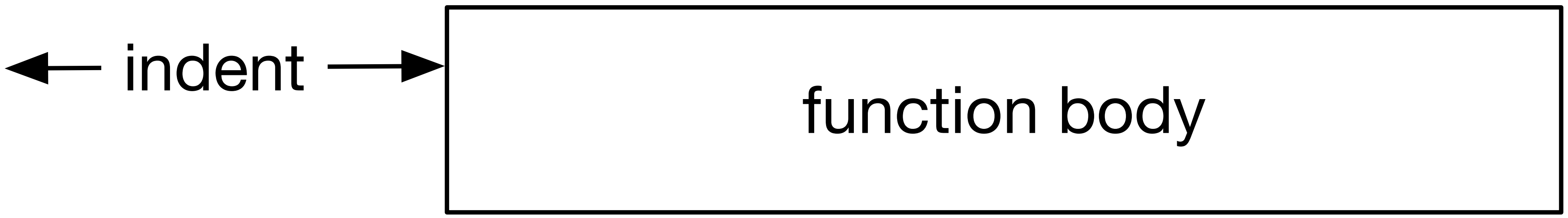
Functions in Python

Python Functions

- Functions defined by keyword def
- Can return value with keyword return

```
def  function_name ( list of arguments) :
```



Python Functions

- Example: A Heron square-root function

```
def square(S, precision):
    """ This is a square root function that uses Heron's Method
    """
    x = 1
    while not -precision < S-x**2 < precision:
        x = 0.5*(x+S/x)
    return x
```

Python Functions

- Without return:
  - Function returns when code is exhausted
- Peculiarities:
  - Neither argument nor return types are specified

Python Functions

- This is weird, but legal

```
def example(x):
    if x == 1:
        return 1
    if x == 2:
        return "two"
    if x == 3:
        return 3.0
```

- Returns a None value for x = 4
- Returns int for x=1, string for x=2, float for x=3

Simple Examples

- A “fruitless” function that does not return a value:
    - Printing the first n square numbers

```
def first_squares(n):
    for i in range(1, n+1):
        print(i**2)
```

Simple Examples

- A function that returns a power of 2

```
def power_of_two(x):
    return 2**x
```

Simple Example

- A function that calculates the nth harmonic number

```
def har(n):
    accu = 0
    for i in range(1, n+1):
        accu += 1/i
    return accu
```

Simple Example

- The harmonic numbers are known to diverge:
  - They go to infinity
    - Though numeric inaccuracies make it hard to show this
  - Create a function that for $x \in \mathbb{R}_{>0}$ returns
    - $\min(\{n : H_n > x\})$

Simple Examples

- To determine the smallest number such that H_n is larger than x:
    - Use a while loop
        - Keep adding until harmonic number is larger
            - The while condition is that the accumulator is smaller than x

Simple Example

- while-solution

```
def min_har(x):
    accu = 0
    i = 0
    while accu < x:
        i+= 1
        accu += 1/i
    return i
```

Functions of Functions

- Functions are full-fledged objects in Python
  - This means you can pass functions as parameters
  - Example: Calculate the average of the values of a function at -n, -n+1, -n+2, …, -2, -1, 0, 1, 2, … , n-2, n-1, n
    - The function needs to be a function of one integer variable
  - Example:
    - *n* = 2, function is squaring
    - Return value is $((-2)^2 + (-1)^2 + 0^2 + 1^2 + 2^2)/5 = 2$

Functions of Functions

- We first define the averaging function with two arguments
  - The number *n*
  - The function over which we average, called `func`

```
def averaging(n, func):
```

Functions of Functions

- Inside the function, we create an accumulator and a loop index, running from *-n* to *n.*

```
def averaging(n, func):
    accu = 0
    for i in range(-n, n+1):
```

Functions of Functions

- Inside the loop, we modify the accumulator `accu` by adding the value of the function at the loop variable.

```
def averaging(n, func):
    accu = 0
    for i in range(-n, n+1):
        accu += func(i)
```

Functions of Functions

- There are $2n+1$ points at which we evaluate the function.
- We then return the average as the accumulator over the number of points

```
def averaging(n, func):
    accu = 0
    for i in range(-n, n+1):
        accu += func(i)
    return accu/(2*n+1)
```

Functions of Functions

- In order to try this out, we need to use a function
- We can just define one in order to try out our averaging function

```
def square(number):
   return number*number

def averaging(n, func):
    accu = 0
    for i in range(-n, n+1):
        accu += func(i)
return accu/(2*n+1)

print(averaging(2, square))
```

Local Functions

- Can have a function definition inside a function
    - Not many use cases

```
def factorial(number):
    if not isinstance(number, int):
        raise TypeError("sorry", number, "must be an integer")
    if not number >= 0:
        raise ValueError("sorry", number, "must be positive")

    def inner_factorial(number):
        if number <= 1:
            return 1
        return number * inner_factorial(number-1)

    return inner_factorial(number)
```

Local and Global Variables

- A Python function is an independent part of a program
    - It has its own set of variables
        - Called local variables
    - It can also access variables of the environment in which the function is called.
        - These are global variables
    - The space where variables live is called their scope
    - We will revisit this issue in the future

Examples

```
a=3
b=2
def foo(x):
    return a+x
def bar(x):
    b=1
    return b+x

print(foo(3), bar(3))
```

- a and b are two global variables
- In function foo:
  - *a* is global, its value remains 3
- In function bar:
  - *b* is local, since it is redefined to be 1

The global keyword

- In the previous example, we generated a local variable *b* by just assigning a value to it.
- There are now two variables with name *b*
- In bar, the global variable is hidden
- If we want to assign to the global variable, then we can use the keyword global to make *b* refer to the global variable. An assignment then does not create a new local variable, but rather changes the value of the old one.

Example

- In foo:
  - A local variable *b*
  - A global variable *a*
  - The value of *a* changes by executing *foo*( )

```
a = 1
b = 2

def foo():
    global a
    a = 2
    b = 3
    print("In foo:" , "a=", a, " b=", b)

print("Outside foo: " ,"a=", a, " b=", b)
foo()
print("Outside foo: " ,"a=", a, " b=", b)

##Outside foo:  a= 1  b= 2
##In foo: a= 2  b= 3
##Outside foo:  a= 2  b= 2
```

Scoping

- Global scope:
  - Names that we define are visible to all our code
- Local scope:
  - Names that we define are only visible to the current function

Scoping

- LEGB — rule to resolve names
  - Local
  - Enclosed (e.g. enclosing function)
  - Global
  - Built-in

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

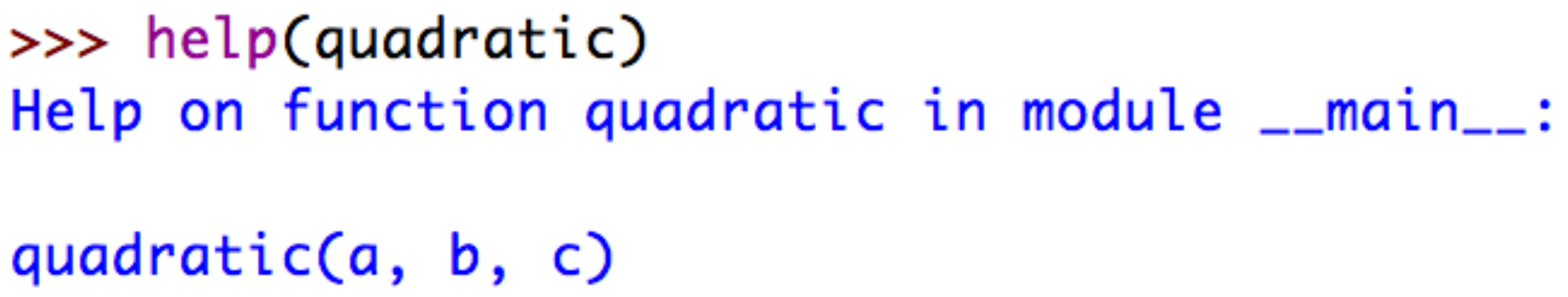
- We can also use the names of the variables in the function definition.
- Example: (we soon learn how to deal better with errors)

```
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 ):
```

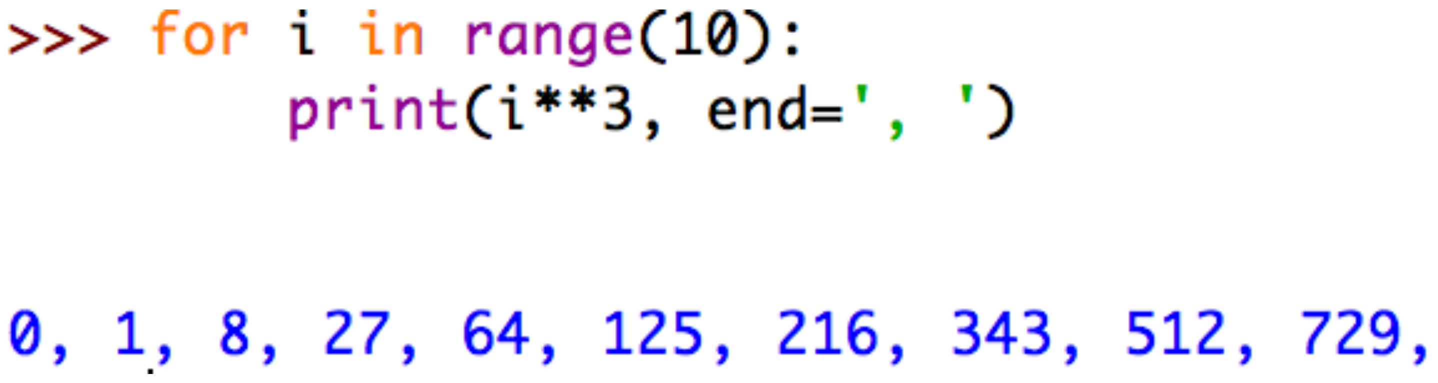
```
def fun(a, b, *, c):
    …

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

Pythonic Tip

- If you want to write understandable code:
  - **Use keyword arguments**
  - Use reasonable default values
  - Write a documentation
    - Three quotation marks followed by remarks followed by three quotation marks

Default arguments

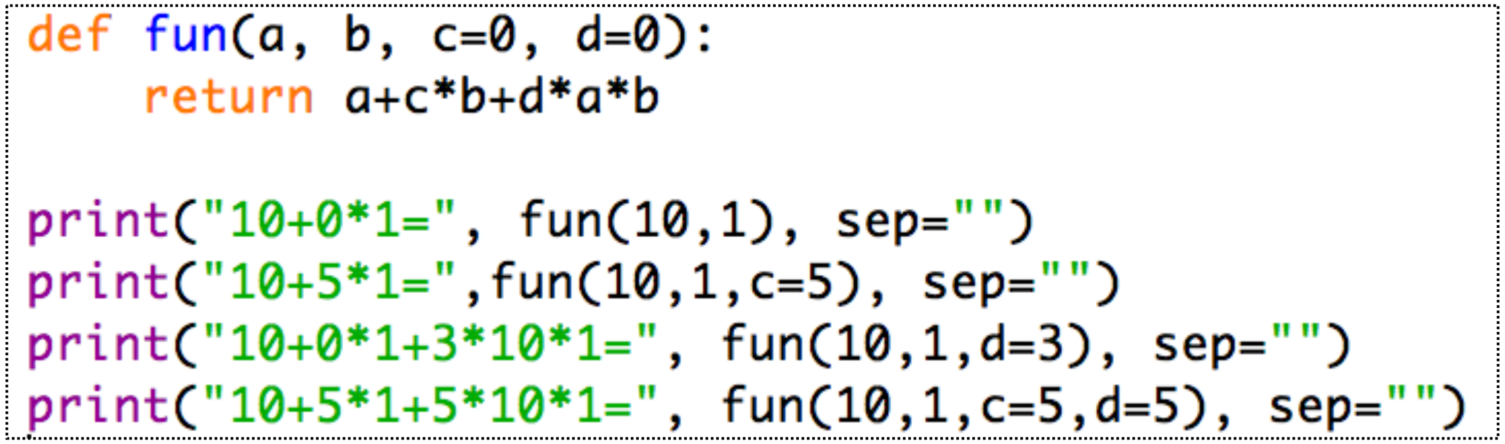
- You have already interacted with built-in functions that use default arguments
    - Print:
        - end: How the string is terminated (default is new-line character)
        - sep: What comes between different outputs (default is space)
        - file: Location of output (default is "standard output")



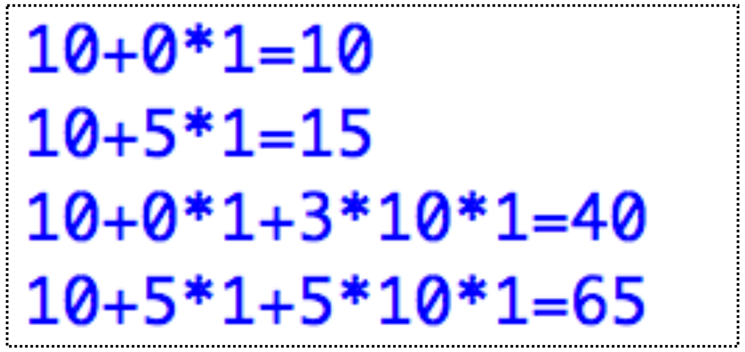
```
>>> for i in range(10):
        print(i**3, end=', ')


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



```
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

Example

- A slightly better square root

```
def square(S, precision=10**-9):
    """ This is a square root function that uses Heron's Method
    """
    x = 1
    while not -precision < S-x**2 < precision:
        x = 0.5*(x+S/x)
    return x
```

```
>>> square(3)
1.7320508075688772
>>>
```

Example: Numerical Differentiation

- Numerical Differentiation Formula
  - $$\left.\frac{df}{dx}\right|_{x=x_0} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x-\delta)}{2\delta}$$
- Write a function with a small default value for δ

Example: Numerical Differentiation

- We want a function that takes a function as an argument and returns a function
    - This is easy in Python
        - Inside the function, define a new function and return it

Example: Numerical Differentiation

- First things first:
  - How do we define the derivative function?

Example: Numerical Differentiation

- We find f and delta in the scope

```
def fprime(x):
        return (f(x+delta) - f(x-delta))/(2*delta)
```

- Now we have to define the “exterior” function

Example: Numerical Differentiation

- Here is the exterior function

```
def diff(f, delta):
    def fprime(x):
        return (f(x+delta) - f(x-delta))/(2*delta)
    return fprime
```

Example: Numerical Differentiation

- Final step:
  - Use a default value for delta

```
def diff(f, delta=10**-7):
    def fprime(x):
        return (f(x+delta) - f(x-delta))/(2*delta)
    return fprime
```

Preview: Numerical Differentiation

- This simple differentiation method is actually quite good
    - Let’s draw some graphs, as we will learn in the second half
    - We import two modules that need to be installed with pip (or pip3 if you also have python2)

Preview: Numerical Differentiation

- Importing modules

```
import matplotlib.pyplot as plt
import numpy as np
```

- Creating an array of X-values

```
X=np.linspace(0, 2*np.pi, 1001)
```

- And three different versions of the derivative of sine

```
df = diff(np.sin, delta=0.1)
df2 = diff(np.sin, delta=0.01)
df3 = diff(np.sin, delta=0.001)
```

Preview: Numerical Differentiation

- Then we plot the graphs
  - Need an array of X-values and an array of Y-values
    - Labels are added for a legend

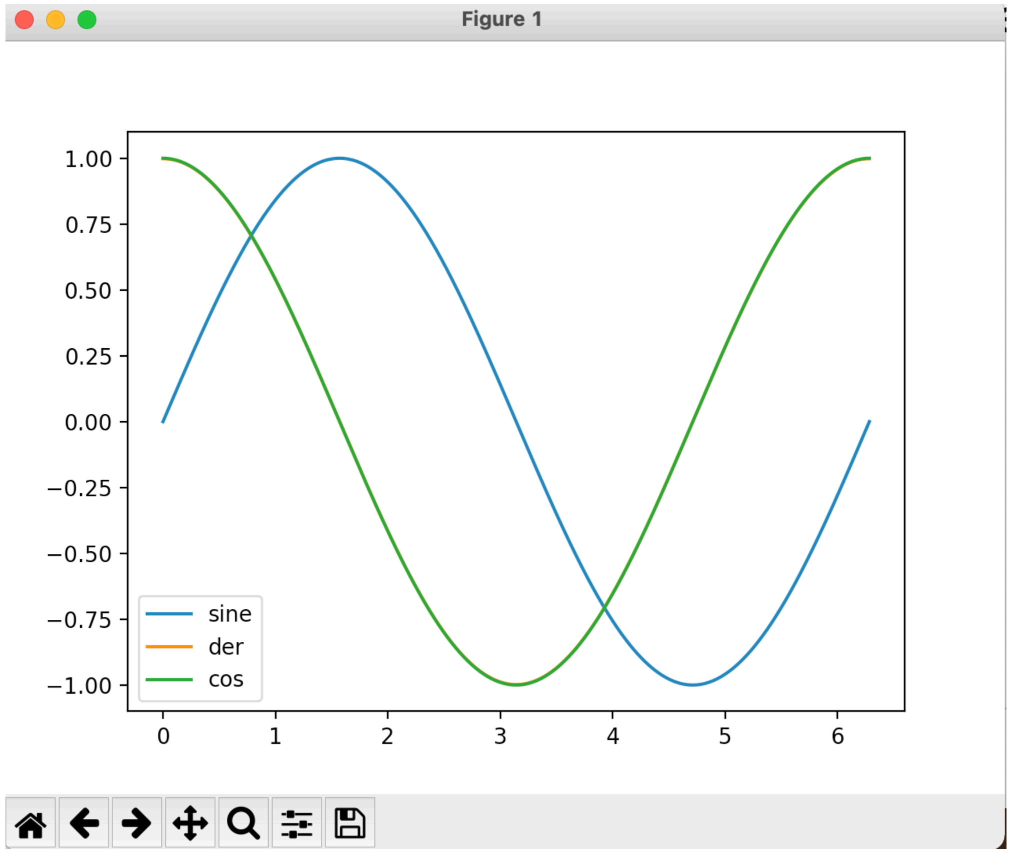
```
plt.plot(X, np.sin(X), label='sine')
plt.plot(X, df(X), label='der')
plt.plot(X, np.cos(X), label='cos')
```

    - Then we create the legend and call show

```
plt.legend()
plt.show()
```

Preview: Numerical Differentiation

- When we run this, we get



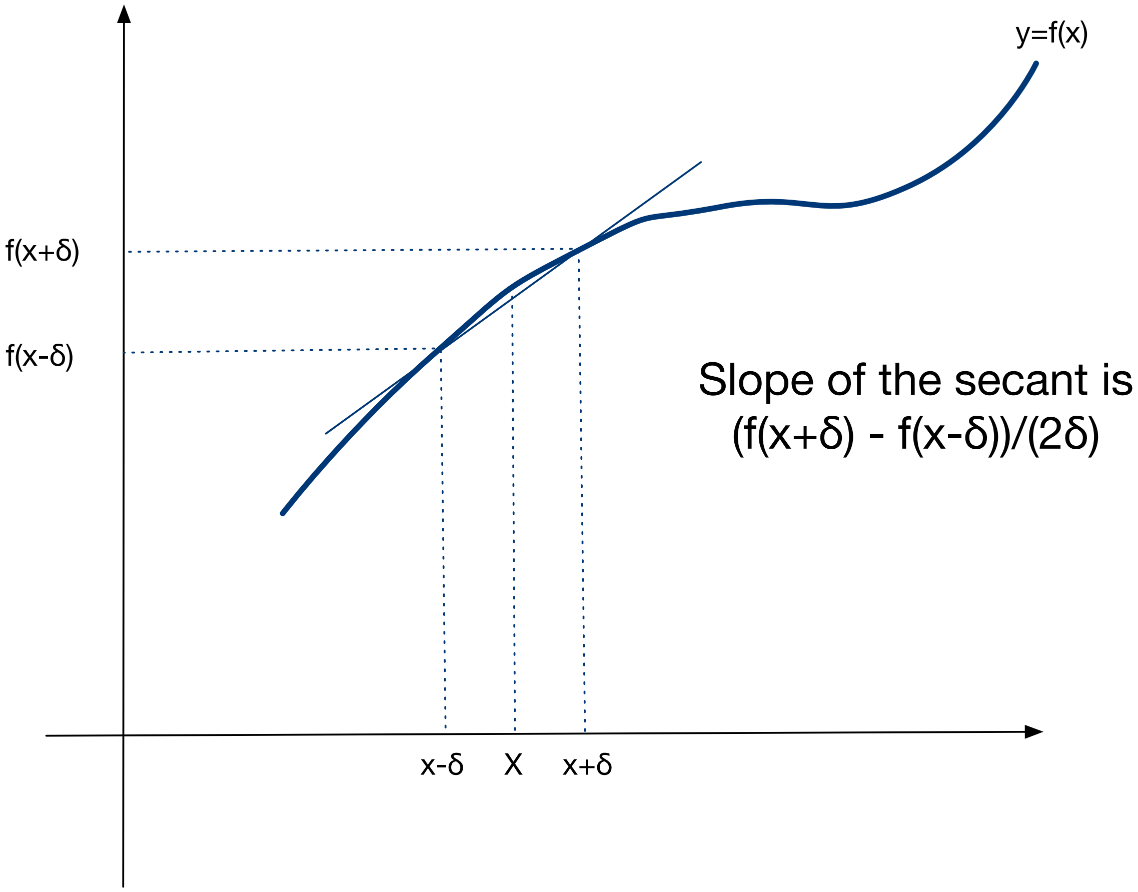
- Result is so good that we cannot see the difference between the numerical derivative and the cosine function

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

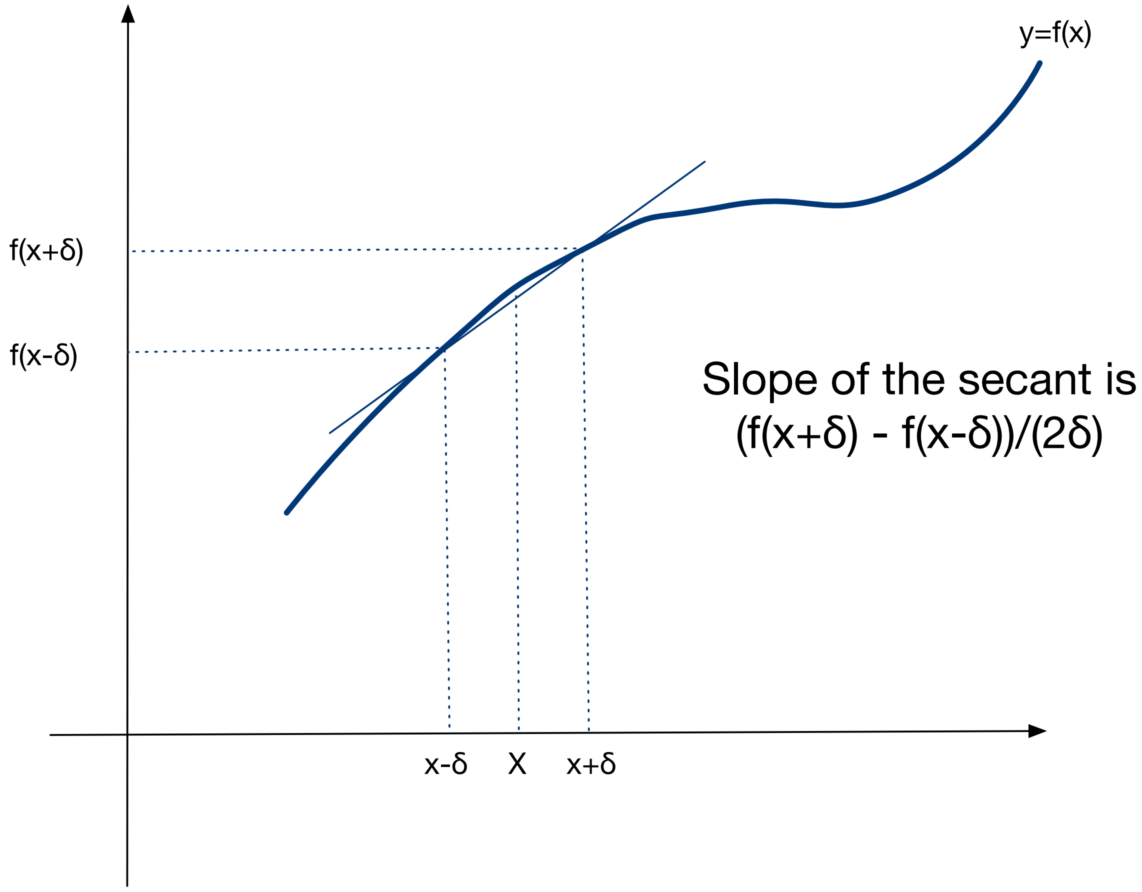
Anonymous Function

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



Slope of the secant is
(f(x+δ) - f(x-δ))/(2δ)

Anonymous Functions



Slope of the secant is (f(x+δ) - f(x-δ))/(2δ)

- 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 $x \to 5x^2 - 4x + 3$

Anonymous Functions

- To test our numerical differentiation function, we pass it the function $x \to x^2$, 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))
```

Annotations

- Completely optional way to make function definitions easier to read
  - Uses swift language convention
    - for arguments: name colon type
      - where type is either a Python type or a string
    - for return value: use ->

```
def quadratic(a: 'number', b: 'number', c: 'number') -> float :
    disc = (b**2-4*a*c)**0.5
    return (-b+disc)/(2*a)
```

Decorators

- Functions are also return values
  - One way to use this are decorators (for the future)
    - A decorator is put on top of a function
    - The decorator then takes the function and replaces it with another function

Decorators

- This is an example of a function factory!

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_namaste():
    print("Namaste!")

say_when = my_decorator(say_whee)
```

- We can automatically apply the decorator

```
@my_decorator
def say_namaste():
    print("Namaste!")
```

Decorators

- Some decorators are provided in modules
  - lru_cache in functools
    - stores the result of functions in an lru cache

Future topics on functions

- Memoization
- Decorators

In Class Exercises

- From the easy to the hard

1. Write a function of no arguments that prints out an empty line, followed by "Namaste" or "Viva la revolución" or "Laudate Dominum"*, followed by an empty line

* According to your preference

Solution

- Python 3 allows the use of unicode characters

```
def one():
    print()
    print('Namaste')
    print()
```

```
def one():
    print()
    print('नमस्ते ')
    print()
```

```
def one():
    print()
    print('¡Viva la revolución!
    print()
```

```
def one():
    print()
    print('LAUDATEDOMINUM ')
    print()
```

In Class Exercises

- Write a function that calculates the median of three numbers
    - E.g. median(2,3,4) return 3, median(1,0,10) returns 1

In Class Exercises

```
def median(a, b, c):
    if a<=b<=c or c<=b<=a:
        return b
    elif a<=c<=b or b<=c<=a:
        return c
    else:
        return a
```

In Class Exercises

- Define a function $x \to x^2 + \dfrac{1}{x^2 + 1}$ using a lambda expression

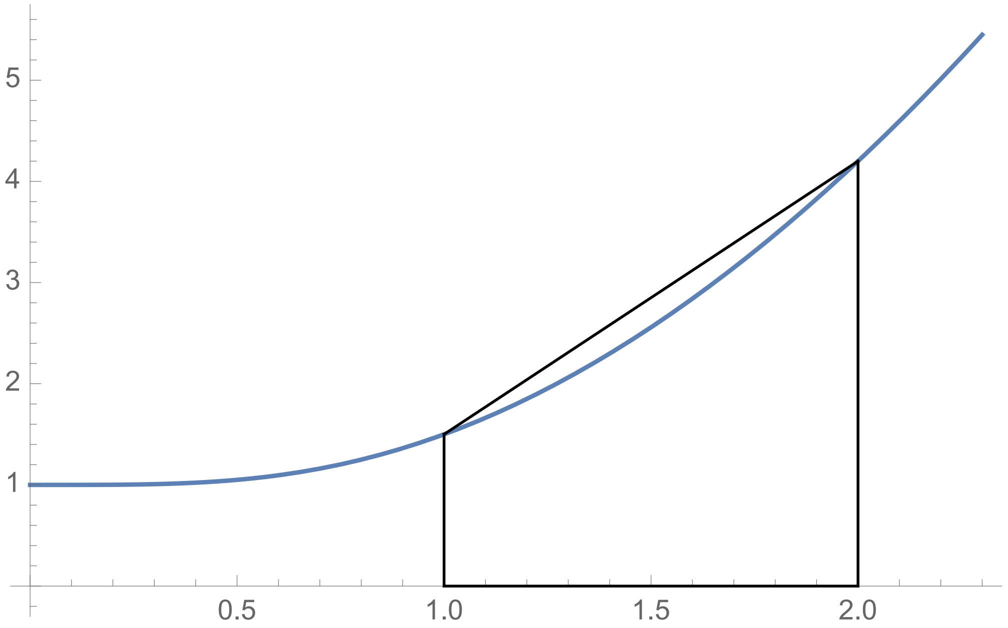
In Class Exercises

```
func = lambda x: x**2+1/(1+x**2)
```

In Class Exercises

- Write a function of a function f and arguments a and b that calculates $\frac{1}{2}(f(a)+f(b))(b-a)$

- This is the trapezoid formula that approximates $\int_a^b f(x)dx$



In Class Exercises

```
def trap(f, a, b):
    return 0.5*(f(a)+f(b))*(b-a)
```

```
>>> trap(lambda x: x**2+1/(1+x**2), 1, 2)
2.85
```

In Class Exercises

- Write a function that prints out a checker board of *n* by *n* fields

```
>>> checker(4)
***   ***
***   ***
   ***   ***
   ***   ***
***   ***
***   ***
   ***   ***
   ***   ***
```

```
>>> checker(8)
***   ***   ***   ***
***   ***   ***   ***
   ***   ***   ***   ***
   ***   ***   ***   ***
***   ***   ***   ***
***   ***   ***   ***
   ***   ***   ***   ***
   ***   ***   ***   ***
***   ***   ***   ***
***   ***   ***   ***
   ***   ***   ***   ***
   ***   ***   ***   ***
***   ***   ***   ***
***   ***   ***   ***
   ***   ***   ***   ***
   ***   ***   ***   ***
```

In Class Exercises

```
def checker(n):
    fields = n//2
    for _ in range(fields):
        for i in range(2):
            print(fields*(3*'*'+3*' '))
        for i in range(2):
            print(fields*(3*' '+3*'*'))
```

Strings in Python

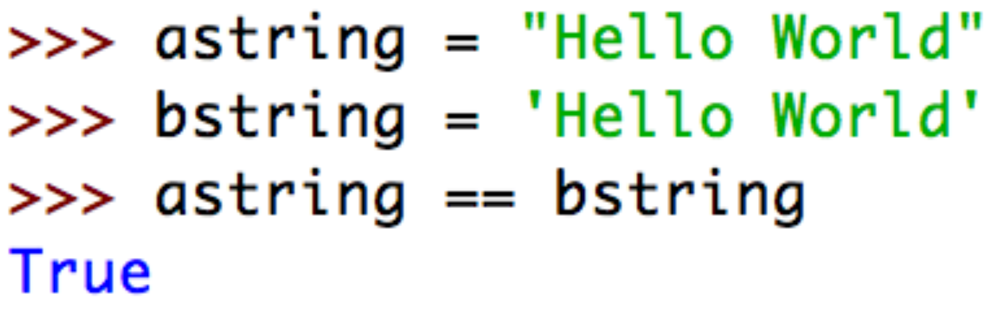
Thomas Schwarz, SJ

Strings

- Basic data type in Python
  - Strings are immutable, meaning they cannot be changed
    - Why?
      - It’s complicated, but string literals are very frequent. If strings cannot be changed, then multiple occurrences of the same string in a program can be placed in a single memory location.
      - More importantly, strings can serve keys in key-value pairs.

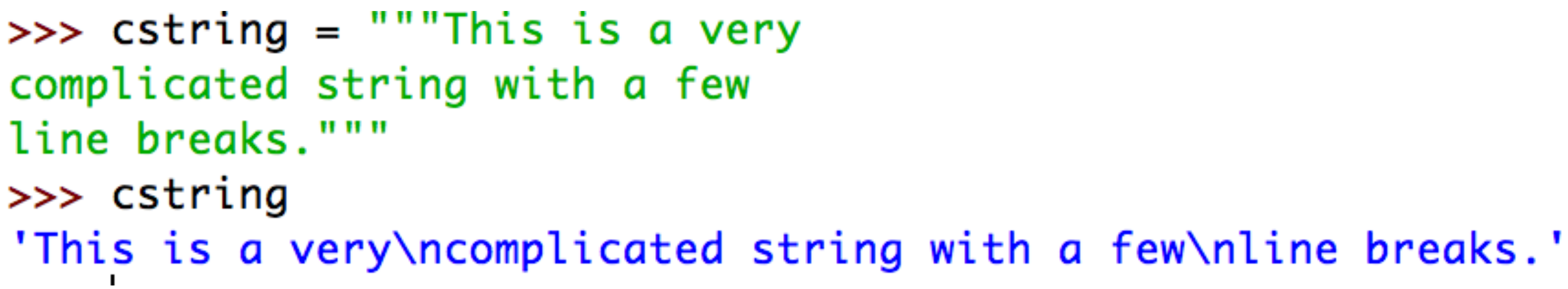
String Literals

- String literals are defined by using quotation marks
  - Example:



```
>>> astring = "Hello World"
>>> bstring = 'Hello World'
>>> astring == bstring
True
```

- To create strings that span newlines, use the triple quotation mark



```
>>> cstring = """This is a very
complicated string with a few
line breaks."""
>>> cstring
'This is a very\ncomplicated string with a few\nline breaks.'
```

Escapes

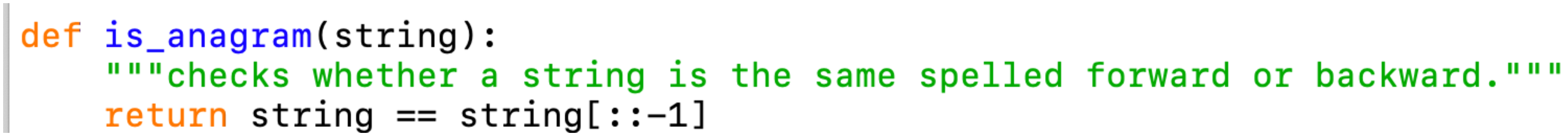
- Python is very good at detecting your intentions when processing string literals
  - E.g.: `"It's mine"`
  - Still sometimes need to use the escape character
    - \t, \n, \", \', \\, \r
    - \xhh —> character with hex value 0xhh
  - Python 3 uses machine conventions for endings
- Python 3 uses utf-8 natively
  - `greetings = ("शुभ प्रभात", "સુપ્રભાત", "शुभ प्रभात")`

Docstrings

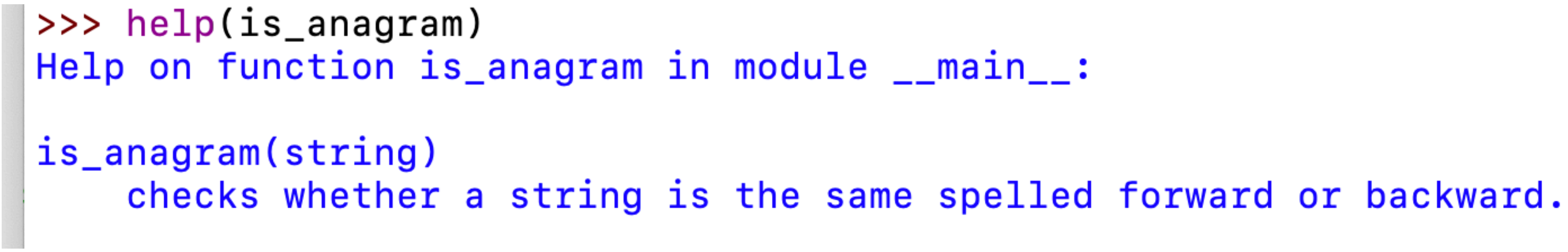
- Doc strings
  - String literals that appear as the first line of a module, function, class, method definition
  - All these items should have a docstring
  - The docstring replaces the help string in Idle and IPython/Jupyter
  - Indent them under the indentation of the object they describe

Docstrings

- Always use triple quotation marks
    - Even for one-liners



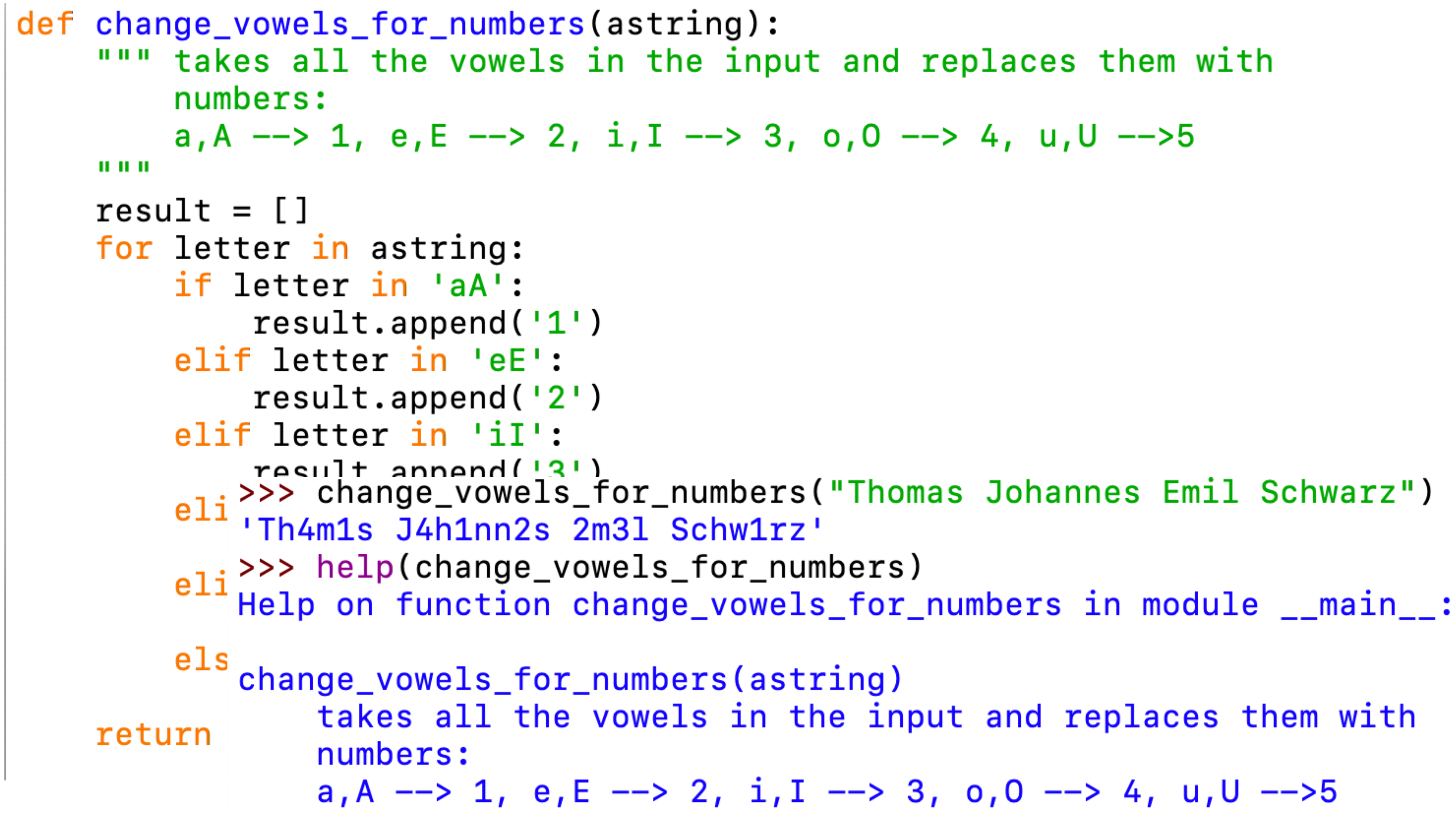
```python
def is_anagram(string):
    """checks whether a string is the same spelled forward or backward."""
    return string == string[::-1]
```



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

is_anagram(string)
    checks whether a string is the same spelled forward or backward.
```

Docstrings

- Example



```
def change_vowels_for_numbers(astring):
    """ takes all the vowels in the input and replaces them with
        numbers:
        a,A --> 1, e,E --> 2, i,I --> 3, o,O --> 4, u,U -->5
    """
    result = []
    for letter in astring:
        if letter in 'aA':
            result.append('1')
        elif letter in 'eE':
            result.append('2')
        elif letter in 'iI':
            result.append('3')
        eli
        eli
        els
    return
```

```
>>> change_vowels_for_numbers("Thomas Johannes Emil Schwarz")
'Th4m1s J4h1nn2s 2m3l Schw1rz'
>>> help(change_vowels_for_numbers)
Help on function change_vowels_for_numbers in module __main__:

change_vowels_for_numbers(astring)
    takes all the vowels in the input and replaces them with
    numbers:
    a,A --> 1, e,E --> 2, i,I --> 3, o,O --> 4, u,U -->5
```

String Methods

- Strings are classes and have many built in methods
- `s.lower(), s.upper()` : returns the lowercase or uppercase version of the string
- `s.strip()`: returns a string with whitespace removed from the start and end
- `s.isalpha() / s.isdigit() / s.isspace()` tests if all the string chars are in the various character classes
- `s.startswith('other'), s.endswith('other')` tests if the string starts or ends with the given other string

String Methods

- There are a number of methods for strings. Most of them are self-explaining
  - `s.find('other')` : searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
  - `s.replace('old', ‘new')`: returns a string where all occurrences of 'old' have been replaced by 'new'
  - `len(s)` returns the length of a string

Strings and Characters

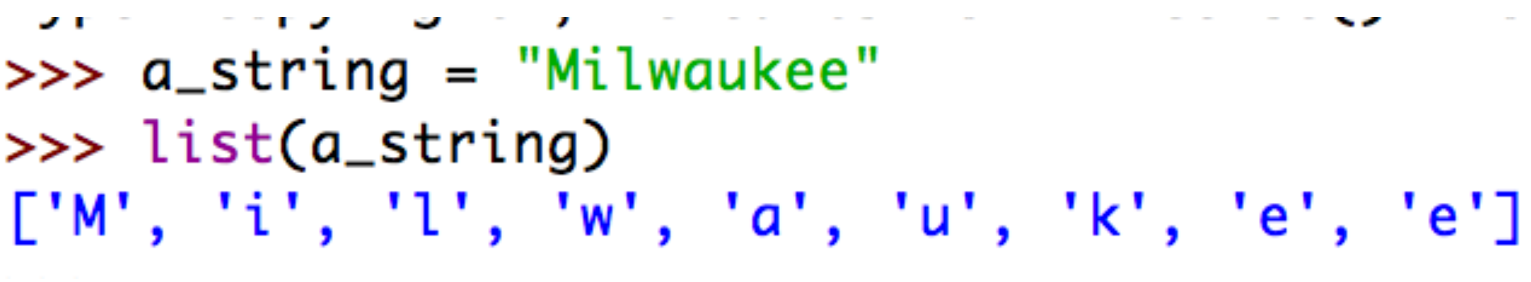
- Python does not have a special type for characters
    - Characters are just strings of length 1.

Accessing Elements of Strings

- We use the bracket notation to gain access to the characters in a string
    - a_string[3] is character number 3, i.e. the fourth character in the string

String Processing

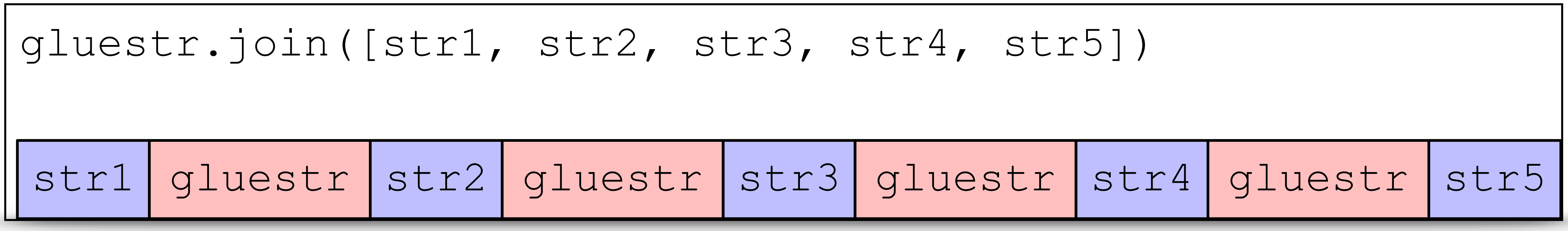
- Since strings are immutable, we process strings by turning them into lists, then processing the list, then making the list into a string.
  - String to list: Just use the list-command



```
>>> a_string = "Milwaukee"
>>> list(a_string)
['M', 'i', 'l', 'w', 'a', 'u', 'k', 'e', 'e']
```

String Processing

- Turn lists into strings with the join-method
  - The join-method has weird syntax
    - `a_string = "".join(a_list)`
      - The method is called on the empty string `""`
      - The sole parameter is a list of characters or strings
    - You can use another string on which to call join
      - This string then becomes the glue

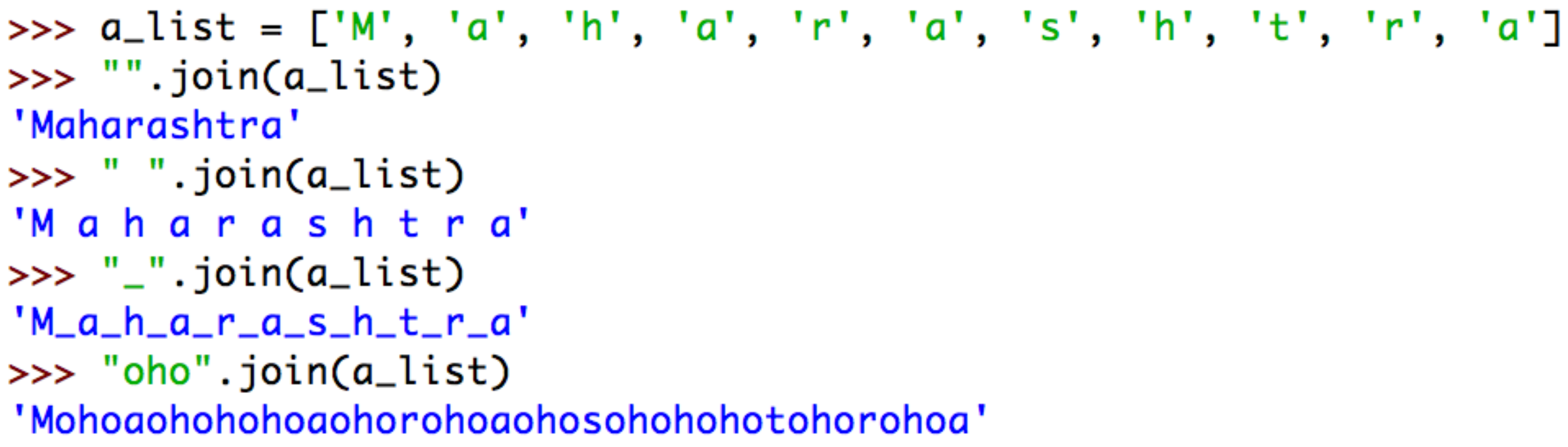


```
gluestr.join([str1, str2, str3, str4, str5])
```

str1	gluestr	str2	gluestr	str3	gluestr	str4	gluestr	str5

String Processing

- Examples



```
>>> a_list = ['M', 'a', 'h', 'a', 'r', 'a', 's', 'h', 't', 'r', 'a']
>>> "".join(a_list)
'Maharashtra'
>>> " ".join(a_list)
'M a h a r a s h t r a'
>>> "_".join(a_list)
'M_a_h_a_r_a_s_h_t_r_a'
>>> "oho".join(a_list)
'Mohoaohohohoaohorohoaohosohohohotohorohoa'
```

String Processing

- Procedure:
  - Take a string and convert to a list
  - Change the list or create a new list
  - Use join to recreate a new string

- Alternative Procedure:
  - Build a string one by one, using concatenation (+ -operator)
  - Creates lots of temporary strings cluttering up memory
    - Which is bad if you are dealing with large strings.