Functions with Default Arguments and Anonymous Functions

Python
Marquette University

Functions with Default Arguments

- We have created functions that have *positional* arguments
    - Example:

```
def fun(foo, bar):
    print(2*foo+bar)

fun(2, 3)
```

- When we invoke this function, the first argument (2) gets plugged into variable foo and the second argument (3) get plugged into variable bar

Keyword (Named) Arguments

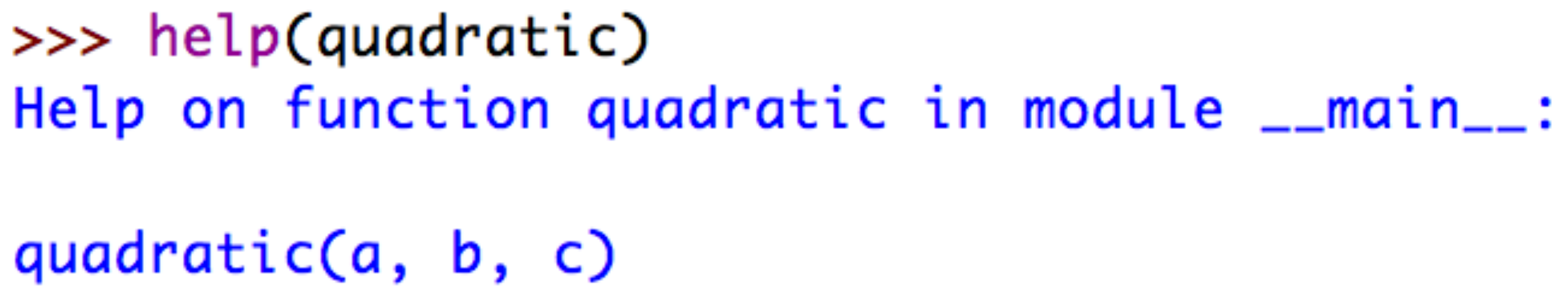
- We can also use the names of the variables in the function definition.
- Example: (we soon learn how to deal better with errors)

```
def quadratic(a, b, c):
    if b**2-4*a*c >= 0:
        return -b/(2*a) + math.sqrt(b**2-4*a*c)/(2*a)
    else:
        print("Error: no solution")


print(quadratic(1, -4, 4))  #CALL BY POSITION
print(quadratic(c=4, a=1, b=-4)  #CALL BY KEYWORD
```

Keyword (Named) Arguments

- Keyword arguments have advantages
  - If you have a function with many positional arguments, then you need to carefully match them up
  - At least, you can use the help function in order to figure out what each argument does, if you named them well in the function definition



```
>>> help(quadratic)
Help on function quadratic in module __main__:

quadratic(a, b, c)
```

Keyword (Named) Arguments

- You can force the user of a function to use keywords by introducing an asterisk into the definition of the function:
  - All arguments after the asterisk need to be passed by keyword
  - The arguments before the asterisk can be positional

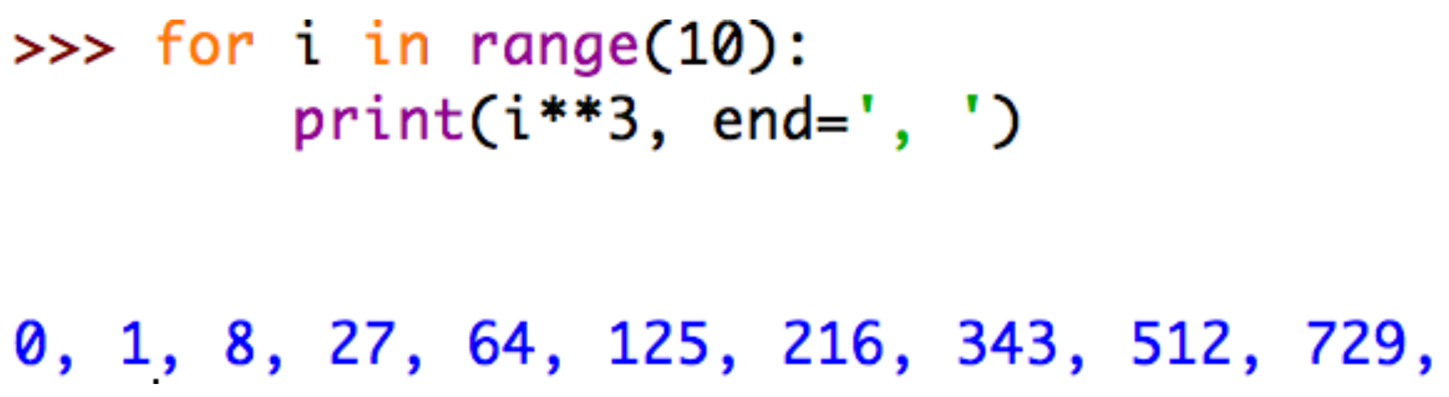
```
def function ( posarg1, * , keywarg1 ):
```

```
def fun(a, b, *, c):
    …

print(fun(2, 3, c=5)
```

Default arguments

- You have already interacted with built-in functions that use default arguments
    - Print:
        - end: How the string is terminated (default is new-line character)
        - sep: What comes between different outputs (default is space)
        - file: Location of output (default is "standard output")

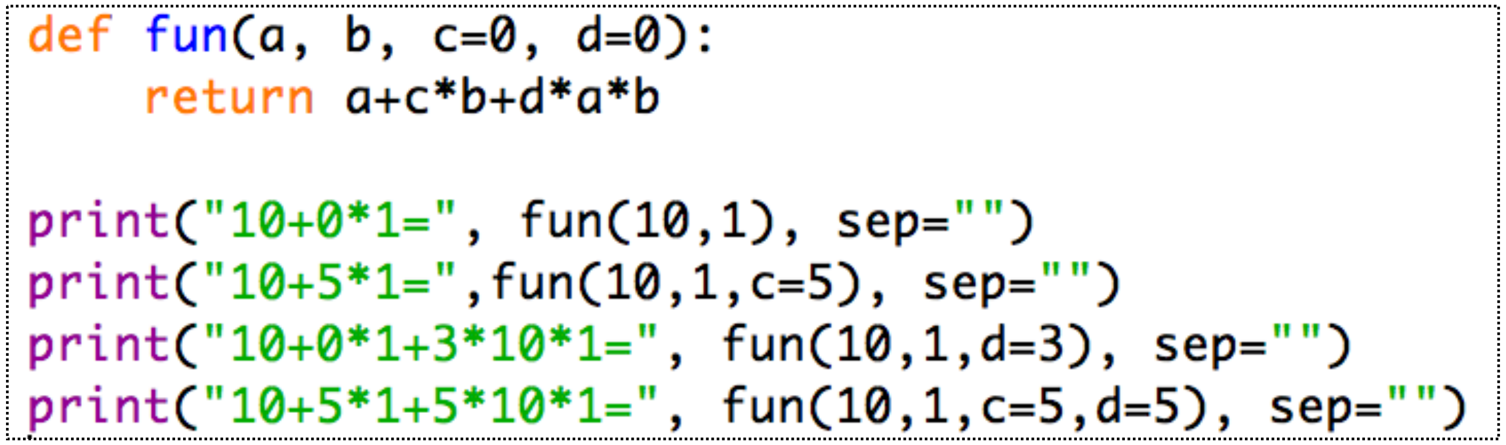


```
>>> for i in range(10):
        print(i**3, end=', ')


0, 1, 8, 27, 64, 125, 216, 343, 512, 729,
```

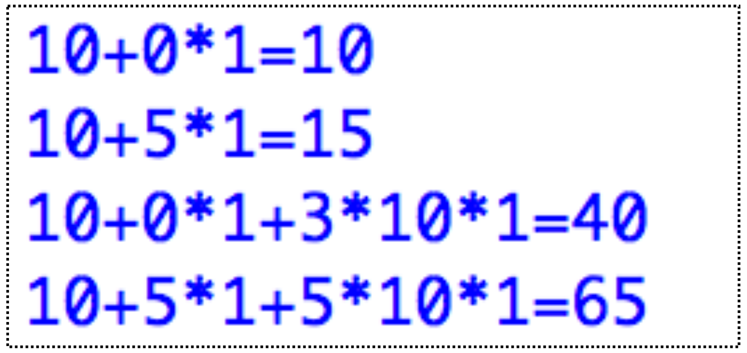
Default arguments

- Defining default arguments is easy
  - Just use the arguments with default arguments last and assign default values in the function definition



```python
def fun(a, b, c=0, d=0):
    return a+c*b+d*a*b

print("10+0*1=", fun(10,1), sep="")
print("10+5*1=",fun(10,1,c=5), sep="")
print("10+0*1+3*10*1=", fun(10,1,d=3), sep="")
print("10+5*1+5*10*1=", fun(10,1,c=5,d=5), sep="")
```



```
10+0*1=10
10+5*1=15
10+0*1+3*10*1=40
10+5*1+5*10*1=65
```

Default Arguments

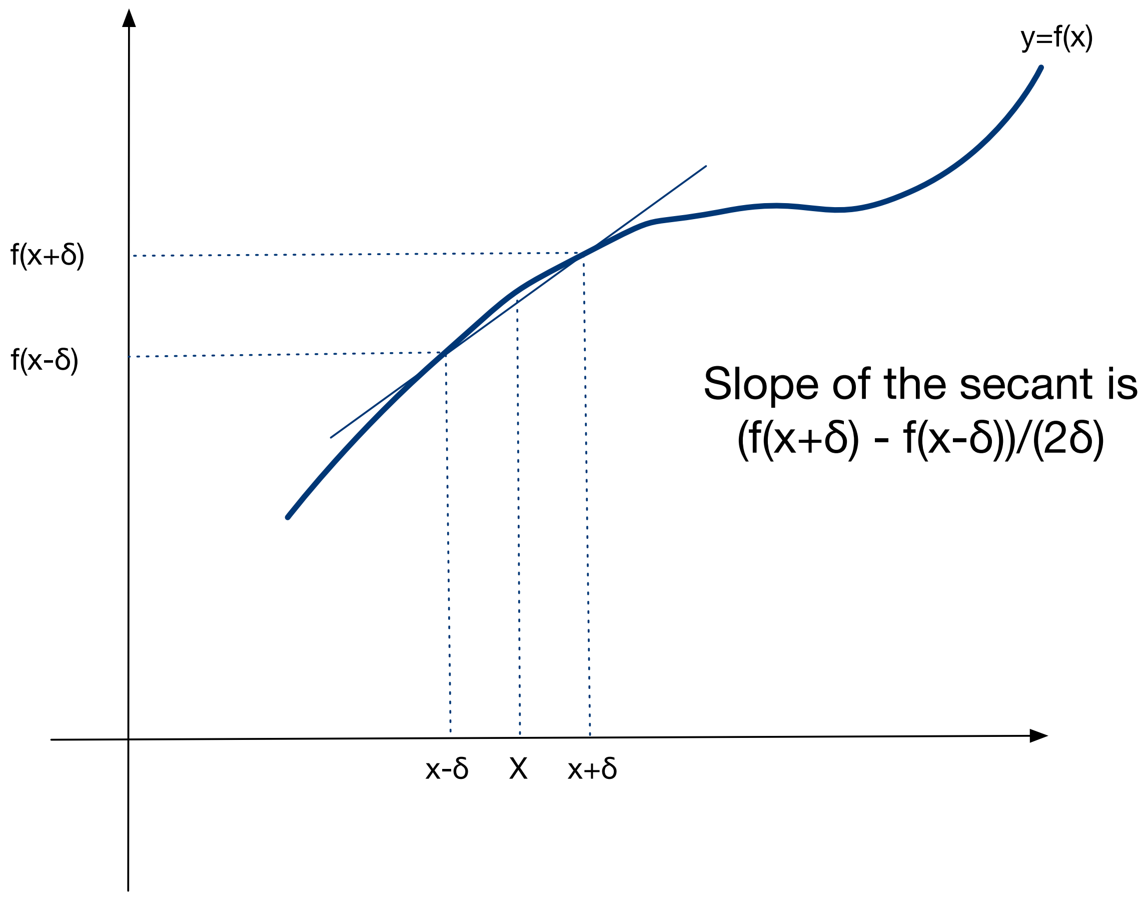
- How to write readable code:
  - Named arguments and default arguments with well-chosen names make code more readable
  - Most effort in software engineering goes towards maintaining code

Anonymous Functions

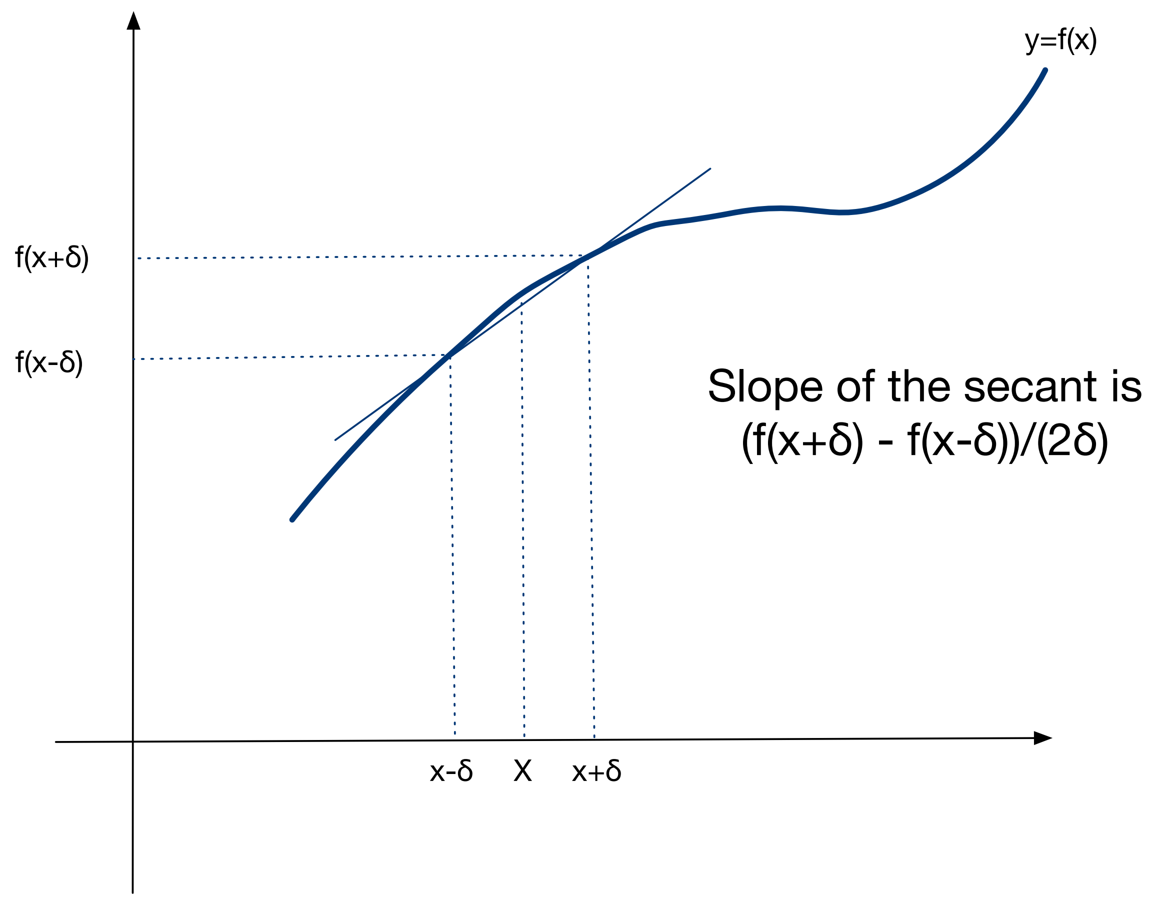
- Up till now, we used the def-construct in order to define functions
- Sometimes it is necessary to pass functions to another function, but not necessary to define the argument for future uses

Anonymous Function

- Example:
  - Numerical Differentiation
    - Derivative of a function *f* at a point is the slope of the tangent
    - Approximated by a secant



Anonymous Functions



- The slope of the secant is the difference of values over the difference of arguments:

$$\frac{f(x+\delta)-f(x-\delta)}{x+\delta-(x-\delta)}=\frac{f(x+\delta)-f(x-\delta)}{2\delta}$$

- If δ is small, then this is a good approximation of the derivative

Anonymous Functions

- A simple method for derivation uses a fixed, but small value for δ.

```
def derivative(function, x):
    delta = 0.000001
    return (function(x+delta)-function(x-delta))/(2*delta)
```

- To test this, we try it out with sine, whose derivative is cosine

```
for i in range(20):
    x = i/20
    print(x, math.cos(x), derivative(math.sin, x))
```

Anonymous Functions

- It turns out that the numerical derivative is quite close in this test

```
0.0 1.0 0.9999999999998334
0.05 0.9987502603949663 0.9987502603940601
0.1 0.9950041652780257 0.9950041652759256
0.15 0.9887710779360422 0.9887710779310499
0.2 0.9800665778412416 0.9800665778519901
0.25 0.9689124217106447 0.9689124216977207
0.3 0.955336489125606 0.9553364891112803
0.35 0.9393727128473789 0.9393727128381713
0.4 0.9210609940028851 0.9210609939747094
0.45 0.9004471023526769 0.9004471023255078
0.5 0.8775825618903728 0.8775825618978494
0.55 0.8525245220595057 0.8525245220880606
0.6 0.8253356149096783 0.8253356149623414
0.65 0.7960837985490559 0.7960837985487856
0.7 0.7648421872844885 0.7648421873063249
0.75 0.7316888688738209 0.7316888688824186
0.8 0.6967067093471655 0.6967067094354462
0.85 0.6599831458849822 0.6599831459119798
0.9 0.6216099682706645 0.6216099682765375
0.95 0.5816830894638836 0.5816830894733727
```

Anonymous Functions

- Notice that in the test, we specified math.sin and not math.sin(x),
- The former is a function (which we want)
- The latter is a value (which we do not want)

```
for i in range(20):
    x = i/20
    print(x, math.cos(x), derivative(math.sin, x))
```

Anonymous Functions

- To specify a function argument, I can use a **lambda-expression**
  - Lambda-expressions were used in Mathematical Logic to investigate the potential of formal calculations

```
lambda  x :  5*x*x-4*x+3
```

  - Lambda expression consists of a keyword lambda
    - followed by one or more variables
    - followed by a colon
    - followed by an expression for the function
  - This example implements the function $x \rightarrow 5x^2 - 4x + 3$

Anonymous Functions

- To test our numerical differentiation function, we pass it the function $x \rightarrow x^2$, which has derivative $2x$

```
for i in range(20):
    x = i/20
    print("{:5.3f} {:5.3f} {:5.3f}”.format(
      x,
      derivative(lambda x: x*x, x),
      2*x))
```

Anonymous Functions

- Since we are rounding to only three digits after the decimal point, we get perfect results

```
0.000 0.000 0.000
0.050 0.100 0.100
0.100 0.200 0.200
0.150 0.300 0.300
0.200 0.400 0.400
0.250 0.500 0.500
0.300 0.600 0.600
0.350 0.700 0.700
0.400 0.800 0.800
0.450 0.900 0.900
0.500 1.000 1.000
0.550 1.100 1.100
0.600 1.200 1.200
0.650 1.300 1.300
0.700 1.400 1.400
0.750 1.500 1.500
0.800 1.600 1.600
0.850 1.700 1.700
0.900 1.800 1.800
0.950 1.900 1.900
```

Anonymous Functions

- I can even use lambda expressions as an alternative way of defining functions:

```
norm = lambda x, y: math.sqrt(x*x+y*y)
```

- Since there are two variables, norm is a function of two arguments:

```
print(norm(2.3, 1.7))
```