python for Computational Problem Solving - pCPS - Functional_Programming_Testing Lecture Slides - Class #38

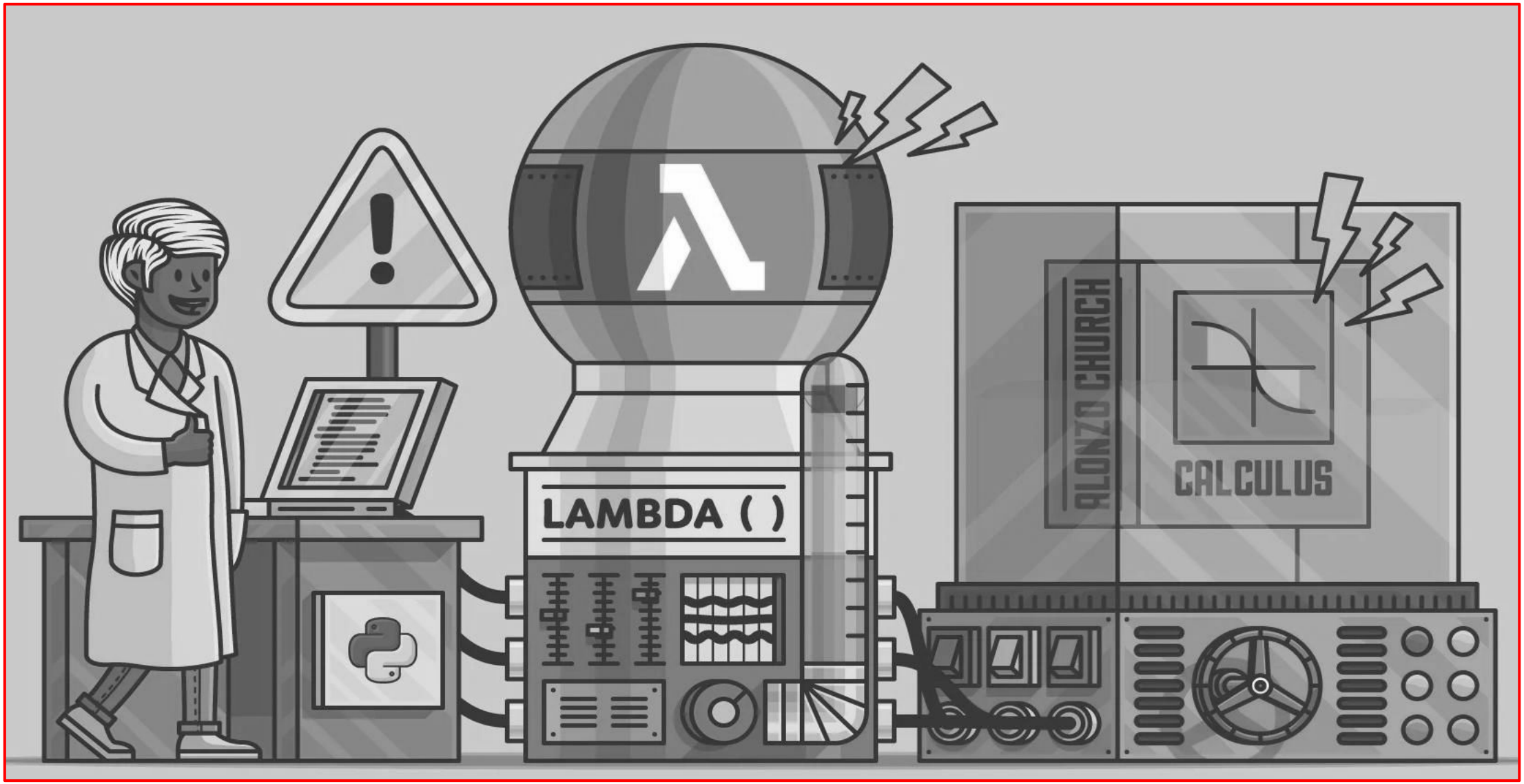
Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

python for Computational Problem Solving Syllabus

Unit IV: Functional Programming, Modules,Testing and Debugging - 10 Hours

- Functional Programming - map, filter, reduce, max, min, lambda function
- list comprehension,
- Modules - import mechanisms
- Testing
  - Pytest , Function testing with Doctest
  - pdb debugger commands.

Functional Programming in python



Functional Programming in python

- In python, **functions** are treated on par with strings and numbers, meaning anything you would expect to be able to do with a string or number you can do with a function as well.

- When you pass a **function** to **another** function, the **passed-in function** sometimes is referred to as a **callback** because a call back to the inner function can **modify** the **outer** function's **behavior**.

```
def F1():
    print('I am f1()')


F1()

Alias = F1
F1()
Alias()

F1 = 100

print('I understood', Alias, 'Now F1 is -->',F1)
```

```
I am f1()
I am f1()
I am f1()
I understood <function F1 at 0x7f7f2f5f5dc0> Now F1 is --> 100
```

```
def Inner():
    return ('I am the function Named Inner')

def Outer(Alias):
    print(Alias())

Outer(Inner)
```

```
I am the function Named Inner
```

Functional Programming in python

```
def F1():
    return ('I am happy')


F1()
d ={F1:'I am Happy', 'Alias1':F1, 'Alias2':F1, 'Alias3':F1}

print(d.keys())
print(d.values())

print(d[F1])
print(d['Alias1']())
print(d['Alias2']())
print(d['Alias3']())
```

```
dict_keys([<function F1 at 0x7f7f2f5f58b0>, 'Alias1', 'Alias2', 'Alias3'])
dict_values(['I am Happy', <function F1 at 0x7f7f2f5f58b0>, <function F1 at 0x7f7f2f5f58b0>, <function F1 at 0x7f7f2
f5f58b0>])
I am Happy
I am happy
I am happy
I am happy
```

```
def F1():
    return ('I am happy')


F1()
d ={F1:'I am Happy', 'Alias1':F1(), 'Alias2':F1(), 'Alias3':F1()}

print(d.keys())
print(d.values())

print(d[F1])
print(d['Alias1'])
print(d['Alias2'])
print(d['Alias3'])
```

```
dict_keys([<function F1 at 0x7f7f2f5f5dc0>, 'Alias1', 'Alias2', 'Alias3'])
dict_values(['I am Happy', 'I am happy', 'I am happy', 'I am happy'])
I am Happy
I am happy
I am happy
I am happy
```

Functional Programming in python

- In python, **functions** are treated on par with strings and numbers, meaning anything you would expect to be able to do with a string or number you can do with a function as well.
- If you **pass** a **list** of **string** values to **sorted**(), then it **sorts** them in **lexical** order
- **sorted()** takes an **optional key argument** that **specifies** a **callback** function that can **serve** as the **sorting** key
- **sorted()** can also take an optional argument that **specifies** sorting in **reverse** order.
- But you could manage the same thing by defining your own **callback** function that **reverses** the sense of **len()**

```
List1 = ['A','a','AA','aa','AAA','aaa','AAAA','aaaa']
print(List1)
print(sorted(List1))
print(sorted(List1,key=len))
```

```
['A', 'a', 'AA', 'aa', 'AAA', 'aaa', 'AAAA', 'aaaa']
['A', 'AA', 'AAA', 'AAAA', 'a', 'aa', 'aaa', 'aaaa']
['A', 'a', 'AA', 'aa', 'AAA', 'aaa', 'AAAA', 'aaaa']
```

```
List1 = ['A','a','AA','aa','AAA','aaa','AAAA','aaaa']
print(List1)
print(sorted(List1,reverse=True))
print(sorted(List1,key=len, reverse=True))
```

```
['A', 'a', 'AA', 'aa', 'AAA', 'aaa', 'AAAA', 'aaaa']
['aaaa', 'aaa', 'aa', 'a', 'AAAA', 'AAA', 'AA', 'A']
['AAAA', 'aaaa', 'AAA', 'aaa', 'AA', 'aa', 'A', 'a']
```

```
List2 = [1,-4,5,3,2]
print(List2)
print(sorted(List2))
print(sorted(List2,reverse=True))
```

```
[1, -4, 5, 3, 2]
[-4, 1, 2, 3, 5]
[5, 3, 2, 1, -4]
```

Defining an Anonymous Function With lambda in python

- **Functional programming** is all about calling functions and passing them around, so it naturally involves defining a lot of functions
- It's sometimes **convenient** to be able to define an **anonymous function** on the fly, without having to give it a **name**.
- In python, you can do this with a **lambda expression**.
- The syntax of a lambda expression is
  lambda <parameter_list>: <expression>
- The **value** of a **lambda** expression is a **callable function**, just like a function defined with the def keyword.
- It **takes arguments**, as specified by **<parameter_list>**, and **returns** a **value**, as **indicated** by **<expression>**
- It's **not necessary** to assign a **variable** to a **lambda** expression before calling it.
- You can also **call** the function defined by a **lambda expression** directly

The following table summarizes the parts of a `lambda` expression:

Component	Meaning
`lambda`	The keyword that introduces a `lambda` expression
`<parameter_list>`	An optional comma-separated list of parameter names
`:`	Punctuation that separates `<parameter_list>` from `<expression>`
`<expression>`	An expression usually involving the names in `<parameter_list>`

Defining an Anonymous Function With lambda in python

```
print('Output of the code segment')
print(lambda MyWay:MyWay[::-1])
print(callable(lambda MyWay:MyWay[::-1]))

Reverse = lambda MyWay:MyWay[::-1]
print(Reverse)

Input = input('Enter an Input-->')
print('Given Input type is-->',type(Input))
print('Given Input is-->',Input)

Reversed =Reverse(Input)

print('Input Reversed is',Reversed)

if Input== Reversed:
    print(Input,' is a palindrome')
else:
    print(Input,' is not a palindrome')
```

```
Output of the code segment
<function <lambda> at 0x7f7f2f4414c0>
True
<function <lambda> at 0x7f7f2f1100d0>
Enter an Input-->1,2,3,4,5
Given Input type is--> <class 'str'>
Given Input is--> 1,2,3,4,5
Input Reversed is 5,4,3,2,1
1,2,3,4,5  is not a palindrome
```

```
Output of the code segment
<function <lambda> at 0x7f7f2f441790>
True
<function <lambda> at 0x7f7f2f441a60>
Enter an Input-->[1,2,3,4,5]
Given Input type is--> <class 'str'>
Given Input is--> [1,2,3,4,5]
Input Reversed is ]5,4,3,2,1[
[1,2,3,4,5]  is not a palindrome
```

```
 <function <lambda> at 0x7f7f2f4418b0>
 True
 <function <lambda> at 0x7f7f2f4411f0>
 Enter an Input-->nitin
 Given Input type is--> <class 'str'>
 Given Input is--> nitin
 Input Reversed is nitin
 nitin  is a palindrome
```

```
Output of the code segment
<function <lambda> at 0x7f7f2f4414c0>
True
<function <lambda> at 0x7f7f2f110c10>
Enter an Input-->11.11
Given Input type is--> <class 'str'>
Given Input is--> 11.11
Input Reversed is 11.11
11.11  is a palindrome
```

```
Output of the code segment
<function <lambda> at 0x7f7f2f441820>
True
<function <lambda> at 0x7f7f2f4414c0>
Enter an Input-->(1,2,3,4,5)
Given Input type is--> <class 'str'>
Given Input is--> (1,2,3,4,5)
Input Reversed is )5,4,3,2,1(
(1,2,3,4,5)  is not a palindrome
```

```
Output of the code segment
<function <lambda> at 0x7f7f2f110160>
True
<function <lambda> at 0x7f7f2f4418b0>
Enter an Input-->1234512
Given Input type is--> <class 'str'>
Given Input is--> 1234512
Input Reversed is 2154321
1234512  is not a palindrome
```

Defining an Anonymous Function With lambda in python

```
print((lambda MyWay:MyWay[::-1])('PES University'))
print((lambda MyWay:MyWay[::-1])('nitin'))
print((lambda MyWay:MyWay[::-1])(12345))
```

```
ytisrevinU SEP
nitin

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_6780/2287956518.py in <module>
      1 print((lambda MyWay:MyWay[::-1])('PES University'))
      2 print((lambda MyWay:MyWay[::-1])('nitin'))
----> 3 print((lambda MyWay:MyWay[::-1])(12345))

/tmp/ipykernel_6780/2287956518.py in <lambda>(MyWay)
      1 print((lambda MyWay:MyWay[::-1])('PES University'))
      2 print((lambda MyWay:MyWay[::-1])('nitin'))
----> 3 print((lambda MyWay:MyWay[::-1])(12345))

TypeError: 'int' object is not subscriptable
```

```
print((lambda MyWay:MyWay[::-1])('PES University'))
print((lambda MyWay:MyWay[::-1])('nitin'))
print((lambda MyWay:MyWay[::-1])('12345'))
print((lambda MyWay:MyWay[::-1])([1,2,3,4]))
print((lambda MyWay:MyWay[::-1])((4,3,2,1)))
print((lambda MyWay:MyWay[::-1])(str(12345)))
print((lambda MyWay:MyWay[::-1])(list(12345)))
```

```
ytisrevinU SEP
nitin
54321
[4, 3, 2, 1]
(1, 2, 3, 4)
54321

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_6780/3812667719.py in <module>
      5 print((lambda MyWay:MyWay[::-1])((4,3,2,1)))
      6 print((lambda MyWay:MyWay[::-1])(str(12345)))
----> 7 print((lambda MyWay:MyWay[::-1])(list(12345)))

TypeError: 'int' object is not iterable
```

```
List1 = [1,2,3,4,5]
print((lambda MyWay:MyWay[::-1])(List1))

Tuple1 = ((1,2),(3,4),(-1,-3),(40,50))
print((lambda MyWay:MyWay[::-1])(Tuple1))
```

```
[5, 4, 3, 2, 1]
((40, 50), (-1, -3), (3, 4), (1, 2))
```

Defining an Anonymous Function With lambda in python

- **Functional programming** is all about calling functions and passing them around, so it naturally involves defining a lot of functions
- A **lambda expression** will typically have a parameter list, but it's **not required**.
- You can define a **lambda** function **without parameters**.
- The **return** value is then **not dependent** on any input parameters

```
List1 = ['A','a','AA','aa','AAA','aaa','AAAA','aaaa']

print(sorted(List1, key = lambda MyWay:len(MyWay)))
print(sorted(List1, key = lambda MyWay:-len(MyWay)))
print('----------------------------------------------------')

print(sorted(List1, key = lambda MyWay:len(MyWay)*0))
print(sorted(List1, key = lambda MyWay:-len(MyWay)*0))
```

```
['A', 'a', 'AA', 'aa', 'AAA', 'aaa', 'AAAA', 'aaaa']
['AAAA', 'aaaa', 'AAA', 'aaa', 'AA', 'aa', 'A', 'a']
----------------------------------------------------
['A', 'a', 'AA', 'aa', 'AAA', 'aaa', 'AAAA', 'aaaa']
['A', 'a', 'AA', 'aa', 'AAA', 'aaa', 'AAAA', 'aaaa']
```

```
Mydefault = lambda:'PES University'
Mydefault()
```

```
'PES University'
```

Defining an Anonymous Function With lambda in python

- you can **return** a **tuple** from a **lambda** function. You just have to denote the tuple explicitly with **parentheses**.
- You can also return a **list** or a **dictionary** from a **lambda** function
- A **lambda expression** has its **own local** namespace, so the parameter names don't conflict with identical names in the global namespace.
- A **lambda** expression can **access variables** in the **global namespace**, but it **can't modify** them

```
TR = (lambda Parameter:(Parameter, Parameter+100,Parameter+300))
print(type(TR(10)))
print(TR(10))
```

```
<class 'tuple'>
(10, 110, 310)
```

```
TL = lambda Parameter:[Parameter, Parameter+100,Parameter+300]
print(type(TL(20)))
print(TL(20))
```

```
<class 'list'>
[20, 120, 320]
```

```
TD = lambda Parameter:{1:Parameter, 2:Parameter+100,3:Parameter+300}
print(type(TD(30)))
print(TD(30))
```

```
<class 'dict'>
{1: 30, 2: 130, 3: 330}
```

End of class #38

Thank you

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University
nitin.pujari@pes.edu

For Course Digital Deliverables visit www.pesuacademy.com