

# IT5001 Software Development Fundamentals

9a. Higher Order Functions

# Functions in Python

- Functions can be
  - ➤ Assigned to variables

> Passed as arguments to functions

> Returned from functions

# "Callability"

Normal variables are NOT <u>callable</u>

```
>>> x = 1
>>> x()
Traceback (most recent call last):
   File "<pyshell#3>", line 1, in <module>
        x()
TypeError: 'int' object is not callable
```

• A function is callable

# Assignments

Normal variables can store values

>>> 
$$x = 1$$
  
>>>  $y = x$   
>>>  $x = 2$ 

Can a variable store a function?!

Can!!!!!!

# Assignments

- The function f is stored in the variable x
   ➤ So x is a function, same as f

#### See the difference

```
>>> def f2():
                            return 999
           With '()'
                                 Without '()'
                      >>> y = f2
>>> x = f2()
>>> print(x)
                      >>> print(y)
999 ← values
                     ><function f2 at 0x0000007ACE8C5A60>
                      >>> type(y)
>>> type(x)
                      <class 'function'>
<class 'int'>
              types
```

# Assigning to a variable

```
def inc_func(x):
    return x+1

my_func = inc_func

print(id(my_func) == id(inc_func))
print(f'ID of inc_func is: {id(inc_func)}')
print(f'ID of my_func is: {id(my_func)}')
print(f'inc_func(1) returns {inc_func(1)} ')
print(f'my_func(1) returns {my_func(1)} ')
```

#### **Output**:

```
True
ID of inc_func is: 1608860195432
ID of my_func is: 1608860195432
inc_func(1) returns 2
my_func(1) returns 2
```

#### Functions can be stored in variables

```
>>> from math import cos, sin, tan
>>> f 1 = cos
                                                   Equivalent
>>> f 1(0) ←
                                                   to cos(0)
1.0
>>> print(f 1)
                                                   The type is
<built-in function cos>
                                                   "function"
>>> def f():
        print("Hello")
>>> print(f)
<function f at 0x000000F9F93F4950>
```

### Functions as elements in Lists/Tuples

```
from math import cos, sin, tan
def inc func(x):
  return x+1
my list = [cos, sin, tan, inc func, print]
x = my list[3](1)
print(x)
```

#### **Output:**

```
from math import cos, sin, tan, pi
my func list = [cos, sin, tan]
theta = pi/3
output = [func(theta) for func in my func list]
print(output)
```

#### **Output:**

#### Functions as elements in Dictionaries

```
def square(n):
    return n**2

def power(n,k):
    return n**k

my_func_dict = {square: None, power: 5}

output = []
for func, parameter in my_func_dict.items():
    output.append(func(2) if parameter == None else func(2,parameter))

print(output)
```

# Functions as input arguments

2.8284271247461903

2.8284271247461903

```
Function calling
      from math import sqrt
                                                                   other function
      def distance 1(x,y):
          return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
      def square(x):
          return x**2
                                                                     Nested function
      def distance 2(x,y):
          def square(x):
               return x**2
          return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
      def distance 3(x,y,square):
          return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
      x = (0, 0)
      y = (2, 2)
                                            Function as input argument
      print(distance 1(x,y))
      print(distance 2(x,y))
      print(distance 3(x,y,square))
           2.8284271247461903
Output:
```

#### Functions that return functions

- Functions can return inner functions as output
- Inner functions serves many purposes
  - ➤ Closures
  - ➤ Decorators

#### Closures

- Closure:
  - > Returns inner functions
  - > Function plus the environment (state) in which they execute together
  - Preserve function state across function calls

#### • Example:

```
>>> def generate_increment(inc):
    def increment(num):
        return num+inc
    return increment

>>> increment_by_2 = generate_increment(2)
>>> increment_by_2(10)

12
>>> increment_by_10 = generate_increment(10)
>>> increment_by_10(23)

33
>>> increment_by_2(23)
```

#### Closures

Create Functions to Power a Number

```
def make power func(n):
    return lambda x:x**n
square = make power func(2)
cube = make power func(3)
square root = make power func(0.5)
>>> print(square(3))
9
>>> print(cube(2))
8
>>> print(square root(16))
4.0
```

#### **Decorators**

- Decorator is a closure
  - > Additionally, outer function accepts a function as input argument
- Modify input function's behaviour with an inner function without explicitly changing input function's code

#### Example

```
def deco(func):
    def wrapper():
        #statements
        func()
        pass
    return wrapper

def f():
    pass

f = deco(f)
```

```
def deco(func):
    def wrapper():
        #statements
        func()
        pass
    return wrapper

@deco
def f():
    pass
```

# **Function Composition**

• In math, we can do something like log(sin(x))

# Mix and Match

```
>>> def add1to(x):
                               A function
         return x + 1
                                                 A variable
                                                 (can be a
>>> def square(x):
                                                 function
         return x *
                                                 too!)
>>> def do 3 times (f, n):
         return f(f(f(n)))
>>> do 3 times(add1to,2)
5
                                             Equivalent to
>>> do 3 times(square,2)
256
                      >>> def do 3 times(f,n):
```

add1to(add1to(add1to(2)))

# Examples

#### The "Powerful" Lambda

```
>>> def aFunctionAddN(n):
>>> def add1(x):
                                 return lambda x: x + n
        return x+1
                             >>> f1 = aFunctionAddN(10)
>>> add1(9)
                             >>> f1(1)
10
                             11
>>> func = lambda x: x +1
                             >>> f1(2)
>>> func(9)
                             12
10
                             >>> f2 = aFunctionAddN(99)
                             >>> f2(1)
                             100
                             >>> f2(f1(3))
                             112
```

We know that, given a function f, the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x+dx) - f(x)}{dx}$$

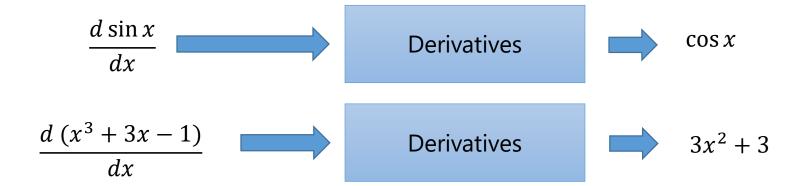
• We know that, given a function f, the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

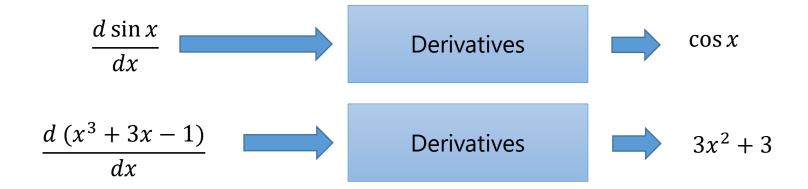
• 
$$\frac{d \sin x}{dx} = \cos x$$

• We know that, given a function f, the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$



- Its input is a function
  - > And output another function



We know that, given a function f, the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x+dx) - f(x)}{dx}$$

Take in a function, returning another function

We know that, given a function f, the derivative of f is

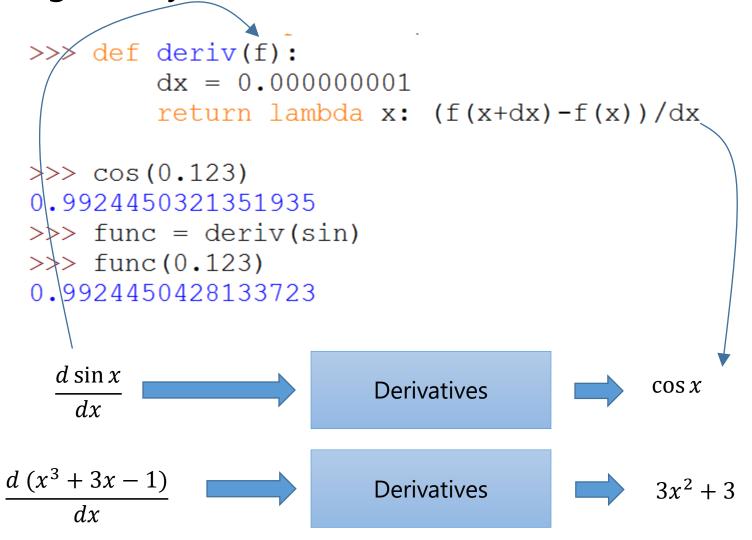
$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x+dx) - f(x)}{dx}$$

```
>>> def deriv(f):
        dx = 0.000000001
        return lambda x: (f(x+dx)-f(x))/dx
                                              Take in a function,
>>> \cos(0.123)
                                              returning another
0.9924450321351935
                                              function
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
>>> def f(x):
        return x**3+3*x-1
>>> deriv(f)(9)
246.00001324870388
>>> x = 9
>>> 3*x**2 +3
```

246



# Application Example of deriv()

- To compute root of function g(x), i.e. find x such that g(x) = 0
- 1. Anyhow assume the answer x = something
- 2. If  $g(x) \approx 0$  then stop: answer is x, return x
- 3. Otherwise
  - x = x g(x)/deriv(x)
- 4. Go to step 2

- To compute root of function g(x), i.e. find x such that g(x) = 0
- 1. Anyhow assume the answer x = something
- 2. If  $g(x) \approx 0$  then stop: answer is x, return x
- 3. Otherwise
  - x = x g(x)/deriv(x)
- 4. Go to step 2

```
def newtonM(g):
    x = 999 #doesnt matter
    err = 0.0000000001
    while(abs(g(x))>err):
        x = x - g(x)/deriv(g)(x)
    return x
```

• To compute the root of function g(x), i.e. find x such that g(x) = 0

```
def deriv(f):
    dx = 0.0000000001
    return lambda x: (f(x+dx)-f(x))/dx

def newtonM(g):
    x = 999 #doesnt matter
    err = 0.0000000001
    while(abs(g(x))>err):
        x = x - g(x)/deriv(g)(x)
    return x
```

Example: Square root of a number A
 ➤ It's equivalent to solve the equation: x² - A = 0

```
    Example: Compute log<sub>10</sub> (A)

   > Solve the equation: 10^x - A = 0
      >>> def my own log10(N):
               return newtonM(lambda x: 10**x - N)
      >>> my own log10(100)
      2.00000000000000013
      >>> x = my own log10(234)
      >>> 10 ** x
      234.00000000000892
```

```
>>> def my_own_log10(N):
    return newtonM(lambda x: 10**x - N)
>>> my_own_log10(100)
2.0000000000000013
>>> x = my_own_log10(234)
>>> 10 ** x
234.000000000000892
```

# You can solve any equation!

.... that Newton Method can solve.

#### Lambda functions

```
>>> f = lambda a, b: lambda x: b(b(a))
>>> f('b', lambda a: a * 3)(lambda a: a[:1])
```

#### Lambda functions

```
>>> f = lambda a, b: lambda x: b(b(a))
     Abstraction is right associative
     f = (lambda a, b: (lambda x: b(b(a)))
>>> f('b', lambda a: a * 3)(lambda a: a[:1])
     Application is left associative
    (f ('b', lambda a: a*3)) ((lambda a: a[:1]))
```

#### Lambda functions

```
(f ('b', lambda a: a*3)) ((lambda a: a[:1]))
(lambda x: 'bbbbbbbbbbb') (lambda a: a[:1])
            'bbbbbbbbbb'
```