SECTION 6:
USER-DEFINED FUNCTIONS

ENGR 103 – Introduction to Engineering Computing

User-Defined Functions

- By now you're accustomed to using ***Python functions*** in your scripts
- Consider, for example, `np.mean()`
  - Commonly-used function to calculate an average value
  - A Python (NumPy) module – written using other Python functions
  - Need not write code each time an average is calculated
- Functions allow ***reuse of commonly-used blocks of code***
  - Executable from any script or the console
- Can also create ***user-defined functions***
  - Just like built-in or library functions
    - Similar syntax, structure, reusability, etc.

Anatomy of a Function

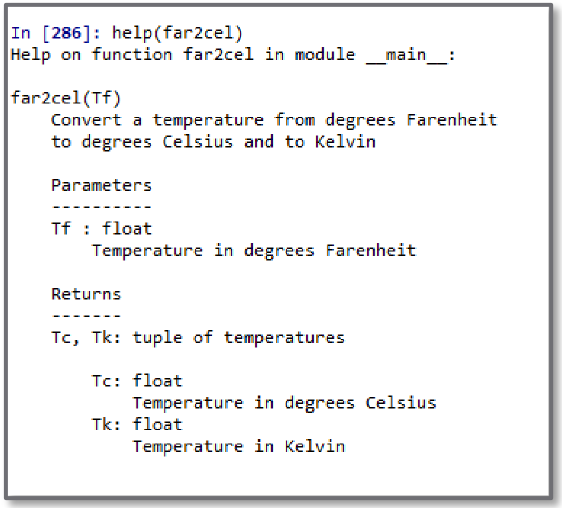
Function m-file must begin with the keyword 'def'

Function Name

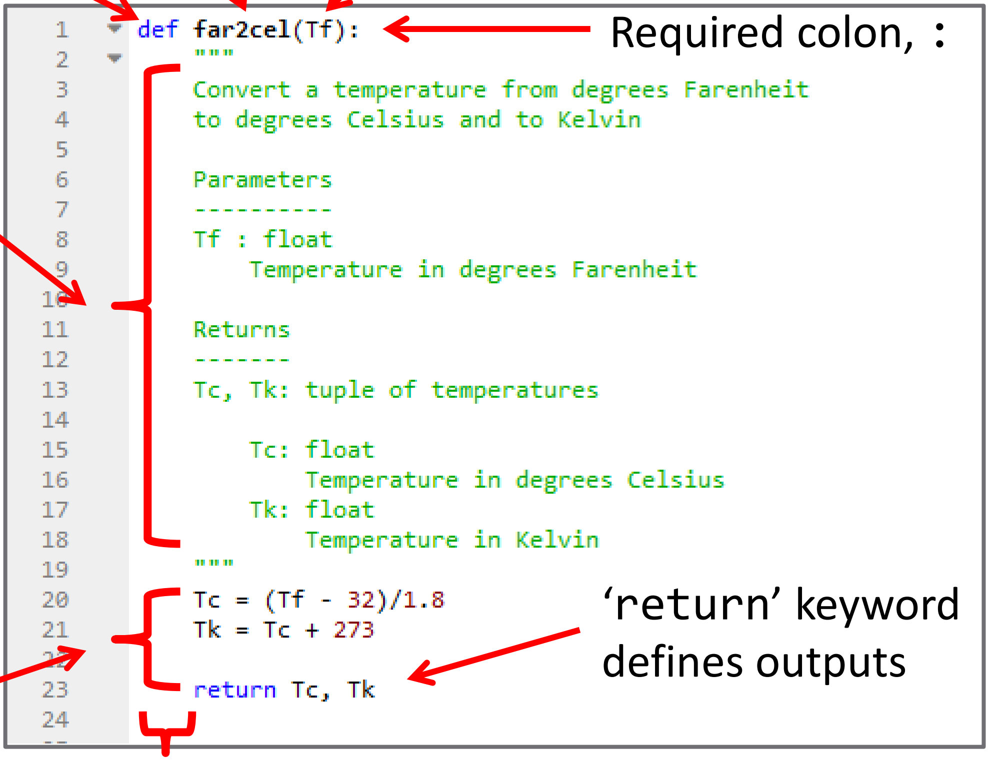
Input Argument(s)

Required colon, :

Doc string – displayed when help is requested in the console:



```
In [286]: help(far2cel)
Help on function far2cel in module __main__:

far2cel(Tf)
    Convert a temperature from degrees Farenheit
    to degrees Celsius and to Kelvin

    Parameters
    ----------
    Tf : float
        Temperature in degrees Farenheit

    Returns
    -------
    Tc, Tk: tuple of temperatures

        Tc: float
            Temperature in degrees Celsius
        Tk: float
            Temperature in Kelvin
```



```
def far2cel(Tf):
    """
    Convert a temperature from degrees Farenheit
    to degrees Celsius and to Kelvin

    Parameters
    ----------
    Tf : float
        Temperature in degrees Farenheit

    Returns
    -------
    Tc, Tk: tuple of temperatures

        Tc: float
            Temperature in degrees Celsius
        Tk: float
            Temperature in Kelvin
    """
    Tc = (Tf - 32)/1.8
    Tk = Tc + 273

    return Tc, Tk
```

'return' keyword defines outputs

Python code that defines the function

Function code defined by whitespace (indents) – no brackets or 'end' statement

User-Defined Functions

- ***Keep your code DRY***
  - "***D***on't ***R***epeat ***Y***ourself"

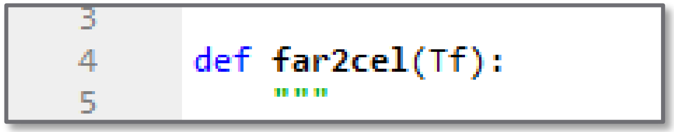
- Do not write the same code more than once
  - ***Create functions*** for frequently-used code blocks
  - Improves conciseness and readability of your code
  - If code needs to be modified, only need to do it once

- Avoid ***WET*** code
  - "***W***rite ***E***verything ***T***wice"
  - "***W***rite ***E***very ***T***ime"
  - "***W***e ***E***njoy ***T***yping"
  - "***W***aste ***E***veryone's ***T***ime"

Function Inputs and Outputs

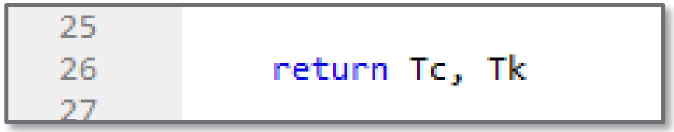
Function Inputs and Outputs

- Just like built-in or library functions, user-defined functions may have ***inputs*** and ***outputs***
  - But, they need not have either
- ***Inputs***
  - Arguments passed into the function
  - Specified inside the parentheses in the function definition



```python
def far2cel(Tf):
    """
```

- ***Outputs***
  - Arguments returned from the function
  - Specified with the `return` statement

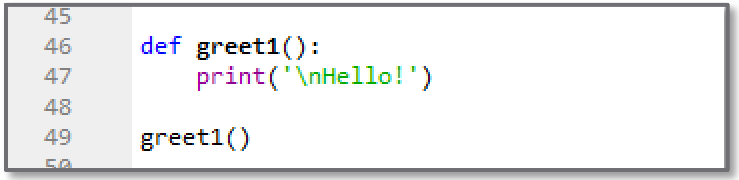


```python
    return Tc, Tk
```

Function Inputs and Outputs

- Functions may or may not have inputs or outputs, e.g.:

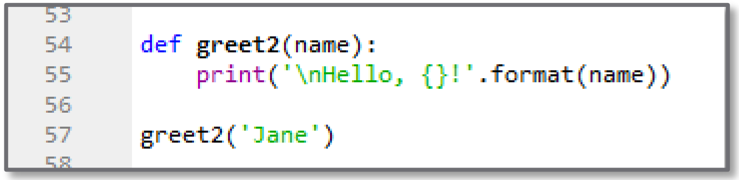
  - No input or output



```python
def greet1():
    print('\nHello!')

greet1()
```

```
Hello!

In [224]:
```

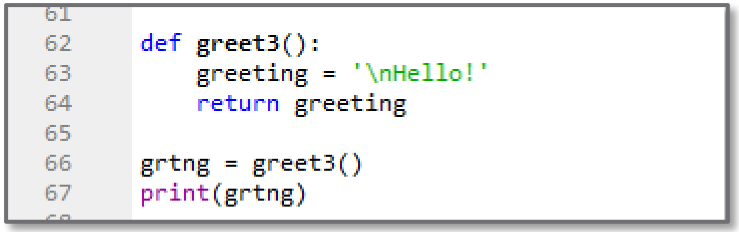
  - Input only



```python
def greet2(name):
    print('\nHello, {}!'.format(name))

greet2('Jane')
```

```
Hello, Jane!

In [225]:
```

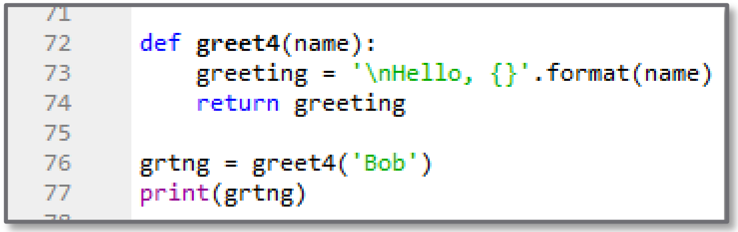
  - Output only



```python
def greet3():
    greeting = '\nHello!'
    return greeting

grtng = greet3()
print(grtng)
```

```
Hello!

In [226]:
```

  - Input and output



```python
def greet4(name):
    greeting = '\nHello, {}'.format(name)
    return greeting

grtng = greet4('Bob')
print(grtng)
```

```
Hello, Bob

In [227]:
```

Positional and Keyword Input Arguments

```
def func(arg1, arg2, …, kwarg1=def1, kwarg2=def2, …)
```

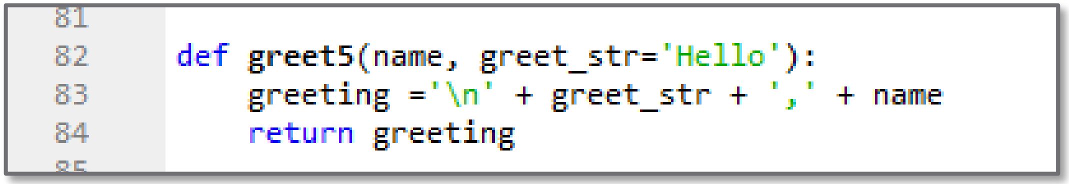
- Two main types of input arguments:
  - ***Positional arguments*** (`arg1, …`)
    - *Required* inputs passed in the specified order
    - *Position* determines what is arg1, arg2, and so on
  - ***Keyword arguments*** (`kwarg1=def1, …`)
    - Passed as ***keyword=value pairs***
    - Order does not matter
    - Useful for specifying default values for optional inputs
      - If kwarg1, above, is not passed, it defaults to `def1`
- For example:

```
plt.plot(x, y, linewidth=2)
```

  - `x` and `y` are ***positional*** arguments
  - `linewidth` is a ***keyword*** argument

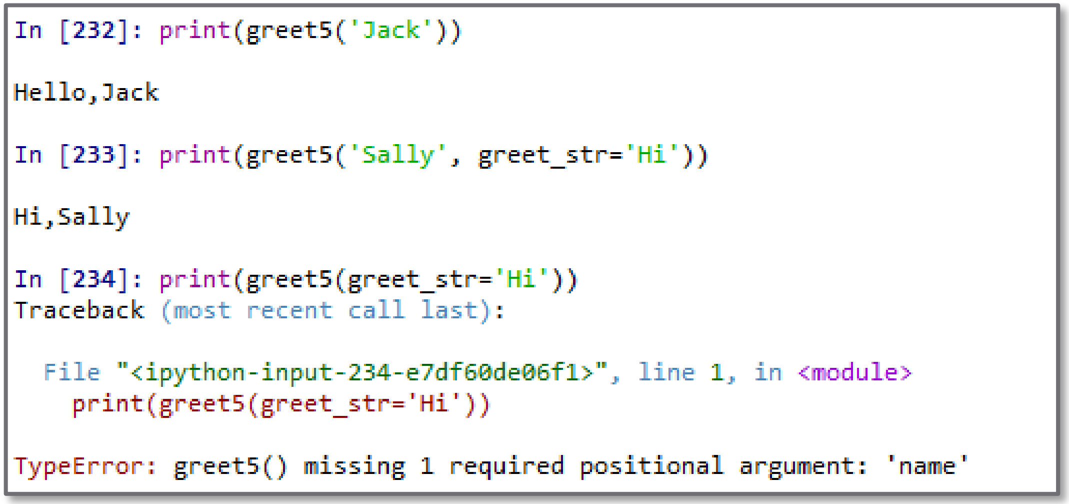
Positional and Keyword Input Arguments

- Consider a function with one positional argument and one keyword argument



```python
def greet5(name, greet_str='Hello'):
    greeting ='\n' + greet_str + ',' + name
    return greeting
```

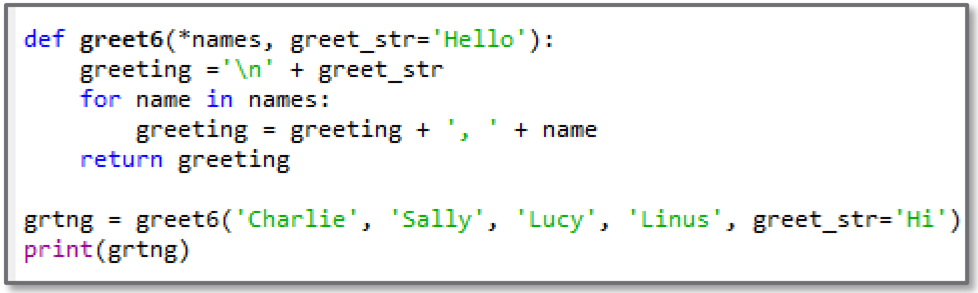
  - `name`: positional argument – required
  - `greet_str`: keyword argument – optional – default: 'Hello'



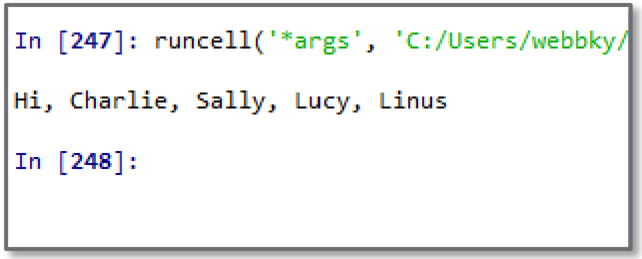
```
In [232]: print(greet5('Jack'))

Hello,Jack

In [233]: print(greet5('Sally', greet_str='Hi'))

Hi,Sally

In [234]: print(greet5(greet_str='Hi'))
Traceback (most recent call last):

  File "<ipython-input-234-e7df60de06f1>", line 1, in <module>
    print(greet5(greet_str='Hi'))

TypeError: greet5() missing 1 required positional argument: 'name'
```

Variable Input Arguments - *args

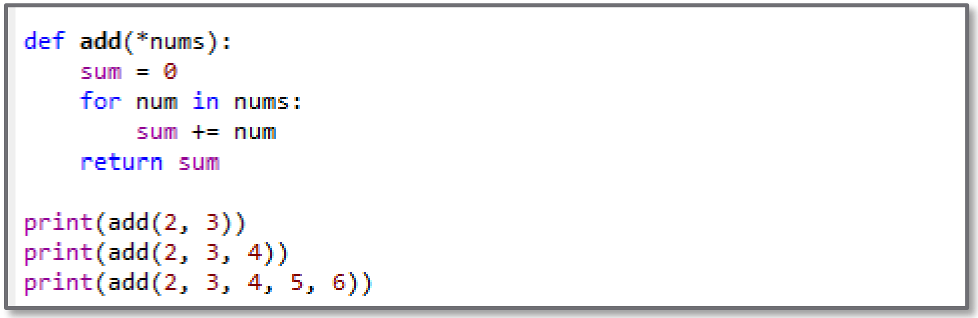
- Some functions allow for a variable number of inputs
  - Use `*args` in the function definition
  - Multiple inputs passed to function as a ***tuple***



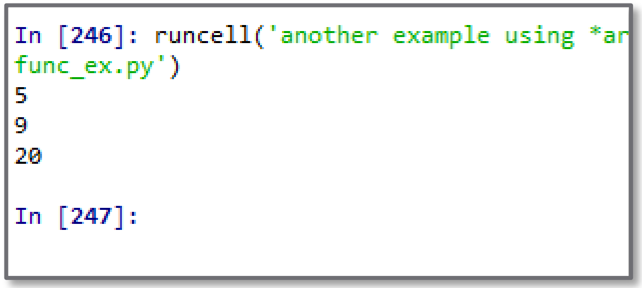
```python
def greet6(*names, greet_str='Hello'):
    greeting ='\n' + greet_str
    for name in names:
        greeting = greeting + ', ' + name
    return greeting

grtng = greet6('Charlie', 'Sally', 'Lucy', 'Linus', greet_str='Hi')
print(grtng)
```



```
In [247]: runcell('*args', 'C:/Users/webbky/

Hi, Charlie, Sally, Lucy, Linus

In [248]:
```



```python
def add(*nums):
    sum = 0
    for num in nums:
        sum += num
    return sum

print(add(2, 3))
print(add(2, 3, 4))
print(add(2, 3, 4, 5, 6))
```



```
In [246]: runcell('another example using *ar
func_ex.py')
5
9
20

In [247]:
```

Multiple Outputs

- Functions may return multiple outputs
  - Returned as a ***tuple*** or ***list***
- To return a tuple:

```
return Tc, Tk
```

or

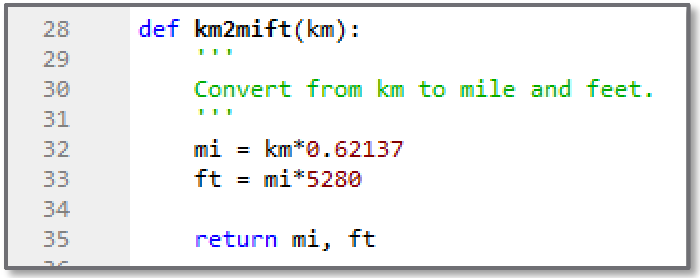
```
return (Tc, Tk)
```

- To return a list:

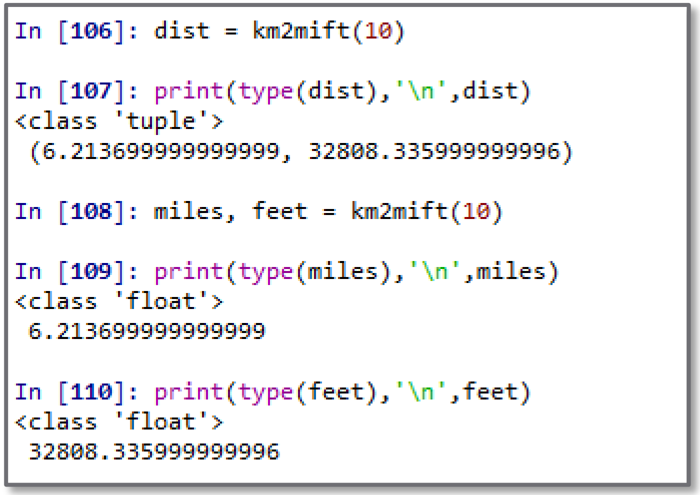
```
return [Tc, Tk]
```

Function – Example

- Consider a function that converts a distance in kilometers to a distance in both miles and feet
  - Outputs returned as tuple:

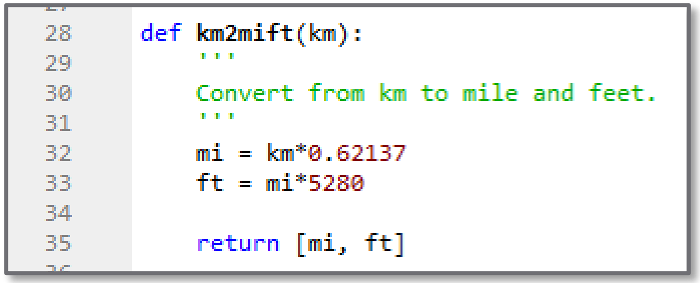


```
def km2mift(km):
    '''
    Convert from km to mile and feet.
    '''
    mi = km*0.62137
    ft = mi*5280

    return mi, ft
```

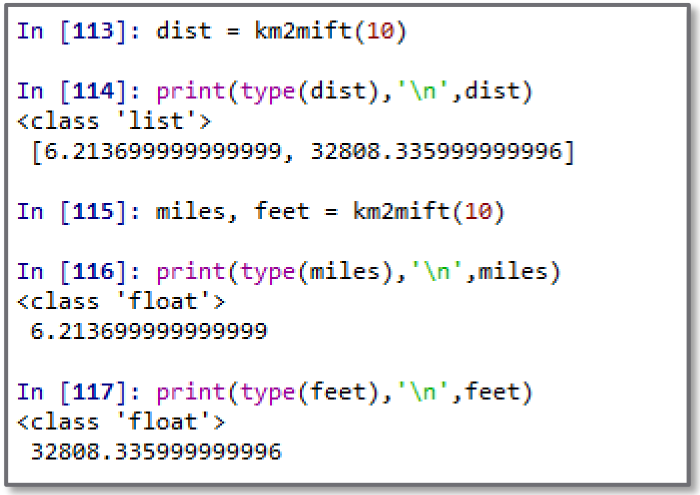


```
In [106]: dist = km2mift(10)

In [107]: print(type(dist),'\n',dist)
<class 'tuple'>
 (6.213699999999999, 32808.335999999996)

In [108]: miles, feet = km2mift(10)

In [109]: print(type(miles),'\n',miles)
<class 'float'>
 6.213699999999999

In [110]: print(type(feet),'\n',feet)
<class 'float'>
 32808.335999999996
```

  - Outputs returned as list:



```
def km2mift(km):
    '''
    Convert from km to mile and feet.
    '''
    mi = km*0.62137
    ft = mi*5280

    return [mi, ft]
```



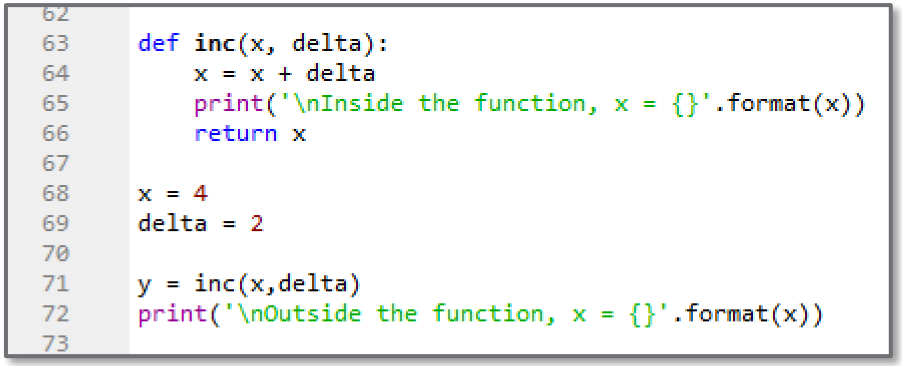
```
In [113]: dist = km2mift(10)

In [114]: print(type(dist),'\n',dist)
<class 'list'>
 [6.213699999999999, 32808.335999999996]

In [115]: miles, feet = km2mift(10)

In [116]: print(type(miles),'\n',miles)
<class 'float'>
 6.213699999999999

In [117]: print(type(feet),'\n',feet)
<class 'float'>
 32808.335999999996
```

Variable Scope

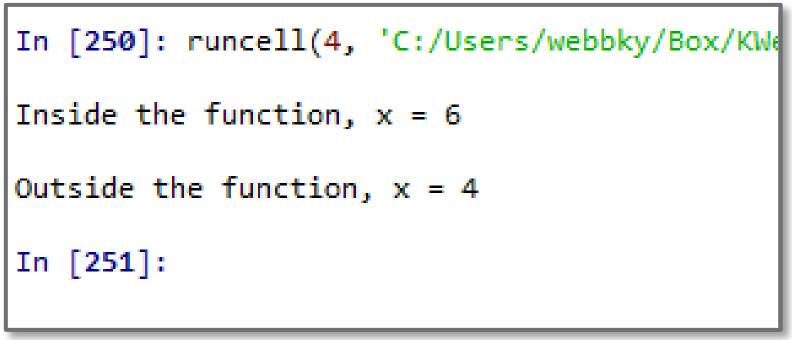
Variable Scope

- ***Inputs*** are values passed to a function
    - Defined in and passed from the calling script
    - Not defined within the function

- A function has its own ***namespace***
    - Separate set of local (to the function) variables and values
    - Variables may have the same names as in the calling script, but they are ***separate*** variables



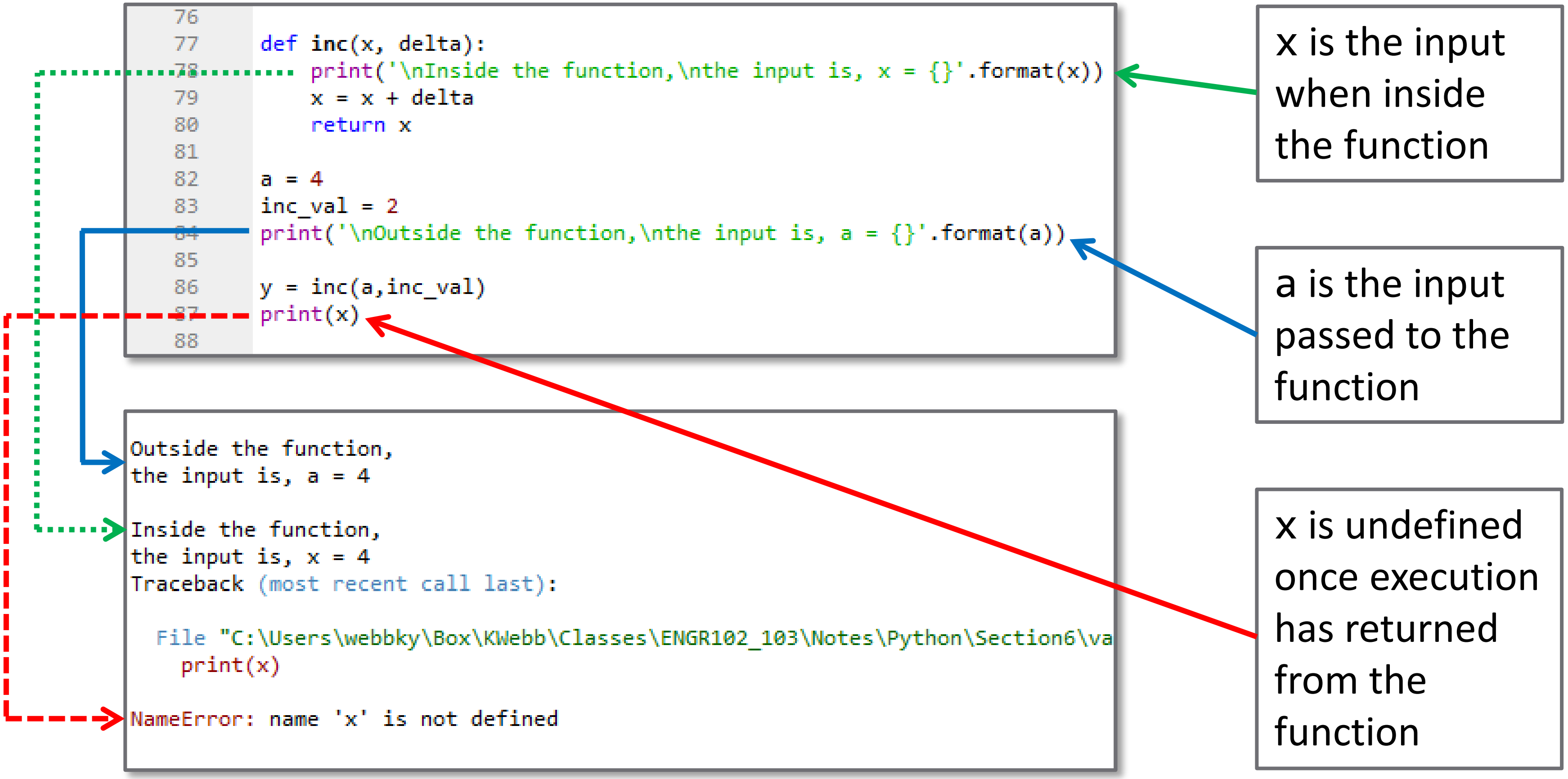
```
def inc(x, delta):
    x = x + delta
    print('\nInside the function, x = {}'.format(x))
    return x

x = 4
delta = 2

y = inc(x,delta)
print('\nOutside the function, x = {}'.format(x))
```



```
In [250]: runcell(4, 'C:/Users/webbky/Box/KW

Inside the function, x = 6

Outside the function, x = 4

In [251]:
```

Variable Scope

- Local function variables are not available in the enclosing script after returning from the function



```python

def inc(x, delta):
    print('\nInside the function,\nthe input is, x = {}'.format(x))
    x = x + delta
    return x

a = 4
inc_val = 2
print('\nOutside the function,\nthe input is, a = {}'.format(a))

y = inc(a,inc_val)
print(x)

```

```
Outside the function,
the input is, a = 4

Inside the function,
the input is, x = 4
Traceback (most recent call last):

  File "C:\Users\webbky\Box\KWebb\Classes\ENGR102_103\Notes\Python\Section6\va
    print(x)

NameError: name 'x' is not defined
```

x is the input when inside the function

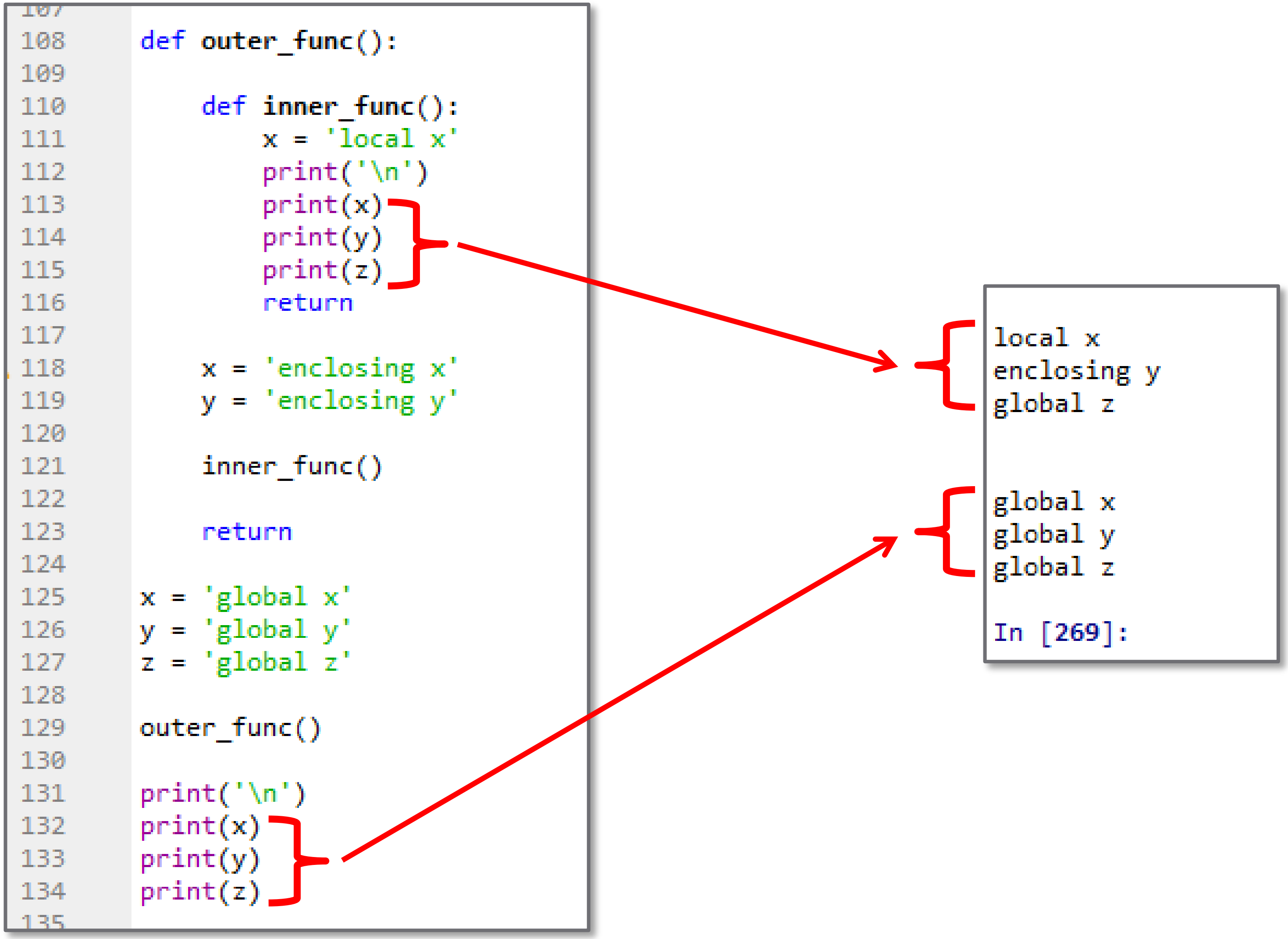
a is the input passed to the function

x is undefined once execution has returned from the function

Variable Scope - LEGB

- Python locates variables used in code according to the ***LEGB rule***
- ***Namespaces*** are searched in ***LEGB*** order to resolve variable names:
  - ***Local***: defined within the function
  - ***Enclosing***: defined in the outer (enclosing) function – applies only to nested functions
  - ***Global***: defined in the top-level script or module
  - ***Built-In***: defined in built-in Python libraries
- The first (in LEGB order) occurrence of a variable is used

Variable Scope - LEGB



```
def outer_func():

    def inner_func():
        x = 'local x'
        print('\n')
        print(x)
        print(y)
        print(z)
        return

    x = 'enclosing x'
    y = 'enclosing y'

    inner_func()

    return

x = 'global x'
y = 'global y'
z = 'global z'

outer_func()

print('\n')
print(x)
print(y)
print(z)
```

```
local x
enclosing y
global z


global x
global y
global z

In [269]:
```

Exercise – Define a Function

Exercise

- Write a script to:
  - Define a function, pwr, to raise an input to a power
    - x: positional input argument
    - pow: keyword input argument – default=2
    - Return: $y = x^{pow}$
  - Test your function using different inputs
    - With and without specifying pow

Function Docstrings

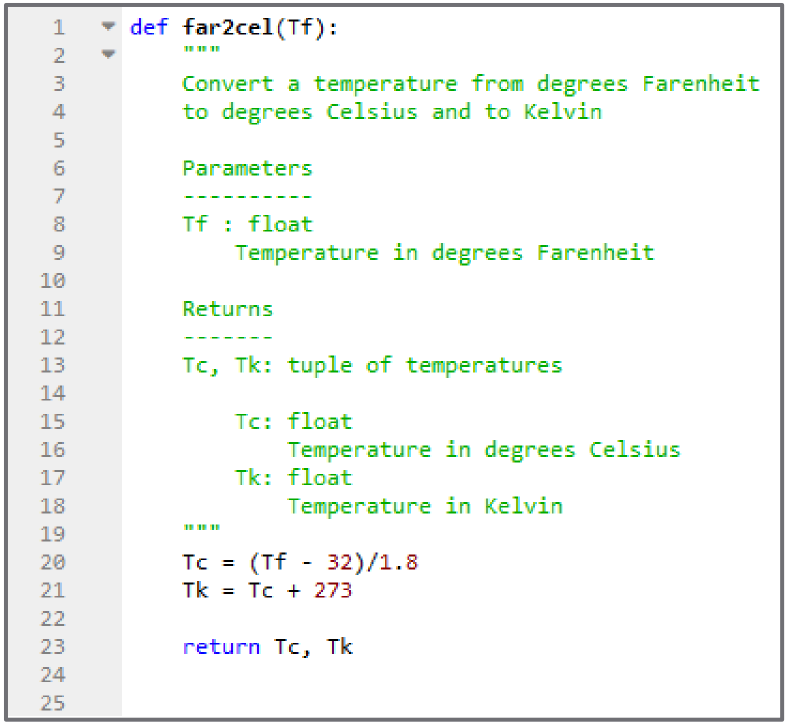
Function Docstrings

- Any function – built-in or user-defined – is accessible by the Spyder help system
  - Console: `help(`*`functionName`*`)`
  - Spyder help pane

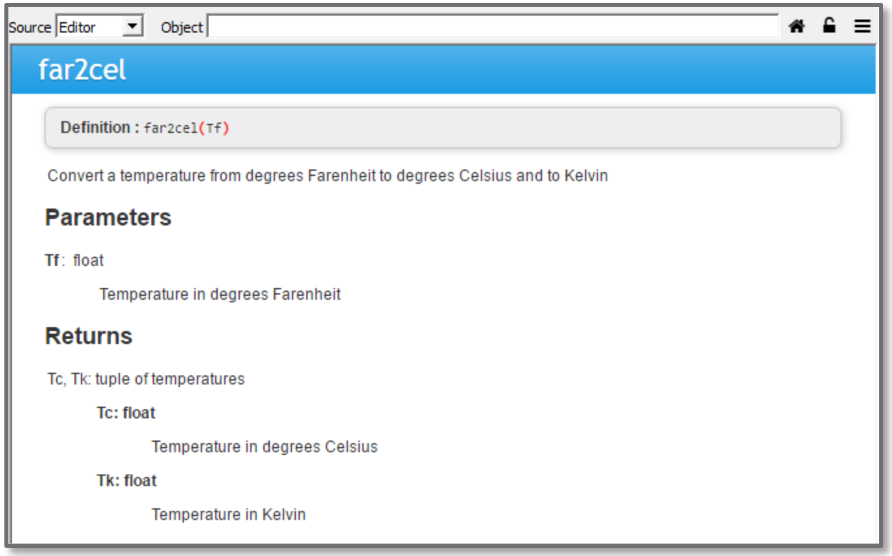
- Help text that appears is the function ***docstring***
  - Comment block following the function definition
  - Enclosed in triple-quotes
  - Describes function behavior, inputs, and outputs

- Docstrings serve as function documentation
  - Particularly important for functions
  - Often reused long after they are written
  - Often used by other users

Function Docstrings



```
def far2cel(Tf):
    """
    Convert a temperature from degrees Farenheit
    to degrees Celsius and to Kelvin

    Parameters
    ----------
    Tf : float
        Temperature in degrees Farenheit

    Returns
    -------
    Tc, Tk: tuple of temperatures

        Tc: float
            Temperature in degrees Celsius
        Tk: float
            Temperature in Kelvin
    """
    Tc = (Tf - 32)/1.8
    Tk = Tc + 273

    return Tc, Tk
```



far2cel

Definition : `far2cel(Tf)`

Convert a temperature from degrees Farenheit to degrees Celsius and to Kelvin

Parameters

Tf: float

Temperature in degrees Farenheit

Returns

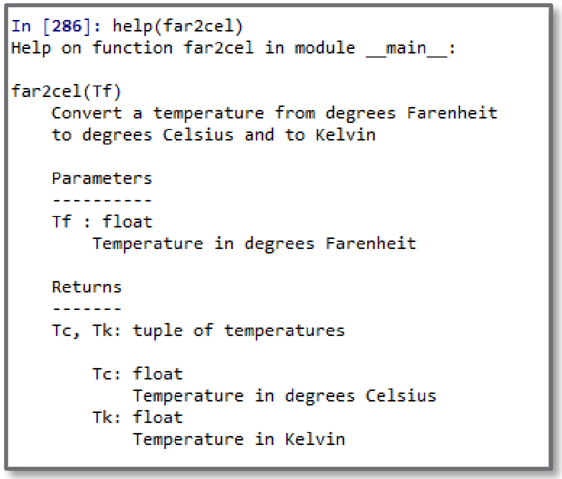
Tc, Tk: tuple of temperatures

Tc: float

Temperature in degrees Celsius

Tk: float

Temperature in Kelvin



```
In [286]: help(far2cel)
Help on function far2cel in module __main__:

far2cel(Tf)
    Convert a temperature from degrees Farenheit
    to degrees Celsius and to Kelvin

    Parameters
    ----------
    Tf : float
        Temperature in degrees Farenheit

    Returns
    -------
    Tc, Tk: tuple of temperatures

        Tc: float
            Temperature in degrees Celsius
        Tk: float
            Temperature in Kelvin
```

- The Spyder editor can automatically generate a function docstring
  - Click 'Generate docstring' popup that appears after typing the opening triple-quote, ''', in the function definition

Importing Modules and Functions

Importing Modules and Functions

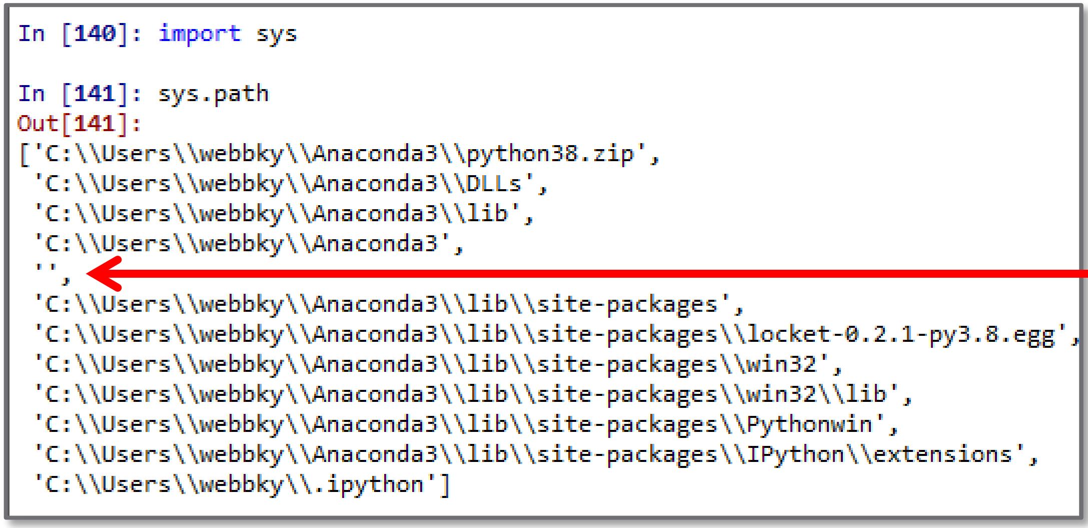
- When we run Python, built-in functions are loaded and accessible by default
- To access other functions, we must first ***import*** the corresponding ***packages*** and ***modules***
  - For example:

```
import numpy as np

from matplotlib import pyplot as plt
```

- We can do the same for our own ***user-defined functions***
  - Can use our user-defined functions in other scripts
  - Must ***import*** them first

The Python Path

- To import a module, it must be in the ***Python path***
  - That is, it must be saved in a directory (folder) that is included in the ***Python path***



```
In [140]: import sys

In [141]: sys.path
Out[141]:
['C:\\Users\\webbky\\Anaconda3\\python38.zip',
 'C:\\Users\\webbky\\Anaconda3\\DLLs',
 'C:\\Users\\webbky\\Anaconda3\\lib',
 'C:\\Users\\webbky\\Anaconda3',
 '',
 'C:\\Users\\webbky\\Anaconda3\\lib\\site-packages',
 'C:\\Users\\webbky\\Anaconda3\\lib\\site-packages\\locket-0.2.1-py3.8.egg',
 'C:\\Users\\webbky\\Anaconda3\\lib\\site-packages\\win32',
 'C:\\Users\\webbky\\Anaconda3\\lib\\site-packages\\win32\\lib',
 'C:\\Users\\webbky\\Anaconda3\\lib\\site-packages\\Pythonwin',
 'C:\\Users\\webbky\\Anaconda3\\lib\\site-packages\\IPython\\extensions',
 'C:\\Users\\webbky\\.ipython']
```

- Frequently-used user-defined functions:
  - Save under site-packages
  - Will always be able to import

- Path includes:
  - Default locations, as shown
  - Present working directory, PWD
- PWD always included
  - Can import anything from the same directory
  - Save related modules in a common directory

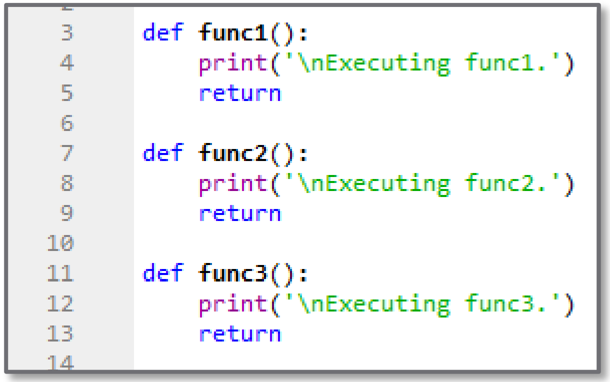
Importing Modules

- Several different ways to import modules and objects from modules
    - How a function is imported affects how it is called

```
import <module_name>

import <module_name> as <loc_name>

from <module_name> import <name>

from <module_name> import <name> as <loc_name>
```

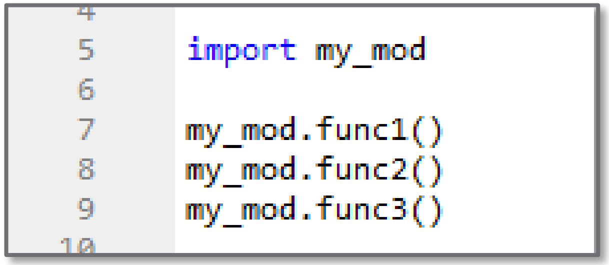
Importing Modules

- Import `my_mod.py` to another script



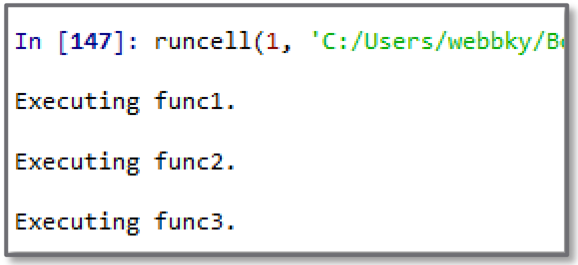
```python
def func1():
    print('\nExecuting func1.')
    return

def func2():
    print('\nExecuting func2.')
    return

def func3():
    print('\nExecuting func3.')
    return
```

- Import the ***entire module*** with the ***same name***
  - Call imported functions as: `my_mod.<fname()>`



```python
import my_mod

my_mod.func1()
my_mod.func2()
my_mod.func3()
```

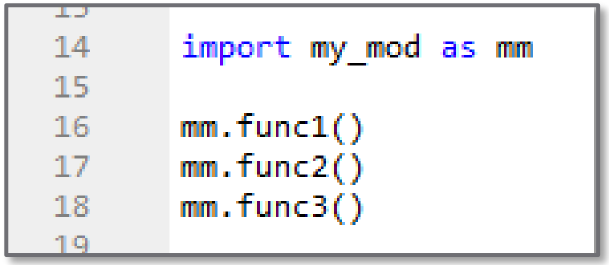
→



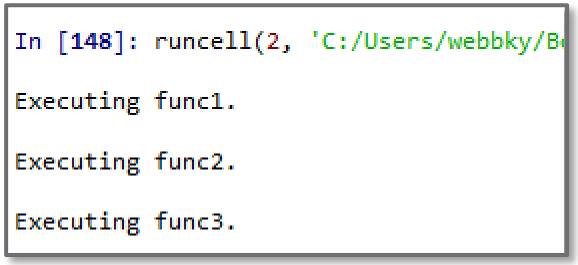
```
In [147]: runcell(1, 'C:/Users/webbky/B

Executing func1.

Executing func2.

Executing func3.
```

Importing Modules

- Import the ***entire module*** but give it a ***local name***
  - Call imported functions as: `<loc_name>.<fname()>`

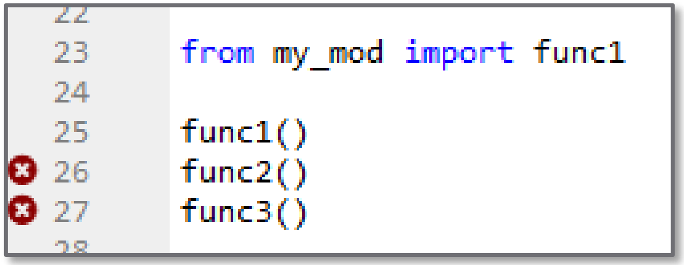


```python
import my_mod as mm

mm.func1()
mm.func2()
mm.func3()
```

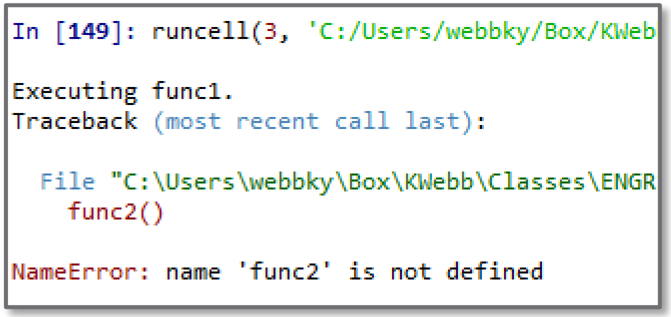


```
In [148]: runcell(2, 'C:/Users/webbky/B

Executing func1.

Executing func2.

Executing func3.
```

- Import a ***function from the module*** and ***keep its name***
  - Call imported functions as: `<fname()>`



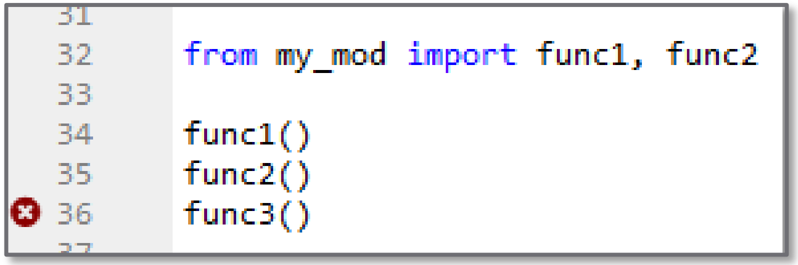
```python
from my_mod import func1

func1()
func2()
func3()
```



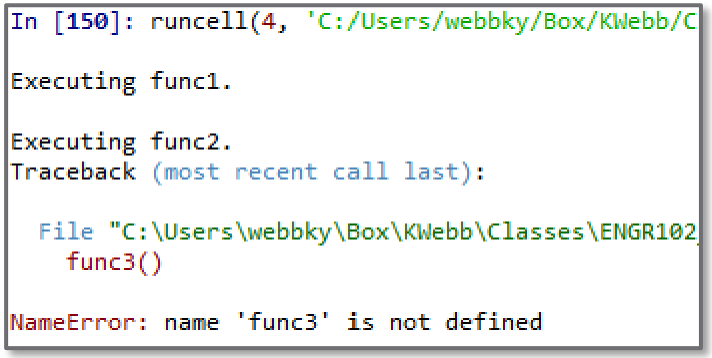
```
In [149]: runcell(3, 'C:/Users/webbky/Box/KWeb

Executing func1.
Traceback (most recent call last):

  File "C:\Users\webbky\Box\KWebb\Classes\ENGR
    func2()

NameError: name 'func2' is not defined
```

Importing Modules

- Import ***multiple functions, keeping names***
  - Call imported functions as: `<fname()>`

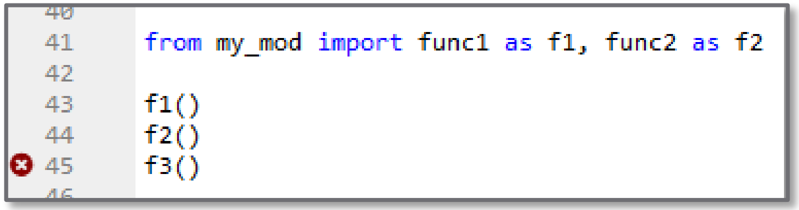


```python
from my_mod import func1, func2

func1()
func2()
func3()
```

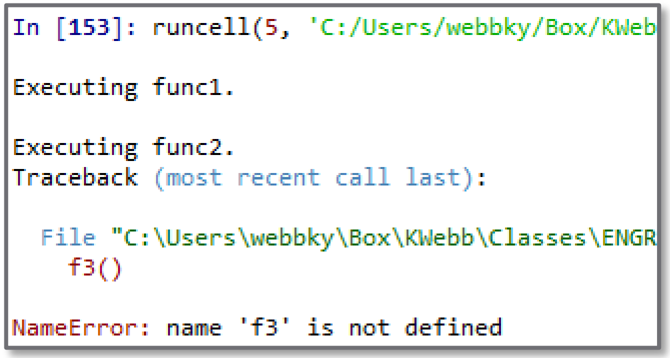


```
In [150]: runcell(4, 'C:/Users/webbky/Box/KWebb/C

Executing func1.

Executing func2.
Traceback (most recent call last):

  File "C:\Users\webbky\Box\KWebb\Classes\ENGR102
    func3()

NameError: name 'func3' is not defined
```

- Import ***multiple functions***, assigning ***local names***
  - Call imported functions as: `<loc_name()>`



```python
from my_mod import func1 as f1, func2 as f2

f1()
f2()
f3()
```



```
In [153]: runcell(5, 'C:/Users/webbky/Box/KWeb

Executing func1.

Executing func2.
Traceback (most recent call last):

  File "C:\Users\webbky\Box\KWebb\Classes\ENGR
    f3()

NameError: name 'f3' is not defined
```

Lambda Functions

Lambda Functions

- Python offers an alternative to the standard function definition syntax: ***Lambda functions***
  - Single-line functions
  - May or may not be named (may be anonymous)
  - Typically intended for one-time or temporary use
- Standard function definition:

```
def add3(x, y, z):
    return x + y + z
```

- Lambda function equivalent:

```
add3 = lambda x, y, z: x + y + z
```

Lambda Functions - Syntax

```
func = lambda arguments: expression
```

`'lambda'` keyword required

Function definition

- A ***single*** executable Python expression
- E.g. `X**2 + 3*y`

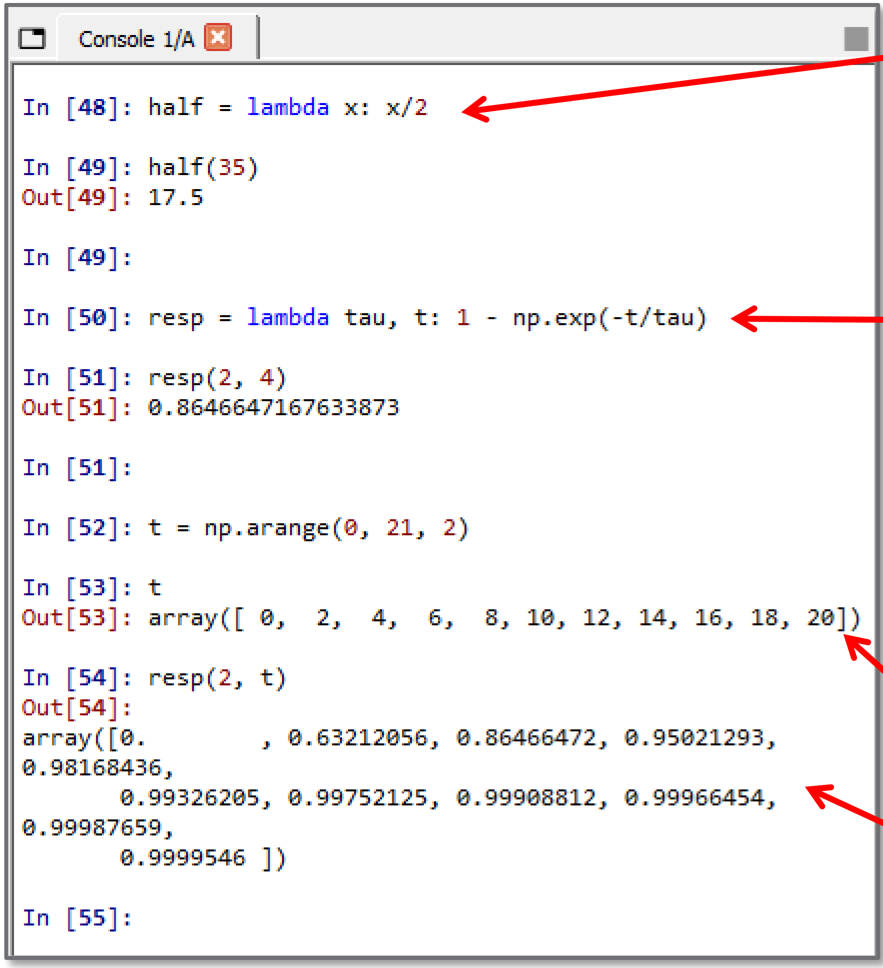
Function name

- Optional
- If not defined, it is an *anonymous function*

A list of input variables

- E.g. `x, y`
- Zero or more arguments
- Separated by commas
- Not enclosed in parentheses

Lambda Functions – Examples

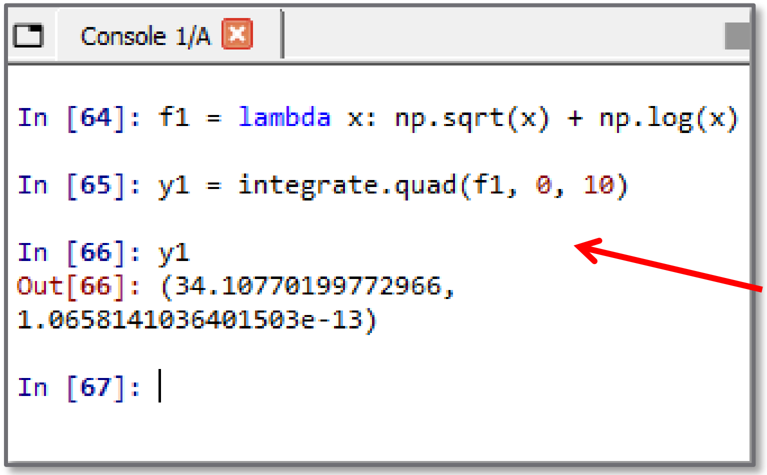


```
In [48]: half = lambda x: x/2

In [49]: half(35)
Out[49]: 17.5

In [49]:

In [50]: resp = lambda tau, t: 1 - np.exp(-t/tau)

In [51]: resp(2, 4)
Out[51]: 0.8646647167633873

In [51]:

In [52]: t = np.arange(0, 21, 2)

In [53]: t
Out[53]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20])

In [54]: resp(2, t)
Out[54]:
array([0.        , 0.63212056, 0.86466472, 0.95021293,
0.98168436,
       0.99326205, 0.99752125, 0.99908812, 0.99966454,
0.99987659,
       0.9999546 ])

In [55]:
```

- Simple function that returns half of the input value
- May have multiple inputs
  - First-order system response – inputs: time constant, value of time
- Inputs may be arrays
- Outputs may be arrays as well

Passing Functions to Functions

- We often want to perform functions on other functions
  - E.g. integration, roots finding, optimization, solution of differential equations
  - Lambda functions commonly passed as inputs to other functions

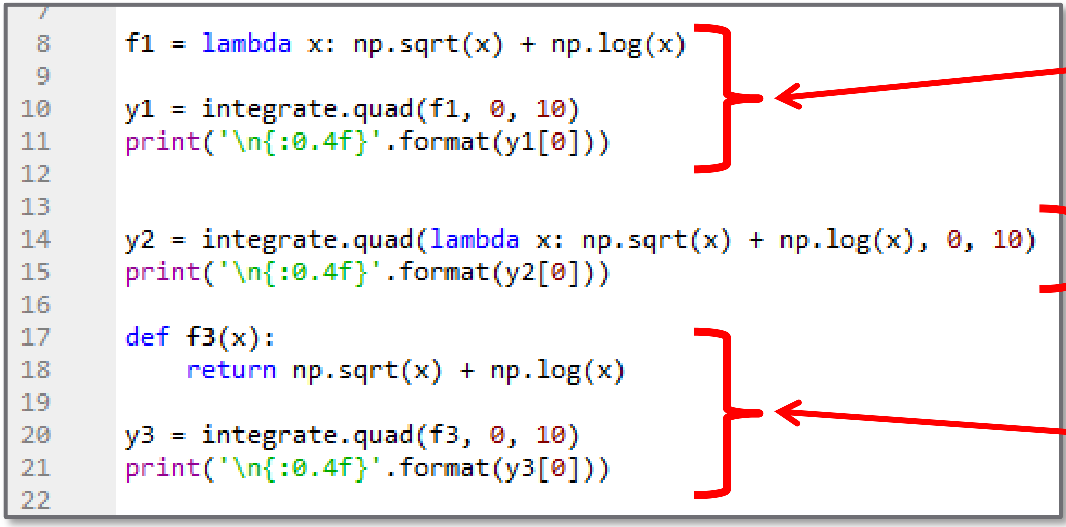


```
In [64]: f1 = lambda x: np.sqrt(x) + np.log(x)

In [65]: y1 = integrate.quad(f1, 0, 10)

In [66]: y1
Out[66]: (34.10770199772966,
1.0658141036401503e-13)

In [67]: |
```

- Define a lambda function
- Pass the function as an input to another function
- Here, integrate the function, f, from 0 to 10 using SciPy's `integrate.quad()` function

Passing Functions to Functions

- Several ways to pass functions to functions:



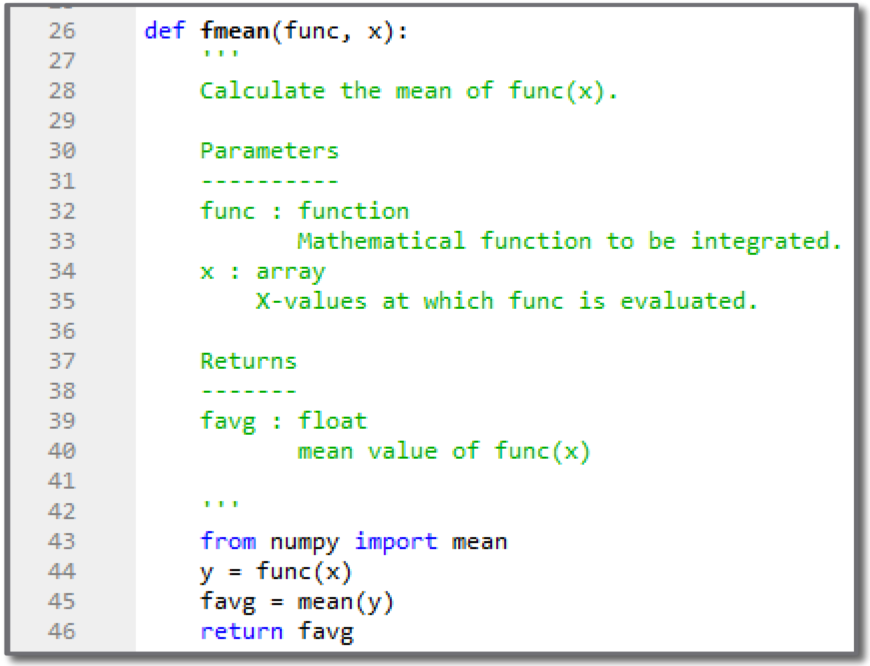
```
f1 = lambda x: np.sqrt(x) + np.log(x)

y1 = integrate.quad(f1, 0, 10)
print('\n{:0.4f}'.format(y1[0]))


y2 = integrate.quad(lambda x: np.sqrt(x) + np.log(x), 0, 10)
print('\n{:0.4f}'.format(y2[0]))

def f3(x):
    return np.sqrt(x) + np.log(x)

y3 = integrate.quad(f3, 0, 10)
print('\n{:0.4f}'.format(y3[0]))

```

- Named lambda function
- Anonymous function
- Standard function definition

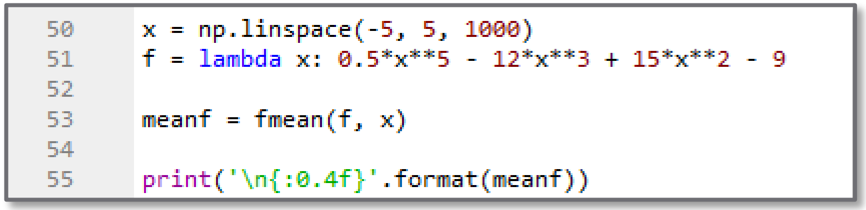
```
34.1077

34.1077

34.1077

In [68]: 
```

Function Function – Example

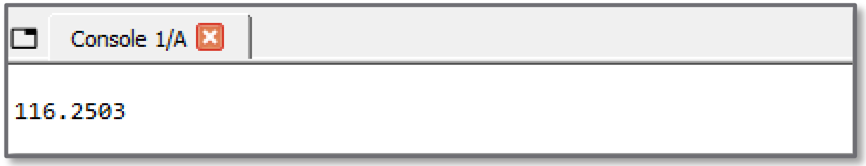
- Consider a function that calculates the mean of a mathematical function evaluated at a vector of independent variable values
- Inputs:
    - Function object
    - Vector of x values
- Output:
    - Mean value of $y = f(x)$



```python
def fmean(func, x):
    '''
    Calculate the mean of func(x).

    Parameters
    ----------
    func : function
           Mathematical function to be integrated.
    x : array
        X-values at which func is evaluated.

    Returns
    -------
    favg : float
           mean value of func(x)

    '''
    from numpy import mean
    y = func(x)
    favg = mean(y)
    return favg
```



```python
x = np.linspace(-5, 5, 1000)
f = lambda x: 0.5*x**5 - 12*x**3 + 15*x**2 - 9

meanf = fmean(f, x)

print('\n{:0.4f}'.format(meanf))
```



```
116.2503
```

Recursion

Recursive Functions

- ***Recursion*** is a problem solving approach in which a larger problem is solved by solving many smaller, self-similar problems

- A ***recursive function*** is one that calls itself
  - Each time it calls itself, it, again, calls itself

- Two components to a recursive function:
  - A ***base case***
    - A single case that can be solved without recursion
  - A ***general case***
    - A recursive relationship, ultimately leading to the base case

Recursion Example 1 – Factorial

- We have considered ***iterative*** algorithms for computing $y = n!$
  - for loop, while loop
- Factorial can also be computed using recursion
  - It can be defined with a base case and a general case:

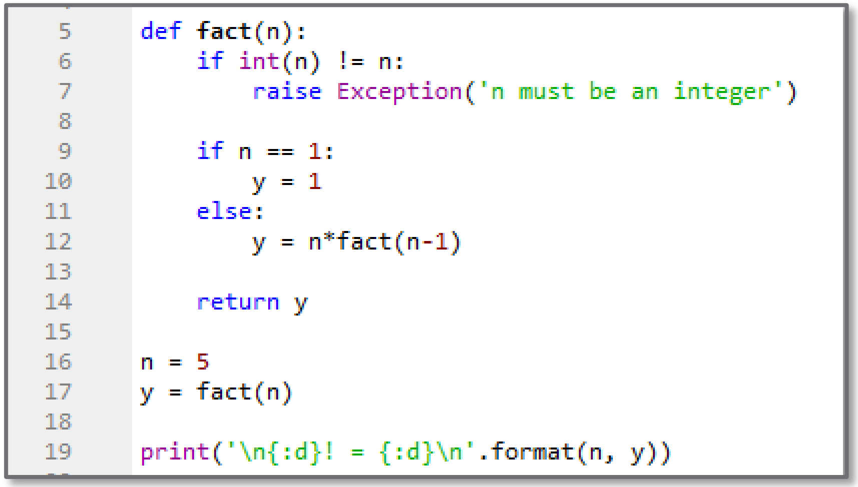
$$n! = \begin{cases} 1 & n = 1 \\ n * (n-1)! & n > 1 \end{cases}$$

  - The general case leads back to the base case
    - $n!$ defined in terms of $(n-1)!$, which is, in turn, defined in terms of $(n-2)!$, and so on
    - Ultimately, the base case, for $n = 1$, is reached

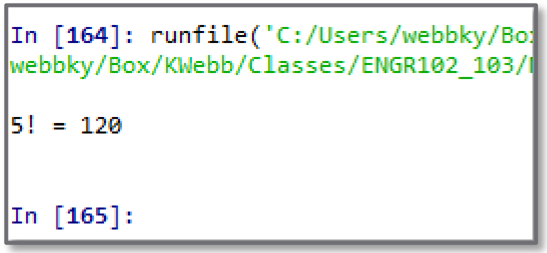
Recursion Example 1 – Factorial

$$n! = \begin{cases} 1 & x = 1 \\ x * (x-1)! & x > 1 \end{cases}$$

- The general case is a recursive relationship, because it defines the factorial function using the factorial function
  - The function calls itself
- In Python:

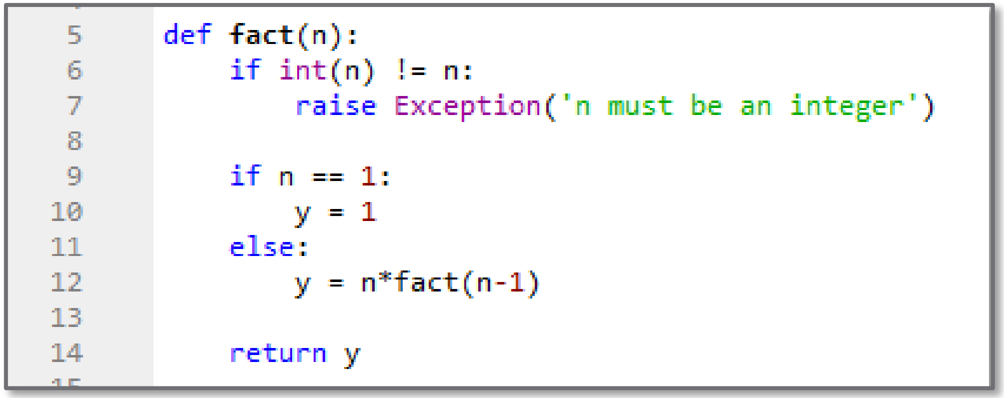


```python
def fact(n):
    if int(n) != n:
        raise Exception('n must be an integer')

    if n == 1:
        y = 1
    else:
        y = n*fact(n-1)

    return y

n = 5
y = fact(n)

print('\n{:d}! = {:d}\n'.format(n, y))
```



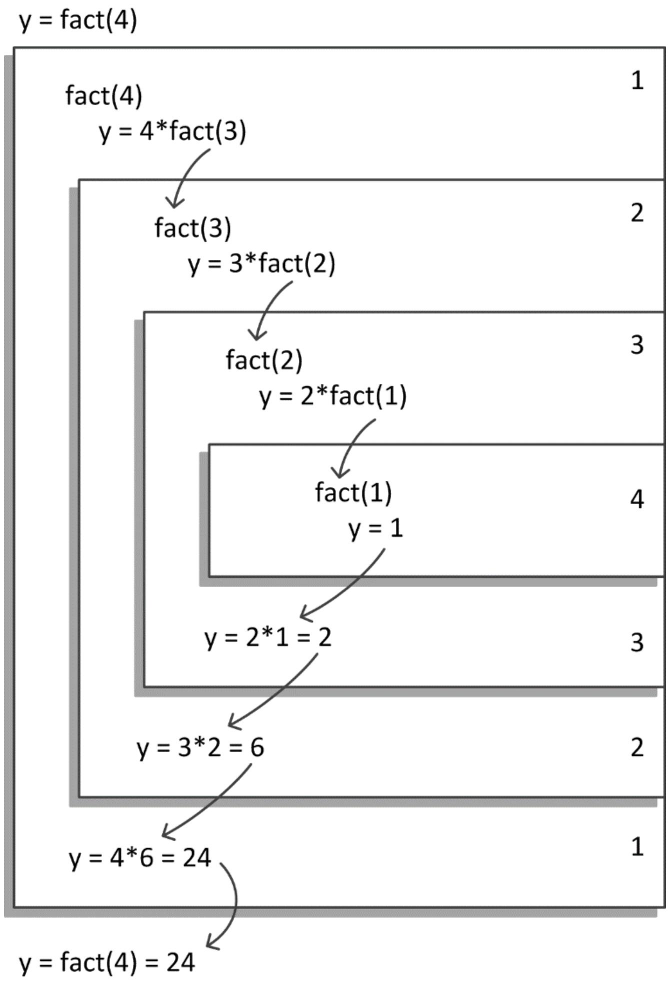
```
In [164]: runfile('C:/Users/webbky/Bo
webbky/Box/KWebb/Classes/ENGR102_103/

5! = 120


In [165]:
```

Recursion Example 1 – Factorial



```
def fact(n):
    if int(n) != n:
        raise Exception('n must be an integer')

    if n == 1:
        y = 1
    else:
        y = n*fact(n-1)

    return y
```

- Consider, for example: $y = 4!$
- `fact()` recursively called four times
- Fourth function call terminates first, once the base case is reached
- Function calls terminate in reverse order
    - Function call doesn’t terminate until all successive calls have terminated



y = fact(4)

fact(4)
y = 4*fact(3)
1

fact(3)
y = 3*fact(2)
2

fact(2)
y = 2*fact(1)
3

fact(1)
y = 1
4

y = 2*1 = 2
3

y = 3*2 = 6
2

y = 4*6 = 24
1

y = fact(4) = 24

Recursion Example 2 – Binary Search

- A common search algorithm is the ***binary search***
  - Similar to searching for a name in a phone book or a word in a dictionary
  - ***Look at the middle value*** to determine if the search item is in the ***upper or lower half***
  - Look at the middle value of the half that contains the search item to determine if it is in that half's upper or lower half, …
- The ***search function gets called recursively***, each time on half of the previous set
  - Search range shrinks by half on each function call
  - Recursion continues until the middle value is the search item – this is the required ***base case***

Recursion Example 2 – Binary Search

- ***Recursive binary search*** – the basic algorithm:
  - Find the index, i, of x in the sorted list, A, in the range of $A\left(i_{low}: i_{high}\right)$

1) Calculate the middle index of $A\left(i_{low}: i_{high}\right)$:

$$i_{mid} = \mathrm{floor}\left(\frac{i_{low} + i_{high}}{2}\right)$$

2) If $A(i_{mid}) == x$, then $i = i_{mid}$, and we're done
3) If $A(i_{mid}) > x$, repeat the algorithm for $A(i_{low}: i_{mid} - 1)$
4) If $A(i_{mid}) < x$, repeat the algorithm for $A\left(i_{mid} + 1: i_{high}\right)$

Recursion Example 2 – Binary Search

- Find the index of the $x = 9$ in:

$$A = [0, 1, 3, 5, \boxed{6,} 7, 9, 12, 16, 20]$$

- $A[i_{mid}] = A[4] = 6$
  - $A[i_{mid}] < x$
  - Start over for $A[5{:}10]$

$$A = [0, 1, 3, 5, 6, 7, 9, \boxed{12,} 16, 20]$$

- $A[i_{mid}] = A[7] = 12$
  - $A[i_{mid}] > x$
  - Start over for $A[5{:}7]$

$$A = [0, 1, 3, 5, 6, \boxed{7,} 9, 12, 16, 20]$$

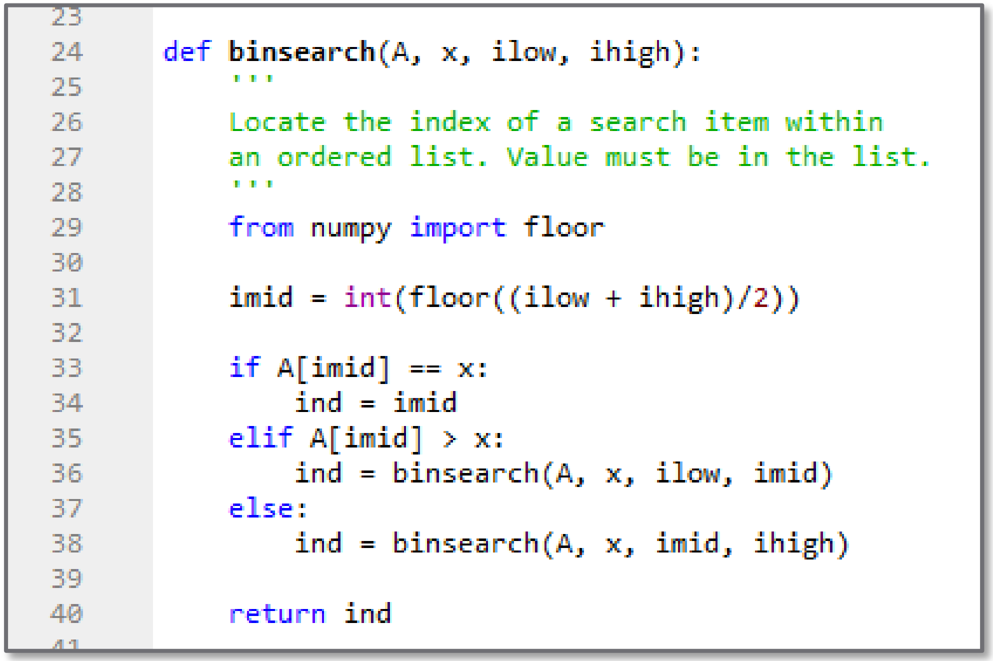
- $A[i_{mid}] = A[5] = 7$
  - $A[i_{mid}] < x$
  - Start over for $A[6]$

$$A = [0, 1, 3, 5, 6, 7, \boxed{9,} 12, 16, 20]$$

- $A[i_{mid}] = A[6] = 9$
  - $A[i_{mid}] == x$
  - $\boxed{i = i_{mid} = 6}$

Recursion Example 2 – Binary Search

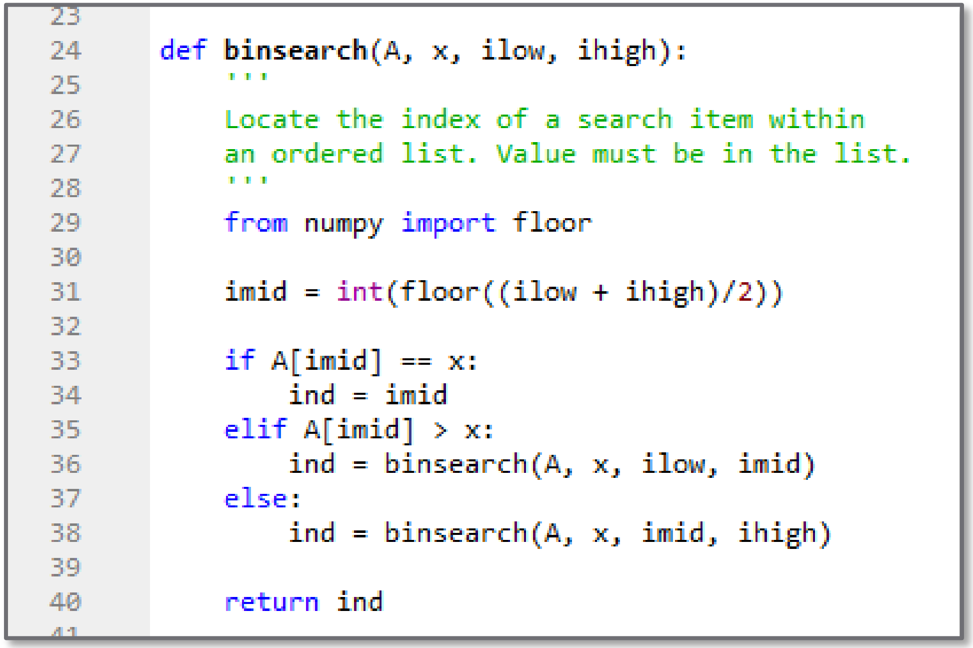
- Recursive binary search algorithm in Python
- Base case for `A[imid] == x`
- Function is called recursively on successively halved ranges until base case is reached



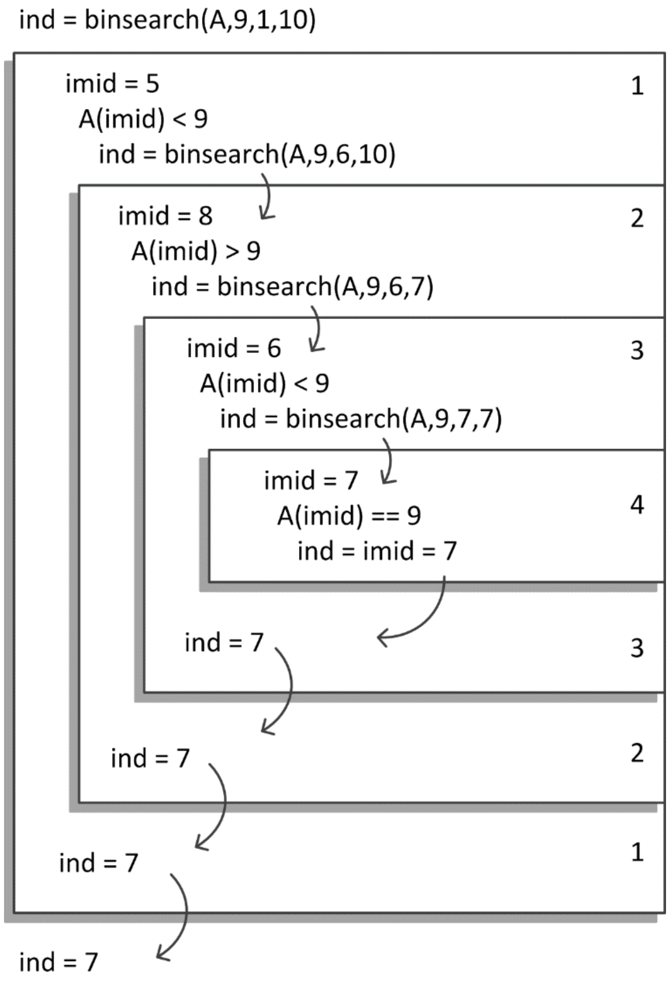
```python
def binsearch(A, x, ilow, ihigh):
    '''
    Locate the index of a search item within
    an ordered list. Value must be in the list.
    '''
    from numpy import floor

    imid = int(floor((ilow + ihigh)/2))

    if A[imid] == x:
        ind = imid
    elif A[imid] > x:
        ind = binsearch(A, x, ilow, imid)
    else:
        ind = binsearch(A, x, imid, ihigh)

    return ind
```

Recursion Example 2 – Binary Search

- A=[0,1,3,5,6,7,9,12,16,20]
- x=9
- ind = binsearch(A,x,1,10)
  - ind = 7



```python
def binsearch(A, x, ilow, ihigh):
    '''
    Locate the index of a search item within
    an ordered list. Value must be in the list.
    '''
    from numpy import floor

    imid = int(floor((ilow + ihigh)/2))

    if A[imid] == x:
        ind = imid
    elif A[imid] > x:
        ind = binsearch(A, x, ilow, imid)
    else:
        ind = binsearch(A, x, imid, ihigh)

    return ind
```



ind = binsearch(A,9,1,10)

1
imid = 5
A(imid) < 9
ind = binsearch(A,9,6,10)

2
imid = 8
A(imid) > 9
ind = binsearch(A,9,6,7)

3
imid = 6
A(imid) < 9
ind = binsearch(A,9,7,7)

4
imid = 7
A(imid) == 9
ind = imid = 7

3
ind = 7

2
ind = 7

1
ind = 7

ind = 7