returns true

```
from functools import reduce
numbers = [1,2,3,4,5]
squared_numbers = list(map(lambda x:x**2,numbers))
print("Squared numbers:",squared_numbers)
#Squared numbers: [1, 4, 9, 16, 25]
even_numbers = list(filter(lambda x:x % 2 ==0, numbers))
print("Even numbers: ", even_numbers)
#Even numbers:  [2, 4]
product = reduce(lambda x,y:x*y,numbers)
print("Product: ", product)
#Product:  120
```

- Lambdas for quick function definition
  - used for anonymous functions
- List comprehensions to create new lists
  - works for sets, dictionary, etc., too

```python
add = lambda a,b: a+ b
square = lambda x: x**2

print(add(4,5)) #9
print(square(5)) #25

numbers = [1,2,3,4,5]
squares = [x**2 for x in numbers]
print("Squares: ", squares)
#Squares:  [1, 4, 9, 16, 25]
even_numbers = [x for x in numbers if x%2 ==0]
print("Even numbers: ", even_numbers)
#Even numbers:  [2, 4]
```

EXTENDED PYTHON EXAMPLE

```python
from functools import reduce
#Using recursive function
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
#Using reduce & lambda
def calculate_factorial(num):
    numbers = range(1, num+1)
    result = reduce(lambda x,y: x*y,numbers)
    return result
#Main program
number = 5
factorial_result = factorial(number)
print(f"The factorial using pure function: {factorial_result}")

factorial_result_fp = calculate_factorial(number)
print(f"The factorial using high-order functions: {factorial_result_fp}")
```

- Combining two or more functions in such a way that the output of one function becomes the input of the second function and so on

```python
def add_one(x):
    return x+1
def square(x):
    return x**2
def compose(f, g):
    return lambda x: f(g(x))
#main program
add_one_then_square = compose(square,add_one)
number = 5
result = add_one_then_square(number)
print(f"Result square(add_one({number})): ",result)
```

- Currying: create new functions from a function that takes multiple arguments, each derived function will then take only a single argument
- partial() function that allows us to use a partial application of a function in a more simplified way

```python
#Currying
def multiply_numbers(a, b):
    return a * b
def multiply_by_three(a):
    return multiply_numbers(a, 3)
print(multiply_by_three(5))
#15
```

```python
#Partial function
from functools import partial
def multiply_numbers(a, b):
    return a * b
multiply_by_three =
partial(multiply_numbers, 3)
print(multiply_by_three(5))
#15
```

- **Yield** is used in Python generators. If the body of a def contains **yield**, the function automatically becomes a generator function.

```python
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
fib_sequence = fibonacci()
for i in range(10):
    print(next(fib_sequence))
# Output: 0 1 1 2 3 5 8 13 21 34
```

```python
squares = [x**2 for x in range(1,11)]
print(squares)
#[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
squares_dict = {x: x**2 for x in range(1,6)}
print(squares_dict)
#{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
even_set = {x for x in range(1,11) if x % 2 ==0}
print(even_set)
#{2, 4, 6, 8, 10}
#Generator of squares of numbers from 1 to 10
squares_gen = (x** 2 for x in range(1,11))
print(tuple(squares_gen))
#(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

- Monad is a functional programming design pattern that enables you to **combine several calculations** or functions into a **single expression** while also **managing error** circumstances and side effects. Every function in the chain should, in theory, return a new monad that may be used as input by the function after it.
- **Maybe** Monad: Represents a computation that may or may not return a value. This is useful for handling error conditions or optional values.

```python
class Maybe:
    def __init__(self,value):
        self.value = value
    def bind(self,func):
        if self.value is None:
            return Maybe(None)
        else:
            return func(self.value)
    def __repr__(self):
        return str(self.value)
def square_root(x):
    if x>=0:
        return Maybe(x**0.5)
    else:
        return Maybe(None)
def reciprocal(x):
    if x!=0:
        return Maybe(1/x)
    else:
        return Maybe(None)
def compute_result(x):
    result = Maybe(x)
    result = result.bind(square_root)
    result = result.bind(reciprocal)
    return result
#Main program
value = 4
result = compute_result(value)
print(result) #Output: 0.5
```

Unit 2

TRANSFER TASK

TRANSFER TASK

1. Write a program to return a distinct values from a list

    E.g., given a list [1, 2, 2, 3, 3, 4, 5], return [1,2,3,4,5]

2. Write a program to compute factorial of a number, using **yield** keyword in a generator function

Please present your results.
The results will be discussed in plenary.