Programming Concepts and Languages

Spring 2024

Functional Python

- Is this a pure function?

```
G = 4
def sum1(n):
    # add two numbers
    return n + G
```

- Given

```
coffee_price = [(2019, 25.05), (2020, 26.03),
  (2021, 25.12), (2022, 25.02), (2023, 24.08)]
```

  - What is the result of **`max(coffee_price)`** **???**

Learning Objectives

- By the end of this session, you should be able to:
  - ✓ explain the fundamentals of functional programming in Python including
    - ✓ lambda expressions
    - ✓ recursion
    - ✓ pure functions
    - ✓ immutability
    - ✓ higher order functions
      - ✓ map(), filter(), reduce(), max(), etc.
  - ✓ implement simple Python programs using the above mentioned

Functional Programming Paradigm

- Recall from previous lessons:
  - Functional programming uses functions as the main building blocks
- Python supports most of the central features of the functional programming paradigm including:
  - Pure functions
  - Lambda
  - Recursion
  - Immutable data
  - Higher-order functions

Functions

- a function is a named sequence of statements that performs a desired operation

```
def FUNNAME( LIST OF PARAMETERS ):
    STATEMENTS
```

- Example: Define in Python: $f(x) = 3x^2 - 2x + 5$

```
def f(x):
    return 3 * x ** 2 - 2 * x + 5
```

- Is this a pure function?

```
G = 4
def sum1(n):
    # add two numbers
    return n + G
```

- .

Pure functions

- In functional programming, functions can be stateless.
- Pure function communicates with the calling program only through parameters
- We can define pure functions – free of side effects

```
def pure_fun_sum(x, y):
    # adds two numbers
    # uses only the local function inputs
    return x + y
```

- Adds two numbers using only the local function inputs

Lambda expression

- We can use a λ (lambda) expression to define functions instead of the def syntax for functions
- When using lambda, you use three different operations to perform tasks
  - Creating functions to pass as variables
  - Binding a variable to the expression
  - Applying a function to an argument

- Add two numbers using lambda

```
lambda_add = lambda x, y: x + y
```

Lambda - callbacks

- Lambda for callbacks:

```
>>> def say_hello_01(name):
...     print('Hellooooo', name)
...
>>> say_hello_02 = lambda name: print('Hellooooo', name)
>>> say_hello_01('Mikkel')
Hellooooo Mikkel
>>> say_hello_02('Adrian')
Hellooooo Adrian
>>> say_hello_01.__qualname__
'say_hello_01'
>>> say_hello_02.__qualname__
'<lambda>'
```

- GUI callbacks
  - Tkinter ...

```
Button(win, text='Show',
    command=(lambda:messagebox.showinfo(
                    'Paradigm', combo.get())))
```

Recursion

- Functional paradigm do away with loops and the overhead of tracking the loops state
- Instead, it relies on the recursive function approach
- A recursive function is a function that calls itself.

```
>>> def main():
...     print_message()
...
>>> def print_message():
...     print("This is a recursive function.")
...     print_message()
```

Problem Solving Using Recursion

- All recursive functions have the following characteristics
  - It is implemented with an if-else that leads to different cases
  - One or more base cases are used to stope the recursion
- Example: We can define addition of two natural numbers recursively:

$$\text{add}(a,b) = \begin{cases} b & \text{if } a = 0 \\ \text{add}(P(a), S(b)) & \text{if } a \neq 0 \end{cases}$$

```
def add(a,b):
    if a == 0: return b
    else: return add(a-1, b+1)
```

Summing a Range of List with Recursion

- Example: The function takes a list that contains the range of elements to be summed with the starting and ending indices – *pcl_range_sum(lst, start, end)*

```
def pcl_range_sum(num_list, start, end):
    if start > end:
        return 0
    else:
        return num_list[start] + pcl_range_sum(num_list,
start + 1, end)
```

The Fibonacci Series

- Some mathematical problems are designed to be solved recursively.
- Example: Fibonacci series:

```
The series: 0  1  1  2  3  5  8  13  21  34  55  89 . . .
```

  - Can be defined as follows:

$$\begin{aligned} &\text{If } n = 0 \text{ then} && \text{Fib}(n) = 0 \\ &\text{If } n = 1 \text{ then} && \text{Fib}(n) = 1 \\ &\text{If } n > 1 \text{ then} && \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \end{aligned}$$

Tail Recursion

- Recall that a recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.
- A tail recursive function is efficient for reducing stack size

```
Recursive function A
    ...
    ...

    ...
    Invoke function A recursively
```

(a) Tail recursion

```
Recursive function B
    ...
    ...
    Invoke function B recursively
    ...
    ...
```

(b) Nontail recursion

Tail Recursion – Check Prime

- A prime number is a natural number, evenly divisible by only 1 and itself.
- coprime to mean that two numbers have only 1 as their common factor. The numbers 2 and 3, for example, are coprime

$$prime(n) = \forall x \left[\left(2 \le x < 1 + \sqrt{n} \right) and \left(n (\text{mod } x) \neq 0 \right) \right]$$

$$\text{prime}(n) = \neg \text{coprime}\left(n, \left[2, 1 + \sqrt{n}\right)\right), \text{ given } \texttt{n} \texttt{ > } \texttt{1.}$$

$$\text{coprime}(n, [a, b)) = \begin{cases} \text{True} & \textbf{if } a = b \\ n \pmod{a} \neq 0 \wedge \text{coprime}(n, [a+1, b)) & \textbf{if } a < b \end{cases}$$

Immutable Data

- Python offers some immutable data types
  - Tuple is is one of the popular types that are immutable.
- Example: Tuple vs List
  - Given the following collections:

```
mutadata = [‘Alfonsy', 123456, ['PME1', 'PCL1', 'ALI1']]
immudata = (‘Alfonsy', 123456, ['PME1', 'PCL1', 'ALI1'])
```

  - Read from the data types and print out only the list of courses.

`print(mutadata[2])` **gives** `['PME1', 'PCL1', 'ALI1']`
`print(immudata[2])` **gives** `['PME1', 'PCL1', 'ALI1']`

  - Observation
    - Reading from the data types are essentially the same.

Immutable Data II

- Example: Tuple vs List
  - Given the following collections:

```
mutadata = [‘Alfonsy', 123456, ['PME1', 'PCL1', 'ALI1']]
immudata = (‘Alfonsy', 123456, ['PME1', 'PCL1', 'ALI1'])
```

  - Change the student number from *123456* to *110220*.

mutadata[1] = 110220 gives

```
['Alfonsy', 110220, ['PME1', 'PCL1', 'ALI1']]
```

immudata[1] = 110220 gives

```
Traceback (most recent call last):  File "<stdin>", line 1, in <module>TypeError: 'tuple' object does not support item assignment
```

  - Observation
    - It fails with tuple

Higher-Order Functions

- Higher-order functions can be used to write expressive and succinct programs.
- accept a function as an argument or return a function as a value.
- We can use higher-order functions as a way to create composite functions from simpler functions
- Some commonly used Python's built-in higher-order functions
  - max()
  - map()
  - filter()
  - reduce()

Higher-Order Functions – max()

- Python supports lambda and higher order functions
- Example:
  - Python max() function is a higher order function
    - Accepts a function as an argument and return a function.
  - Find maximum VIA coffee price in the last 5 years given:

```
coffee_price = [(2019, 25.05), (2020, 26.03),
   (2021, 25.12), (2022, 25.02), (2023, 24.08)]
```

  - What is the result?

```
max(coffee_price) ??? ➔ (2023, 24.08)
```

  - Observation
    - compares each tuple in the sequence and returns largest value on position 0.

Higher-Order Functions – max() with lambda

- Python supports lambda and higher order functions
- Example:
  - Python max() function is a higher order function
    - Accepts a function as an argument and return a function.
  - Find maximum VIA coffee price in the last 5 years

```
coffee_price = [(2019, 25.05), (2020, 26.03),
(2021, 25.12), (2022, 25.02), (2023, 24.08)]
```

  - with lambda as argument

```
max(coffee_price, key=lambda cprice: cprice[1])
```

  - returns `(2020, 26.03)`

Higher-Order Functions – map()

- The map() function takes an iterable (list) creates a new iterable map object with the function applied to every element.
- Example:
  - Adding 1 to every element in a list:

```
list_values = [2018, 2019, 2020, 2021, 2022, 2023]

add_one = list(map(lambda y: y + 1, list_values ))
```

➔ `[2019, 2020, 2021, 2022, 2023, 2024]`

  - Observation
    - The returned map object is converted to a list data structure.

Higher-Order Functions – filter()

- The filter() function takes an iterable (list) creates a new iterable map object with the function that returns a boolean value.
- Example:
  - Filter odd values given a list:

```
lst2 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
odd_num = list(filter(lambda n: n % 2 == 1, lst2))
```

`odd_num` ➔ `[1, 3, 5, 7, 9]`

  - Observation
    - The returned map object is converted to a list data structure.

Higher-Order Functions – reduce()

- The reduce() function from the functools package takes an iterable (list), and reduces the iterable to a single value.
- different from filter() and map() as it takes a function with two input values.

- Example:
  - Use reduce() to sum all elements in a given a list:

```
vlst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

summed_val = reduce(lambda n, m: n + m, vlst)
```

  - Observation
    - You do not have to operate on the second value in the lambda.
    - Example: a function that always returns the first value of an iterable

```
first_val = reduce(lambda n, _: n, vlst)    ➔ 0
```

List Comprehensions

- The provides a syntax for making lists through list comprehensions.
- with this, map() and filter() can be translated to list comprehensions

- Map example:
  - increment by 1 using list comprehension:

```
vlst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

inc_vlst = [i + 1 for i in vlst]
```

- Filter Example:
  - Filter even numbers in the list:

```
even_filter = [i for i in vlst if i % 2 == 0]
```

Functions as Objects– sorted()

- The sorted() function takes an iterable (list) creates a new iterable map object with the list sorted.
- Example:

```
gtg_sales =[('Coffee', 2018, 525.05),
            ('Juice', 2021, 526.03),
            ('Apple', 2020, 525.12),
            ('Green Tea', 2019, 525.02),
            ('Banana', 2022, 524.08)]
```

```
sorted_in_order = sorted(gtg_sales)
```

```
sorted_in_order ➔
  [('Apple', 2020, 525.12), ('Banana', 2022, 524.08),
   ('Coffee', 2018, 525.05), ('Green Tea', 2019, 525.02),
   ('Juice', 2021, 526.03)]
```

- Observation
  - The list is sorted by the first value in the tuple (name)