

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 **callable**

```
>>> 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**

```
>>> def f():
        print("Hello")
```

```
>>> f()
Hello
```

Assignments

- Normal variables can store values

```
>>> x = 1
>>> y = x
>>> x = 2
```

- Can a variable store a function?!

```
>>> def f():
        print("Hello")
```

```
>>> x = f
>>> x()
Hello
```

- Can!!!!!!

Assignments

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

```
>>> def f():  
        print("Hello")
```

```
>>> x = f
```

```
>>> x()
```

```
Hello
```

See the difference

```
>>> def f2():  
        return 999
```

With '()'

```
>>> x = f2()  
>>> print(x)  
999  
>>> type(x)  
<class 'int'>
```

Without '()'

```
>>> y = f2  
>>> print(y)  
<function f2 at 0x0000007ACE8C5A60>  
>>> type(y)  
<class 'function'>
```

values

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
>>> f_1(0)
1.0
>>> print(f_1)
<built-in function cos>
```

Equivalent
to cos(0)

The type is
"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:

2

```
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:


[0.5000000000000001, 0.8660254037844386, 1.7320508075688767]

Functions as elements in Dictionaries

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

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

function as key



```
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

```
from math import sqrt
```

Function calling
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
```

```
def distance_2(x, y):  
    def square(x):  
        return x**2  
    return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
```

Nested function



```
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))
```

Output: 2.8284271247461903
2.8284271247461903
2.8284271247461903

Functions that return functions

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

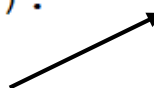
Closures

- 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
```

Access to variable
from outer function



Returns inner function



```
>>> 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)  
25
```

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
 $\log(\sin(x))$

```
>>> def f():  
    print("Hello")
```

```
>>> def do_twice(x):  
    x()  
    x()
```

Equivalent to

```
>>> def do_twice(x):  
    f()  
    f()
```

```
>>> do_twice(f)  
Hello  
Hello
```


Mix and Match

```
>>> def add1to(x):  
    return x + 1
```

```
>>> def square(x):  
    return x * x
```

```
>>> def do_3_times(f, n):  
    return f(f(f(n)))
```

```
>>> do_3_times(add1to, 2)  
5
```

```
>>> do_3_times(square, 2)  
256
```

A function

A variable
(can be a
function
too!)

Equivalent to

```
>>> def do_3_times(f, n):  
    add1to(add1to(add1to(2)))
```

Examples

The “Powerful” Lambda

```
>>> def add1(x):  
    return x+1
```

```
>>> add1(9)
```

```
10
```

```
>>> func = lambda x: x + 1
```

```
>>> func(9)
```

```
10
```

```
>>> def aFunctionAddN(n):  
    return lambda x: x + n
```

```
>>> f1 = aFunctionAddN(10)
```

```
>>> f1(1)
```

```
11
```

```
>>> f1(2)
```

```
12
```

```
>>> f2 = aFunctionAddN(99)
```

```
>>> f2(1)
```

```
100
```

```
>>> f2(f1(3))
```

```
112
```

Agar Agar (Anyhow) Derivative

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

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

- But, if we have very small number dx

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

Agar Agar (Anyhow) Derivative

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

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

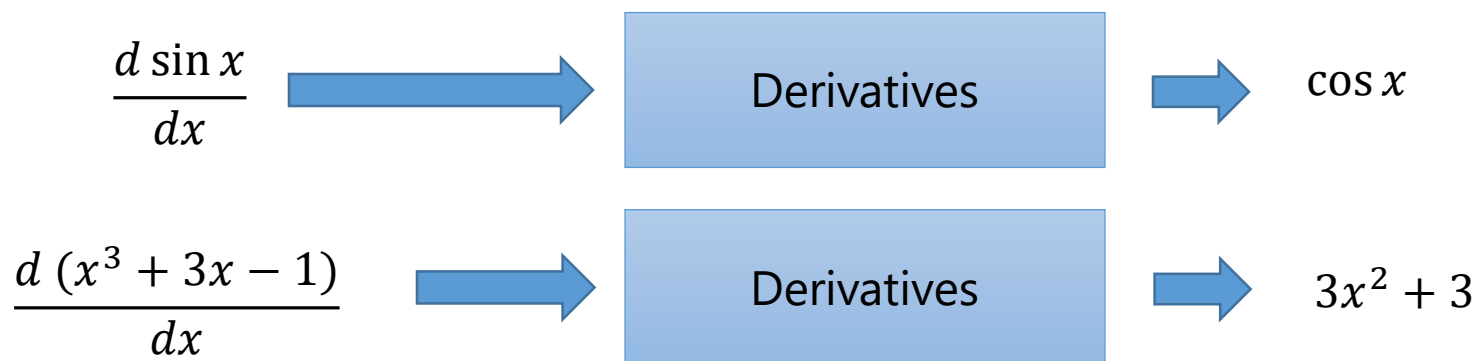
- $\frac{d \sin x}{dx} = \cos x$

- $\frac{d (x^3 + 3x - 1)}{dx} = 3x^2 + 3$

Agar Agar (Anyhow) Derivative

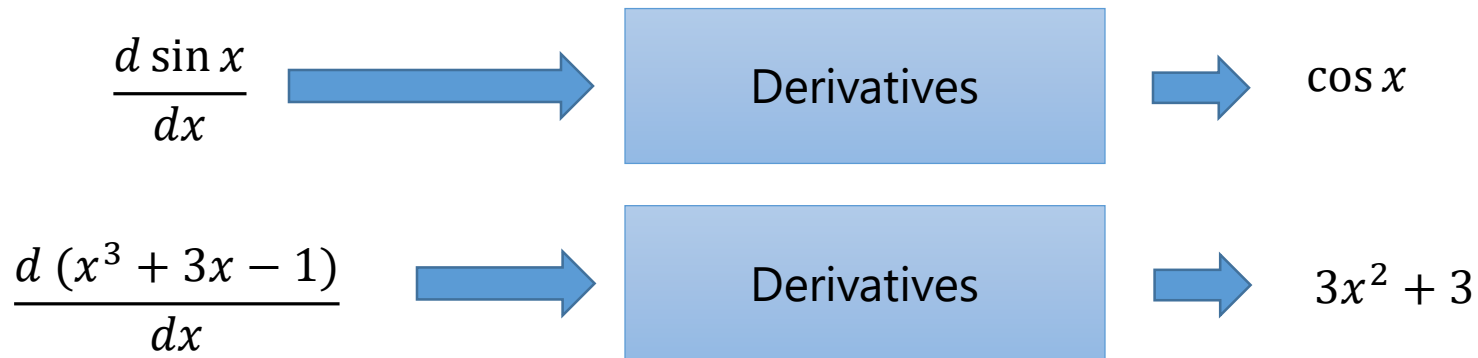
- We know that, given a function f , the derivative of f is

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



Agar Agar (Anyhow) Derivative

- Its input is a function
 - And output another function



Agar Agar (Anyhow) Derivative


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

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

- 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



```
>>> def deriv(f):  
    dx = 0.000000001  
    return lambda x: (f(x+dx) - f(x)) / dx
```

```
>>> cos(0.123)  
0.9924450321351935  
>>> func = deriv(sin)  
>>> func(0.123)  
0.9924450428133723
```


Agar Agar (Anyhow) Derivative

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

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

- But, if we have very small number dx

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

```
>>> def f(x):  
        return x**3+3*x-1
```

```
>>> deriv(f)(9)  
246.00001324870388  
>>> x = 9  
>>> 3*x**2 + 3  
246
```

Agar Agar (Anyhow) Derivative

```
>>> def deriv(f):  
    dx = 0.000000001  
    return lambda x: (f(x+dx) - f(x)) / dx
```

```
>>> cos(0.123)  
0.9924450321351935  
>>> 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
```

Take in a function,
returning another
function

Agar Agar (Anyhow) Derivative

```
>>> def deriv(f):  
    dx = 0.000000001  
    return lambda x: (f(x+dx) - f(x)) / dx  
  
>>> cos(0.123)  
0.9924450321351935  
>>> func = deriv(sin)  
>>> func(0.123)  
0.9924450428133723
```

$$\frac{d \sin x}{dx}$$



Derivatives



$$\cos x$$

$$\frac{d(x^3 + 3x - 1)}{dx}$$



Derivatives



$$3x^2 + 3$$

Application Example of `deriv()`

Example: Newton's method

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

Example: Newton's method

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

```
def newtonM(g):  
    x = 999 #doesn't matter  
    err = 0.00000000001  
    while (abs(g(x)) > err):  
        x = x - g(x)/deriv(g)(x)  
    return x
```

Example: Newton's method

- 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 #doesn't matter  
    err = 0.00000000001  
    while (abs(g(x)) > err):  
        x = x - g(x) / deriv(g)(x)  
    return x
```

Example: Newton's method

- Example: Square root of a number A
 - It's equivalent to solve the equation: $x^2 - A = 0$

```
>>> def my_own_sqrt(A):  
        return newtonM(lambda x:x*x-A)
```

```
>>> x = my_own_sqrt(10)  
>>> x * x  
9.9999999999999998
```


Example: Newton's method

- Example: Compute $\log_{10}(A)$
 - 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  
234.000000000000892
```

```
>>> 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  
234.000000000000892
```

You can solve any equation!

.... that Newton Method can solve.