Functions, Scope & Arguments

HORT 530

Lecture 12

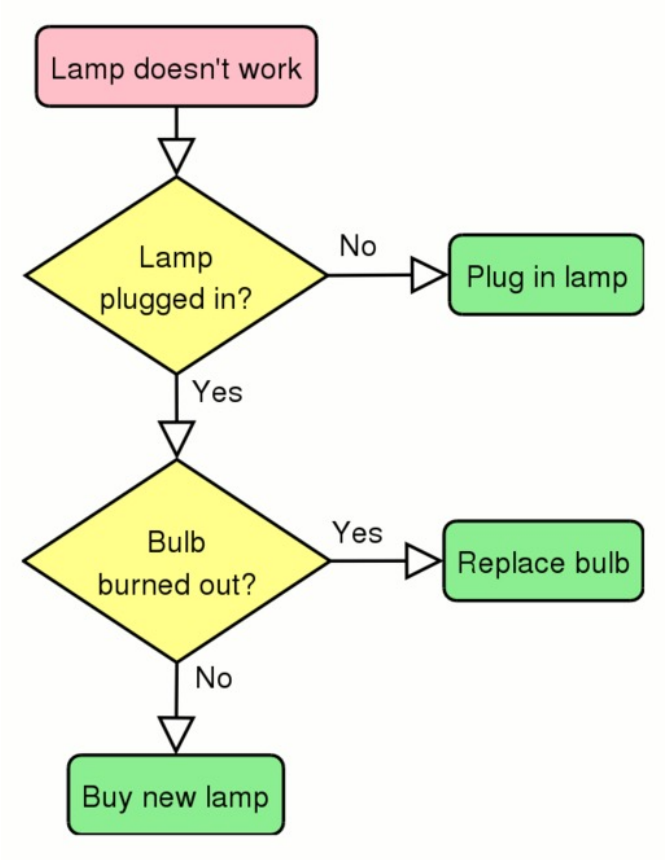
Instructor: Kranthi Varala

Control Flow

- Control flow is the path taken by the interpreter through your script.
- By default control flow is linear i.e., each statement in the script is executed in the order that it appears.
- We have looked at two ways to alter this flow before:
  - Branching via if/elif/else statements.
  - Loops via for/while statements.

Branching logic

- Used to implement alternate paths for the logic flow.



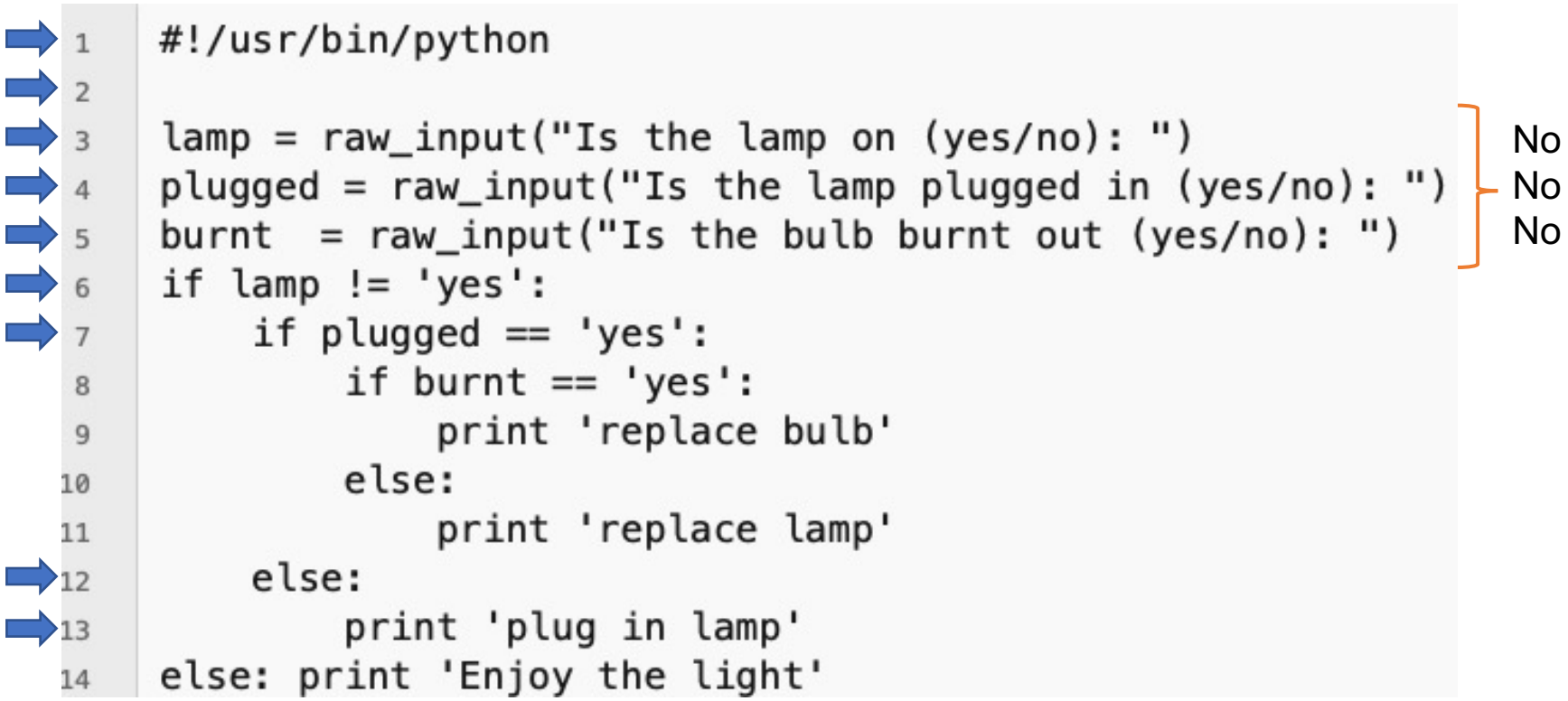
https://upload.wikimedia.org/wikipedia/commons/4/44/LampFlowchart.png

Lamp flowchart with if/else

```
1  #!/usr/bin/python
2
3  lamp = raw_input("Is the lamp on (yes/no): ")
4  plugged = raw_input("Is the lamp plugged in (yes/no): ")
5  burnt  = raw_input("Is the bulb burnt out (yes/no): ")
6  if lamp != 'yes':
7      if plugged == 'yes':
8          if burnt == 'yes':
9              print 'replace bulb'
10         else:
11             print 'replace lamp'
12     else:
13         print 'plug in lamp'
14 else: print 'Enjoy the light'
```

No
Yes
Yes

Lamp flowchart with if/else



Functions

- Functions are a set of statements designed to achieve a single task. For example, a function to calculate the average of a sequence of numbers.
- Functions are pieces of code that are outside the normal control flow of a program and need to be "called" by the main program.
- Functions are typically used to perform repeated tasks.
- Functions improve readability and reusability of code.

Examples of functions

- A function to read a file and retain only the specified columns in a 2-D list.
- A function to read FASTA files and store them in a dictionary.
- A function to fit a linear regression line through a set of observations.
  - A separate function to plot two sets of values as an X-Y scatter plot.
- A function that creates a JSON file from a data structure.

Role of Functions

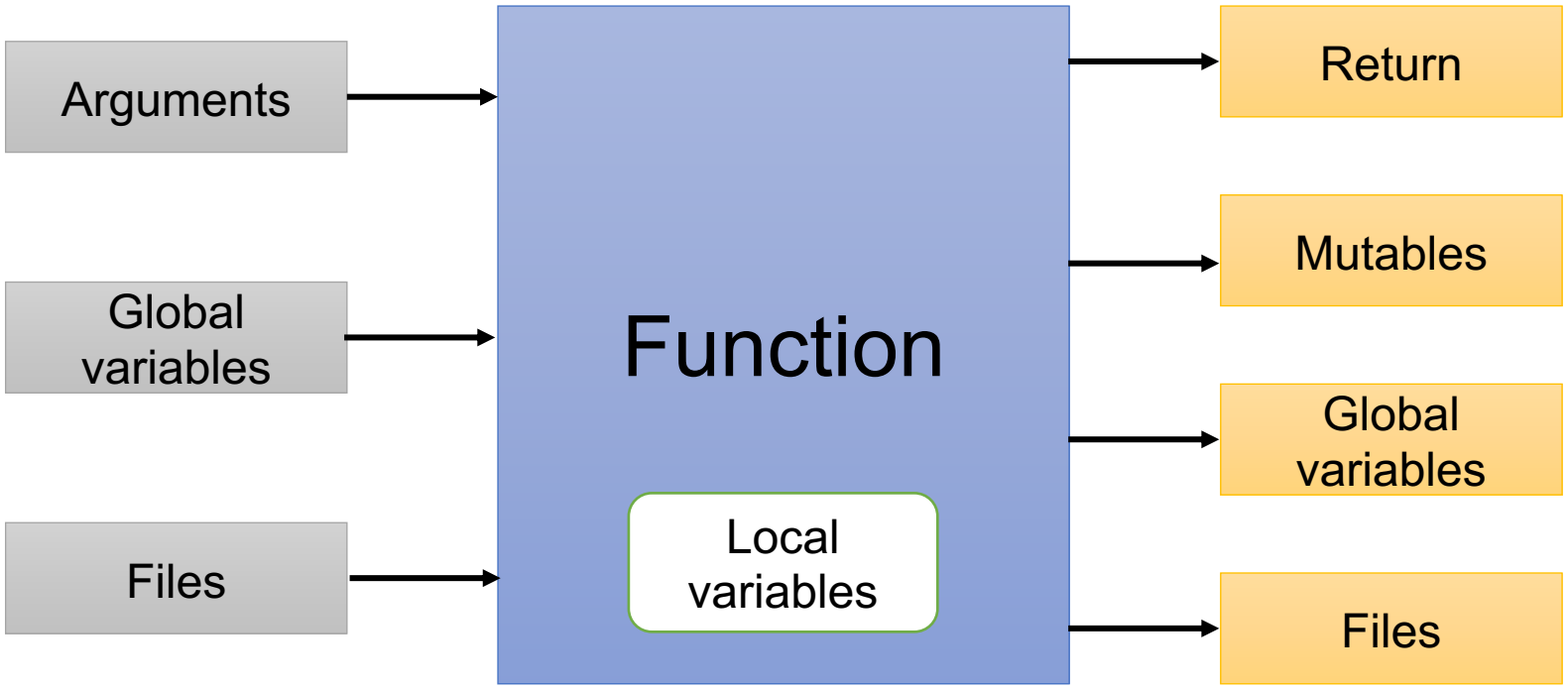


Image Credit: Learning Python by Mark Lutz

Structure of a function

- In the main program a function is called by its name:
  - `myFunction(a,b,c)`
- A function is defined using the keyword def:

```
def myFunction(x,y,z):
    Sum = x+y+z
    print("Sum is :" + Sum)
    return "Done."
```

- Note: Remember to define a function before calling it. This is because Python interpreter goes line-by-line and doesn’t know things that are not yet defined.

Creating a function : def vs. lambda

- `def` is a key word used to “define” a block of executable code. Code within the def block is not available to the interpreter until called.
- `def` creates a function object and assigns the object to the name of the function.
- `def` can appear as a separate code block in the same python file, or inside a loop or condition or even within another function (enclosed).
- `lambda` creates a function object as well, but it has no name. Instead it simply returns the result of the function.

Control flow: call and return

- When a function is called the flow of the main program stops and sends the "control" to the function.
- No code in the main program will be executed as long as the function has the control.
- Function "returns" the control to the main program using the `return` statement.
- Every function has an implicit `return` statement.
- We can use the `return` statement to send a value or object back to the main function.
- Since `return` can send back any object you can send multiple values by packing them into a tuple.
- If no `return` statement is given the function returns the `None` object.

Example of a function

```
#!/usr/bin/python

def mySum(a,b):
        sum = a + b
        return sum

a = 10
b = 20
c = mySum(a,b)
print ("Sum of %d and %d is %d"%(a,b,c));

kvarala@scholar-fe02:/scratch/scholar/k/kvarala/Week12 $ python functionExample.py
Sum of 10 and 20 is 30
```

Example of a function

```
#!/usr/bin/python

def mySum(x,y):
        sum = x + y
        return sum

a = 10
b = 20
c = mySum(a,b)
print ("Sum of %d and %d is %d"%(a,b,c));

kvarala@scholar-fe02:/scratch/scholar/k/kvarala/Week12 $ python functionExample.py
Sum of 10 and 20 is 30
```

Example of a reusable function

```
#!/usr/bin/python

def finder(x,y):
        if x in y:
                return "Found"

a = ('a','e','i','o','u')
b = ["Here","are","some","words"]
#a = (2,3,5,7,11)
#b = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
for w in b:
        for c in a:
                result = finder(c,w)
                if result: print (result + ' %s in %s'%(c,w))
```

```
Found e in Here
Found a in are
Found e in are
Found e in some
Found o in some
Found o in words
```

Example of a reusable function

```
#!/usr/bin/python

def finder(x,y):
        if x in y:
                return "Found"

#a = ('a','e','i','o','u')
#b = ["Here","are","some","words"]
a = (2,3,5,7,11)
b = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
for w in b:
        for c in a:
                result = finder(c,w)
                if result: print (result + ' %s in %s'%(c,w))
```

```
Found 2 in [1, 2, 3, 4]
Found 3 in [1, 2, 3, 4]
Found 5 in [5, 6, 7, 8]
Found 7 in [5, 6, 7, 8]
Found 11 in [9, 10, 11, 12]
```

Scope

- Namespace is the complete list of names, including all objects, functions, etc. that exist in a given context.
- The scope of an object is the namespace within which the object is available.
- A variable/object created in the main program is available all through the program i.e., global namespace
- A variable/object created within a function is only available within that function i.e, local namespace
- When a variable name is used the Python interpreter looks for that variable within the relevant scope first.

LEGB rule for Scope

- Python interpreter follows the LEGB rule for identifying the object by name.
- Local first: Look for this name first in the local namespace and use local version if available. If not, go to higher namespace.
- Enclosing second: If the current function is enclosed within another function, look for the name in that outer function. If not, go to higher namespace.
- Global third: Look for the name in the objects defined at global scale (e.g., main program).
- Built-in last: Finally, look for the variable among Python's built-in names.

Scope: Built-ins

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipe
Error', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError'
, 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'Envi
ronmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'Fut
ureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexErro
r', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryErr
or', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplemente
dError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupErro
r', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsync
Iteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', '
TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'U
nicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroD
ivisionError', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__pa
ckage__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'c
hr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', '
enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'ha
sattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'licens
e', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow',
 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted'
, 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Keeping track of Scope

```
#!/usr/bin/python

def mySum(x,y):
        sum = x + y # Sum is a local variable
        return sum

# Global variables
a = 10
b = 20

#c is also Global
c = mySum(a,b)

print ("Sum of %d and %d is %d"%(a,b,c));
```

- x and y are local variables that only exist in the scope of the function mySum
- BUT, mySum itself is a global function that exists in the global scope.

Output: `Sum of 10 and 20 is 30`

Keeping track of Scope

```
#!/usr/bin/python

def mySum(a,b):
        sum = a + b # Sum is a local variable
        a = 100
        return sum

# Global variables
a = 10
b = 20

#c is also Global
c = mySum(a,b)

print ("Sum of %d and %d is %d"%(a,b,c));
```

- a and b are local variables that only exist in the scope of the function mySum
- Changing the value of local 'a' does not affect global 'a'

Output: `Sum of 10 and 20 is 30`

Using global variables in functions

```
#!/usr/bin/python

def mySum(b):
        global a
        a = 100
        sum = a + b # Sum is a local variable
        return sum

# Global variables
a = 10
b = 20

#c is also Global
c = mySum(b)

print ("Sum of %d and %d is %d"%(a,b,c));
```

- Forces a to be pulled from the global space.
- Any changes made to the global variable here will affect the variable outside the function as well.

Output: `Sum of 100 and 20 is 120`

Arguments

- The objects sent to a function are called its arguments. Arguments are sometimes also called parameters.
- Passing an object as a argument to a function passes a reference to that object.
- Argument names in the def line of the function become new, local names.
- Arguments are passed in two ways:
  - 'Immutables' are passed by *value* eg., String, Integer, Tuple
  - 'Mutables' are passed by *reference* eg., Lists

Argument passing with mutables

```
def change(a,b):
        a = 'Smith'
        b[0] = 25
person = 'Bob'
ages = [5,15,35]
print "Before : " + person,ages
change(person,ages)
print "After : " + person,ages
```

Output:

```
Before : Bob [5, 15, 35]
After : Bob [25, 15, 35]
```

- person is a string (immutable) and thus passed by value to the local variable a.
- ages is a list (mutable) and thus passed to b as a reference.
- a and b are both local variables, but b is the same object as ages.

Arguments passed to function

```
def change(a,b):
        a = 'Smith'
        b[0] = 25
person = 'Bob'
ages = [5,15,35]
print "Before : " + person,ages
change(person,ages)
print "After : " + person,ages
```

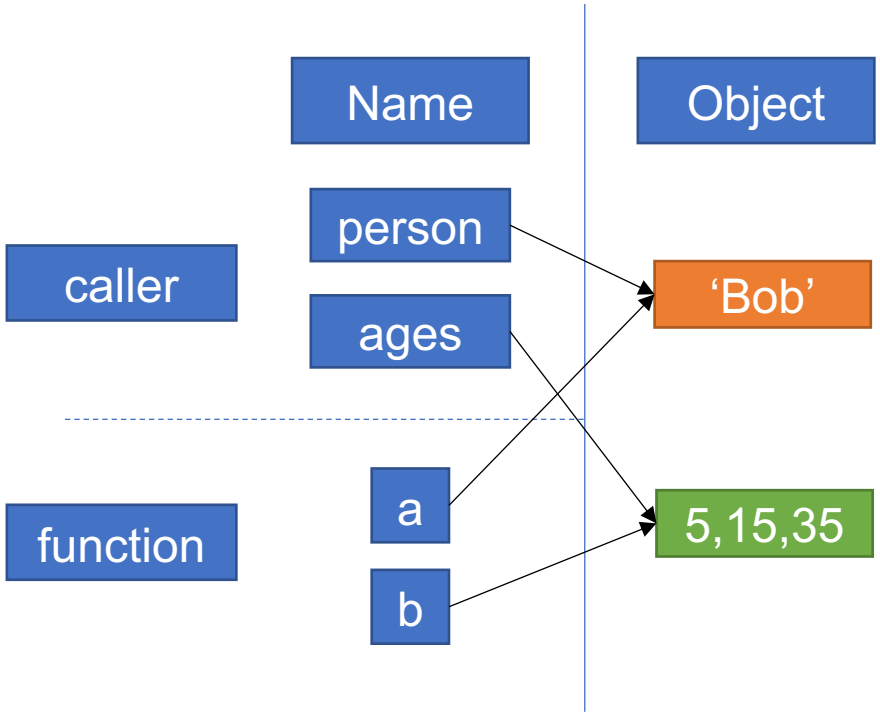


Image adapted from Learning Pyton by Mark Lutz

Control returned to main

```
def change(a,b):
        a = 'Smith'
        b[0] = 25
person = 'Bob'
ages = [5,15,35]
print "Before : " + person,ages
change(person,ages)
print "After : " + person,ages
```

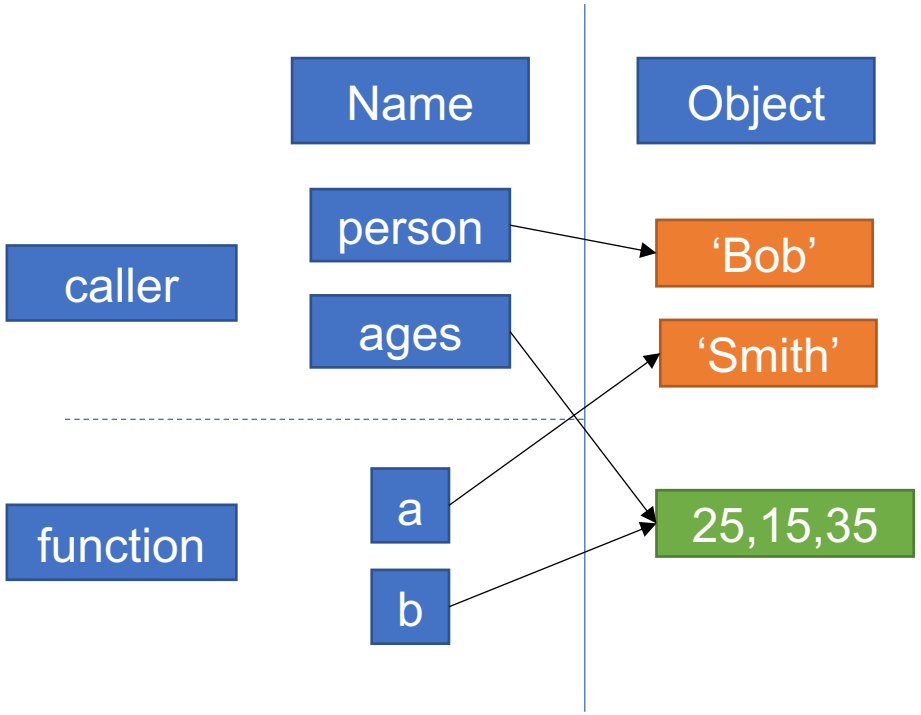


Image adapted from Learning Pyton by Mark Lutz

Maintaining integrity of mutables

- Integrity of lists and other mutables can be maintained by passing an explicit copy.
- For example:

```
change(person,ages[:])
```

Sends a copy of the list ages

```
change(person,ages)
```

Sends a reference to the list ages

Summary: Using functions in your project

- Functions allow separation of code into logical units to improve code readability.
- Functions allow reusing code i.e., ‘cut and paste’ to move code across your scripts. Eg., a function to read FASTA files into a dictionary can be reused across any script that needs to read FASTA files.
- Limits on scope of variables allows reusing variable names while still maintaining data integrity.
- Beware of passing mutable objects as arguments.