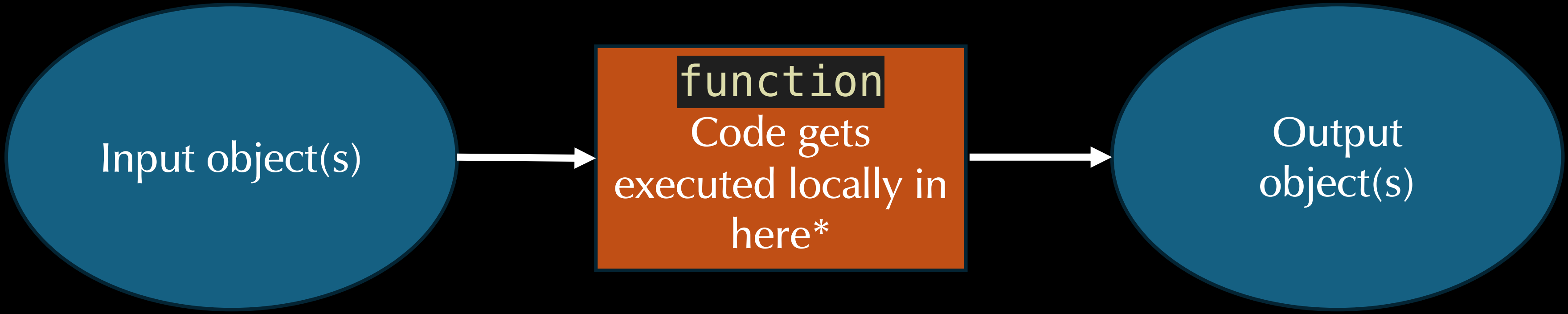
MR2020: Coding for METOC

Module 7: Introduction to Functions

What are functions?

Functions are reusable blocks of code that only run when called. They are useful for actions that need to be repeated many times so that the programmer need not copy/paste the same code repeatedly.



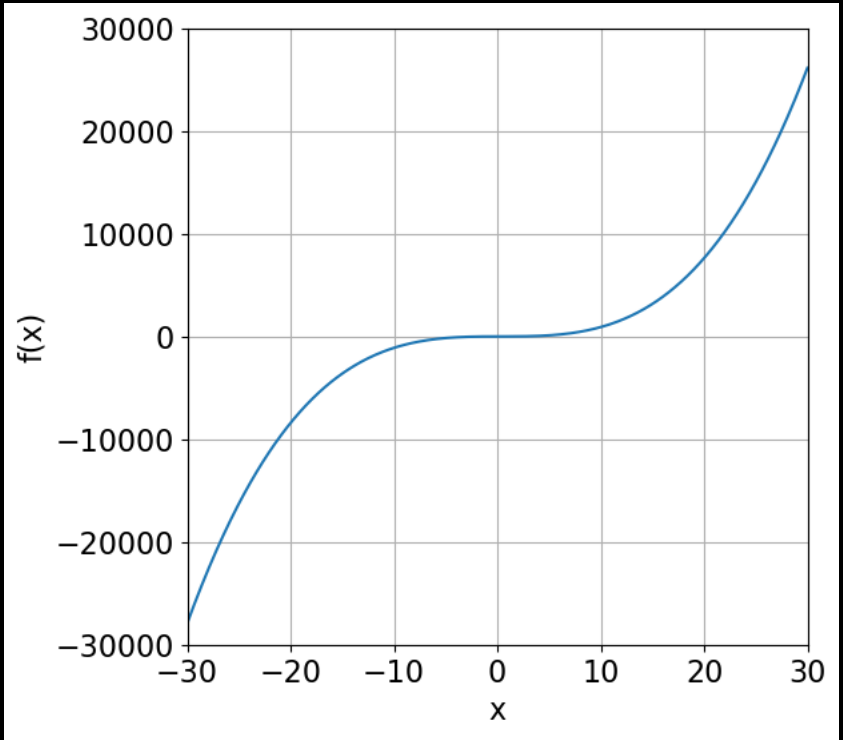
*Objects created inside the function have a scope limited to that function (unless they are made global).

A simple function

Let's consider a simple mathematical function first:

$$f(x) = x^3 - x^2 + x - 1$$

Basic idea: Plug in something for *x*. Get something back for *f*(*x*).



A simple function

Let’s consider a simple mathematical function first:

$$f(x) = x^3 - x^2 + x - 1$$

Basic idea: Plug in something for *x*. Get something back for *f*(*x*).

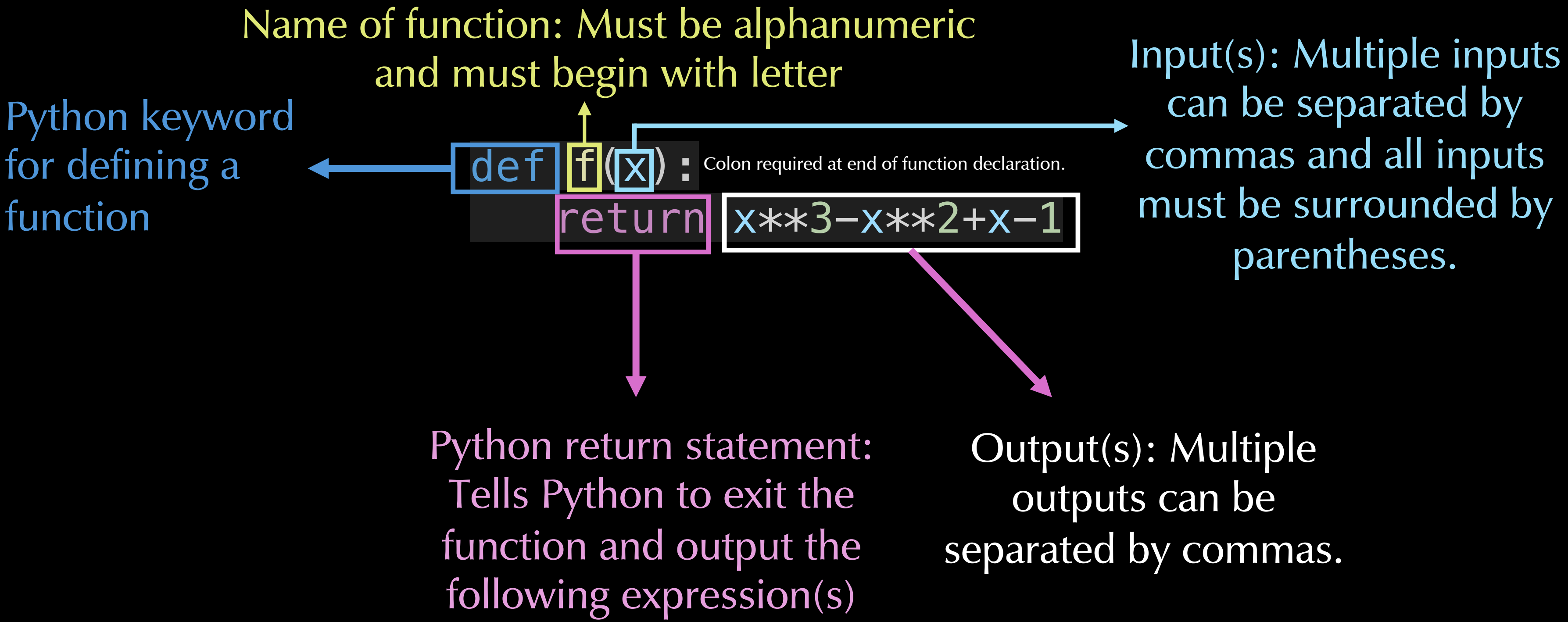
```
import numpy as np

def f(x):
    return x**3-x**2+x-1

x = np.arange(-30,30.1,0.1)
y = f(x)

# Code that plots y against x.
# ...
```

A simple function

Let's consider a simple mathematical function first:

$$f(x) = x^3 - x^2 + x - 1$$

Basic idea: Plug in something for *x*. Get something back for *f*(*x*).



```
def f(x):
    return x**3-x**2+x-1
```

Name of function: Must be alphanumeric and must begin with letter

Python keyword for defining a function

Input(s): Multiple inputs can be separated by commas and all inputs must be surrounded by parentheses.

Colon required at end of function declaration.

Python return statement: Tells Python to exit the function and output the following expression(s)

Output(s): Multiple outputs can be separated by commas.

More complicated function example*

*Just an example from research.

```python
def process_row(row):
    # Find the x, y, z location
    # Initialize search radius
    xc = matchcoordinatetoindex(scale_change*row.x,xh)
    yc = matchcoordinatetoindex(scale_change*row.y,yh)
    zc = matchcoordinatetoindex(scale_change*(row.z-row.zs),zh)

    boxsize = 50 # Size of box to check around point for cloud.

    if mask[zc,yc,xc] == 0 or mask[zc,yc,xc] == 1:
        row['cloud_edge_distance'] = None
    else:
        # Calculate distance to every point within 100 grid cells.
        # Make a meshgrid of x and y distances surrounding the center point.
        xd, yd = np.meshgrid(range(-boxsize,boxsize+1),range(-boxsize,boxsize+1))
        # Get a boxsize by boxsize matrix of distances from the center point.
        dd = 100*np.sqrt(xd**2+yd**2)
        # Calculate the distances and choose the smallest one that meets the criteria.
        smallmask = mask[zc,yc-boxsize:yc+boxsize+1,xc-boxsize:xc+boxsize+1]
        # Set to True if smallmask is NOT cloud.
        cond = (smallmask == 0) | (smallmask == 1)
        try:
        # Find minimum distance to a NOT cloudy point.
        row['cloud_edge_distance'] = dd[cond].min()
        except: # Sometimes the parcel is on the edge of the domain, and we haven't yet
        # handled wrapping the distance calculation around the domain. So for now,
        # we'll just make these have a "bad distance", then figure it out later if
        # needed.
        row['cloud_edge_distance'] = None

    return row
```

More complicated function example

Indented

```
def process_row(row):
    # Find the x, y, z location
    # Initialize search radius
    xc = matchcoordinatetoindex(scale_change*row.x,xh)
    yc = matchcoordinatetoindex(scale_change*row.y,yh)
    zc = matchcoordinatetoindex(scale_change*(row.z-row.zs),zh)

    boxsize = 50 # Size of box to check around point for cloud.

    if mask[zc,yc,xc] == 0 or mask[zc,yc,xc] == 1:
        row['cloud_edge_distance'] = None
    else:
        # Calculate distance to every point within 100 grid cells.
        # Make a meshgrid of x and y distances surrounding the center point.
        xd, yd = np.meshgrid(range(-boxsize,boxsize+1),range(-boxsize,boxsize+1))
        # Get a boxsize by boxsize matrix of distances from the center point.
        dd = 100*np.sqrt(xd**2+yd**2)
        # Calculate the distances and choose the smallest one that meets the criteria.
        smallmask = mask[zc,yc-boxsize:yc+boxsize+1,xc-boxsize:xc+boxsize+1]
        # Set to True if smallmask is NOT cloud.
        cond = (smallmask == 0) | (smallmask == 1)
        try:
        # Find minimum distance to a NOT cloudy point.
        row['cloud_edge_distance'] = dd[cond].min()
        except: # Sometimes the parcel is on the edge of the domain, and we haven't yet
        # handled wrapping the distance calculation around the domain. So for now,
        # we'll just make these have a "bad distance", then figure it out later if
        # needed.
        row['cloud_edge_distance'] = None

    return row
```

More complicated function example

A bunch of stuff happens in this box.

```
def process_row(row):
    # Find the x, y, z location
    # Initialize search radius
    xc = matchcoordinatetoindex(scale_change*row.x,xh)
    yc = matchcoordinatetoindex(scale_change*row.y,yh)
    zc = matchcoordinatetoindex(scale_change*(row.z-row.zs),zh)

    boxsize = 50 # Size of box to check around point for cloud.

    if mask[zc,yc,xc] == 0 or mask[zc,yc,xc] == 1:
        row['cloud_edge_distance'] = None
    else:
        # Calculate distance to every point within 100 grid cells.
        # Make a meshgrid of x and y distances surrounding the center point.
        xd, yd = np.meshgrid(range(-boxsize,boxsize+1),range(-boxsize,boxsize+1))
        # Get a boxsize by boxsize matrix of distances from the center point.
        dd = 100*np.sqrt(xd**2+yd**2)
        # Calculate the distances and choose the smallest one that meets the criteria.
        smallmask = mask[zc,yc-boxsize:yc+boxsize+1,xc-boxsize:xc+boxsize+1]
        # Set to True if smallmask is NOT cloud.
        cond = (smallmask == 0) | (smallmask == 1)
        try:
        # Find minimum distance to a NOT cloudy point.
        row['cloud_edge_distance'] = dd[cond].min()
        except: # Sometimes the parcel is on the edge of the domain, and we haven't yet
        # handled wrapping the distance calculation around the domain. So for now,
        # we'll just make these have a "bad distance", then figure it out later if
        # needed.
        row['cloud_edge_distance'] = None

    return row
```

More complicated function example

A bunch of stuff happens in this box.

```
def process_row(row):
    # Find the x, y, z location
    # Initialize search radius
    xc = matchcoordinatetoindex(scale_change*row.x,xh)
    yc = matchcoordinatetoindex(scale_change*row.y,yh)
    zc = matchcoordinatetoindex(scale_change*(row.z-row.zs),zh)

    boxsize = 50 # Size of box to check around point for cloud.

    if mask[zc,yc,xc] == 0 or mask[zc,yc,xc] == 1:
         row['cloud_edge_distance'] = None
    else:
         # Calculate distance to every point within 100 grid cells.
         # Make a meshgrid of x and y distances surrounding the center point.
         xd, yd = np.meshgrid(range(-boxsize,boxsize+1),range(-boxsize,boxsize+1))
         # Get a boxsize by boxsize matrix of distances from the center point.
         dd = 100*np.sqrt(xd**2+yd**2)
         # Calculate the distances and choose the smallest one that meets the criteria.
         smallmask = mask[zc,yc-boxsize:yc+boxsize+1,xc-boxsize:xc+boxsize+1]
         # Set to True if smallmask is NOT cloud.
         cond = (smallmask == 0) | (smallmask == 1)
         try:
         # Find minimum distance to a NOT cloudy point.
         row['cloud_edge_distance'] = dd[cond].min()
         except: # Sometimes the parcel is on the edge of the domain, and we haven't yet
         # handled wrapping the distance calculation around the domain. So for now,
         # we'll just make these have a "bad distance", then figure it out later if
         # needed.
         row['cloud_edge_distance'] = None

    return row
```

Variables defined in function are local in scope, meaning they are forgotten after the function runs.

More complicated function example

```
def process_row(row):
    # Find the x, y, z location
    # Initialize search radius
    xc = matchcoordinatetoindex(scale_change*row.x,xh)
    yc = matchcoordinatetoindex(scale_change*row.y,yh)
    zc = matchcoordinatetoindex(scale_change*(row.z-row.zs),zh)

    boxsize = 50 # Size of box to check around point for cloud.

    if mask[zc,yc,xc] == 0 or mask[zc,yc,xc] == 1:
        row['cloud_edge_distance'] = None
    else:
        # Calculate distance to every point within 100 grid cells.
        # Make a meshgrid of x and y distances surrounding the center point.
        xd, yd = np.meshgrid(range(-boxsize,boxsize+1),range(-boxsize,boxsize+1))
        # Get a boxsize by boxsize matrix of distances from the center point.
        dd = 100*np.sqrt(xd**2+yd**2)
        # Calculate the distances and choose the smallest one that meets the criteria.
        smallmask = mask[zc,yc-boxsize:yc+boxsize+1,xc-boxsize:xc+boxsize+1]
        # Set to True if smallmask is NOT cloud.
        cond = (smallmask == 0) | (smallmask == 1)
        try:
        # Find minimum distance to a NOT cloudy point.
        row['cloud_edge_distance'] = dd[cond].min()
        except: # Sometimes the parcel is on the edge of the domain, and we haven't yet
        # handled wrapping the distance calculation around the domain. So for now,
        # we'll just make these have a "bad distance", then figure it out later if
        # needed.
        row['cloud_edge_distance'] = None

    return row
```

Eventually, there is a return statement. In this case, the output is the same as the input, meaning that this function is modifying the input in some way and feeding it back to the parent code.

Variable scope

An object defined outside a function has a **global** scope. It can be accessed both inside and out of functions.

An object defined inside a function has a **local** scope. It can only be accessed inside the function in which it is created.

```
def calc_greatest(list1, list2):
    """
    Inputs: Two lists of equal lengths containing numbers.
    Output: List containing largest elementwise values between the inputs.
    ""”
    if len(list1) != len(list2):
        raise(Exception('Lists are not the same length.'))

    newlist = []
    for i, j in zip(list1,list2):
        newlist.append(max(i,j))

    return newlist
```

```
A = [3, 5, 9]
B = [1, 8, 2]
greatest = calc_greatest(A,B)
```

This part is global. `A`, `B`, and `greatest` are global objects.

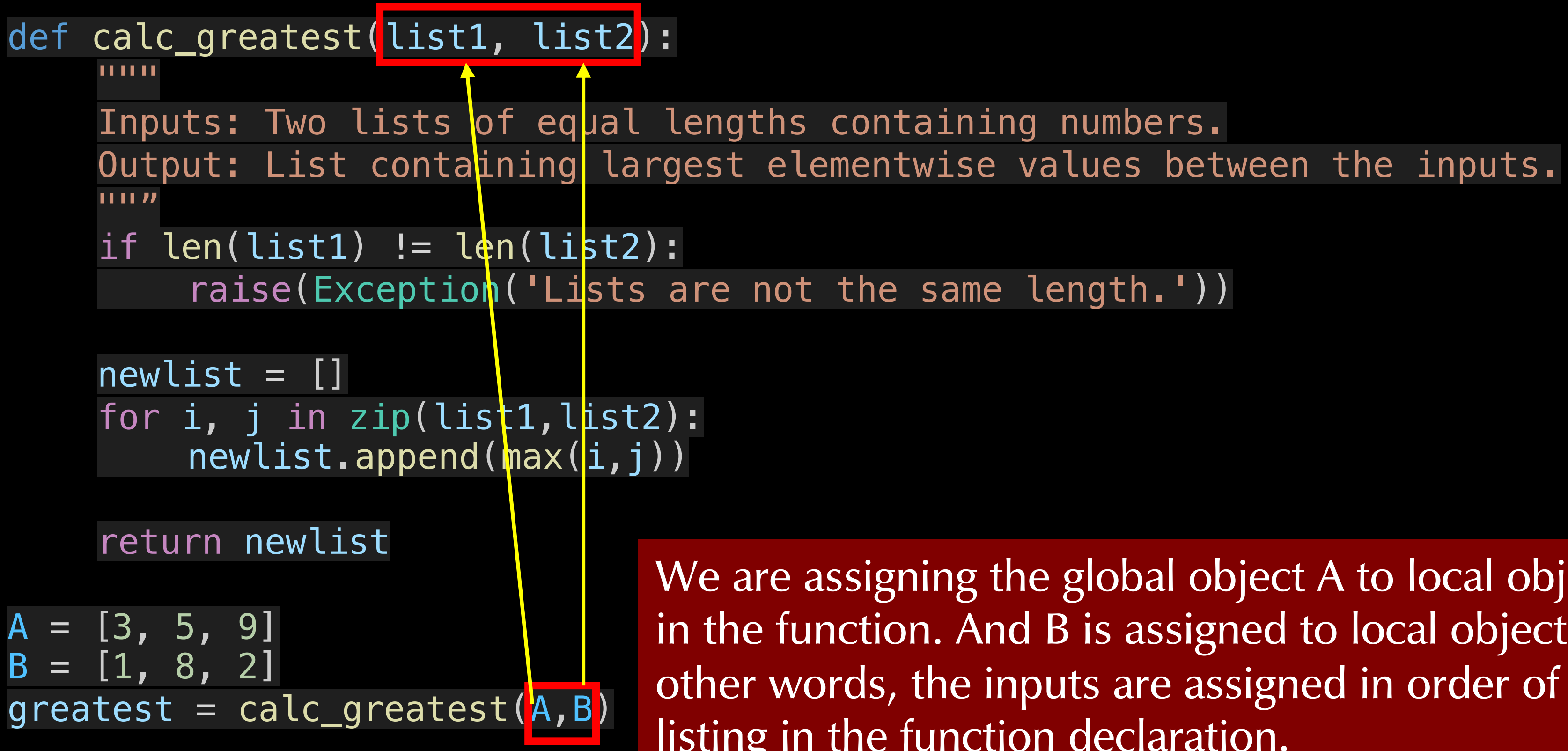
This is the function. Everything that happens in here is local.

```
def calc_greatest(list1, list2):
    """
    Inputs: Two lists of equal lengths containing numbers.
    Output: List containing largest elementwise values between the inputs.
    ""”
    if len(list1) != len(list2):
        raise(Exception('Lists are not the same length.'))

    newlist = []
    for i, j in zip(list1,list2):
        newlist.append(max(i,j))

    return newlist
```

```
A = [3, 5, 9]
B = [1, 8, 2]
greatest = calc_greatest(A,B)
```

And this line calls the function.

The function expects two inputs. They are separated by commas and surrounded by parenthesis on the line that defines the function. The object names have local scope inside the function.



```
def calc_greatest(list1, list2):
    """
    Inputs: Two lists of equal lengths containing numbers.
    Output: List containing largest elementwise values between the inputs.
    ""”
    if len(list1) != len(list2):
        raise(Exception('Lists are not the same length.'))

    newlist = []
    for i, j in zip(list1,list2):
        newlist.append(max(i,j))

    return newlist

A = [3, 5, 9]
B = [1, 8, 2]
greatest = calc_greatest(A,B)
```

We are assigning the global object A to local object list1 in the function. And B is assigned to local object list2. In other words, the inputs are assigned in order of their listing in the function declaration.

If there were modifications to list1 and list2 in the function, they would do nothing to change A and B outside the function.

```
def calc_greatest(list1, list2):
    """
    Inputs: Two lists of equal lengths containing numbers.
    Output: List containing largest elementwise values between the inputs.
    ""”
    if len(list1) != len(list2):
        raise(Exception('Lists are not the same length.'))

    newlist = []
    for i, j in zip(list1,list2):
        newlist.append(max(i,j))

    return newlist
```

The object newlist and iteration variables i and j only exist inside the function. They have local scope.

```
A = [3, 5, 9]
B = [1, 8, 2]
greatest = calc_greatest(A,B)
# Objects list1, list2, newlist, i, j created in function no longer exist here.
```

Lambda functions

Simple functions can be expressed as lambda functions. Lambda functions are anonymous, meaning they are not a specific named function that can be called anywhere in the code. They can contain any number of arguments but only one expression.

They are best used for simple, repeated operations. For example, earlier we saw:

```
def f(x):
    return x**3-x**2+x-1

x = np.arange(-30,30.1,0.1)
y = f(x)
```

As a lambda function, this could look like:

```
x = np.arange(-30,30.1,0.1)
y = [(lambda x : x**3-x**2+x-1)(x)for x in x]
# y gets the output from all values of x without ever
# explicitly defining a function.
```

Where do functions go?

Python interprets code from top to bottom, so functions must be defined before they called (i.e., higher up in the script). Therefore, it is recommended for beginners to define functions either

1) at the top of the code after all necessary module imports

or

2) in a separate file that can be imported.

```python
A = [3, 5, 9]
B = [1, 8, 2]

# Define the function first!
def addlists(A,B):
    return A + B

# Call the function.
addlists(A,B)
```

```python
# The function addlists is now in a
file in the same directory called
otherfile.py
from otherfile import addlists

A = [3, 5, 9]
B = [1, 8, 2]

# Call the function.
addlists(A,B)
```