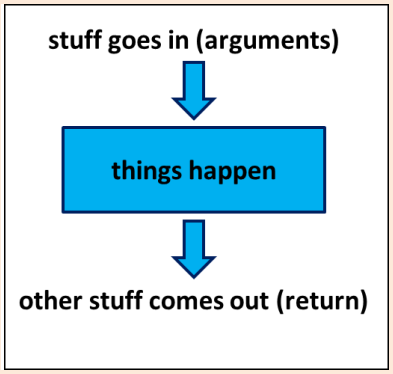
More on Functions, Modules

Genome 559: Introduction to Statistical and Computational Genomics

Elhanan Borenstein

A quick review

- **Functions**:
  - Reusable pieces of code (write once, use many)
  - Take arguments, "do stuff", and (usually) return a value
  - Use to organize & clarify your code, reduce code duplication



- Defining a function:

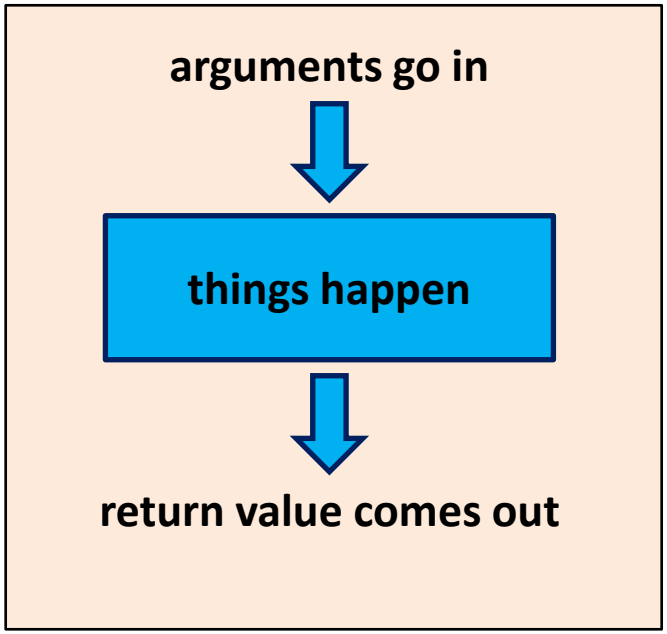
```
def <function_name>(<arguments>):
     <function code block>
     <usually return something>
```

- Using (calling) a function:

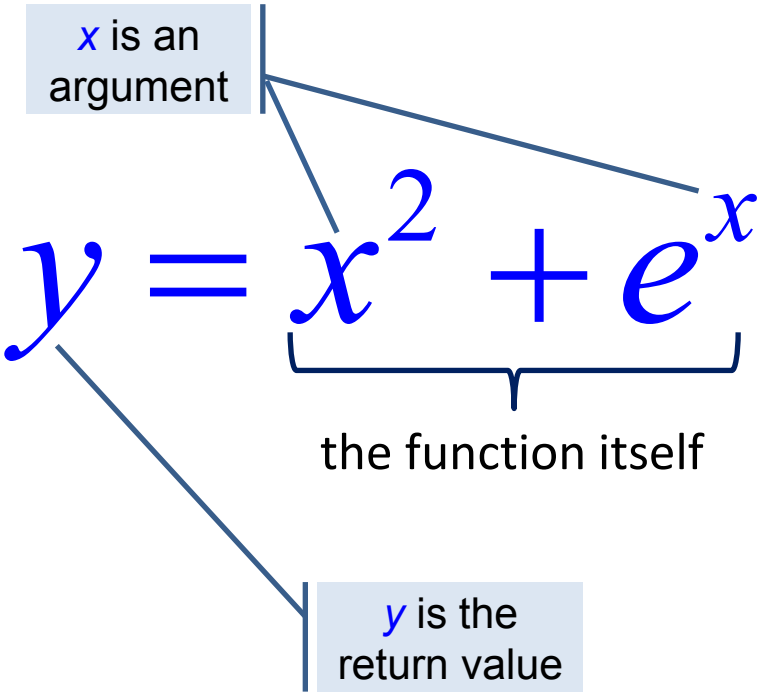
```
<function defined here>

<my_variable> = function_name(<my_arguments>)
```

A close analogy is the mathematical function

A Python Function



A mathematical Function



A quick example

```
import sys

def makeDict(fileName):
    myFile = open(fileName, "r")
    myDict = {}
    for line in myFile:
        fields = line.strip().split("\t")
        myDict[fields[0]] = float(fields[1])
    myFile.close()
    return myDict

FirstFileName = sys.argv[1]
FirstDict = makeDict(FirstFileName)

SecondFileName = sys.argv[2]
SecondDict = makeDict(SecondFileName)

…

FlyGenesDict = makeDict("FlyGeneAtlas.txt")
```

Write once

Use many times

A note about namespace

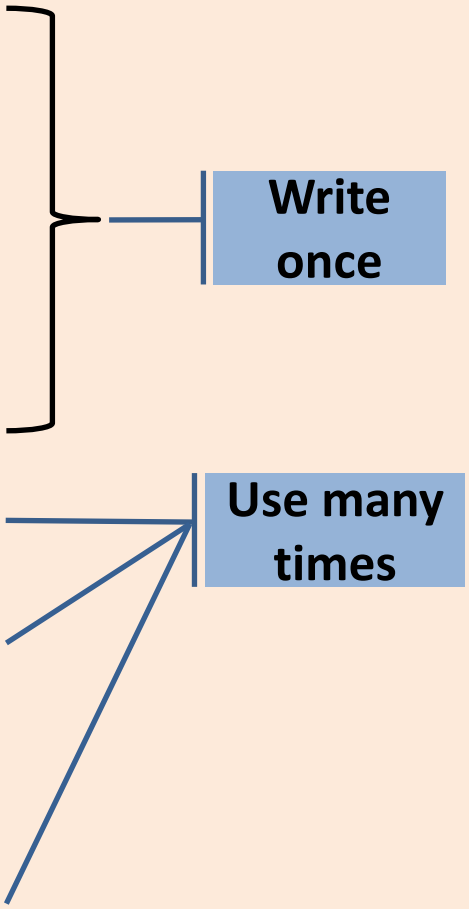
```
import sys

def makeDict(fileName):
    myFile = open(fileName, "r")
    myDict = {}
    for line in myFile:
        fields = line.strip().split("\t")
        myDict[fields[0]] = float(fields[1])
    myFile.close()
    return myDict

FirstFileName = sys.argv[1]
FirstDict = makeDict(FirstFileName)

SecondFileName = sys.argv[2]
SecondDict = makeDict(SecondFileName)

…

FlyGenesDict = makeDict("FlyGeneAtlas.txt")
```

Write once

Use many times

A note about namespace

```
import sys

def makeDict(fileName):
    myFile = open(fileName, "r")
    myDict = {}
    for line in myFile:
        fields = line.strip().split("\t")
        myDict[fields[0]] = float(fields[1])
    myFile.close()
    return myDict

FirstFileName = sys.argv[1]
FirstDict = makeDict(FirstFileName)

SecondFileName = sys.argv[2]
SecondDict = makeDict(SecondFileName)

…

FlyGenesDict = makeDict("FlyGeneAtlas.txt")
```



Returning values

- Check the following function:

```
# This function …
# …
def CalcSum(a_list):
    sum = 0
    for item in a_list:
        sum += item
    return sum
```

- What does this function do?

Returning values

- Check the following function:

```
# This function calculates the sum
# of all the elements in a list
def CalcSum(a_list):
    sum = 0
    for item in a_list:
        sum += item
    return sum
```

- What does this function do?

```
>>> my_list = [1, 3, 2, 9]
>>> print CalcSum(my_list)
15
```

Returning more than one value

- Let's be more ambitious:

```
# This function calculates the sum
# AND the product of all the
# elements in a list
def CalcSumAndProd(a_list):
    sum = 0
    prod = 1
    for item in a_list:
        sum += item
        prod *= item
    return ???
```

- How can we return both values?

Returning more than one value

- We can use a list as a return value:

```
# This function calculates the sum
# AND the product of all the
# elements in a list
def CalcSumAndProd(a_list):
    sum = 0
    prod = 1
    for item in a_list:
        sum += item
        prod *= item
    return [sum, prod]
```

```
>>> my_list = [1, 3, 2, 9]
>>> print CalcSumAndProd(my_list)
[15, 54]
```

```
>>> res = CalcSumAndProd(my_list)
```

List assignment

```
>>> [s,p] = CalcSumAndProd(my_list)
```

multiple assignment

Returning lists

- Recall the increment function:

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    new_list = []
    for item in a_list:
        new_list.append(item+1)
    return new_list

# Now, create a list and use the function
my_list = [1, 20, 34, 8]
print my_list
my_list = incrementEachElement(my_list)
Print my_list
```

```
[1, 20, 34, 8]
[2, 21, 35, 9]
```

- Is this good practice?

Returning lists

- What will happen if we do this?

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    for index in range(len(a_list)):
        a_list[index] +=1

# Now, create a list and use the function
my_list = [1, 20, 34, 8]
incrementEachElement(my_list)
print my_list
```

- **(note: no return value!!!)**

Returning lists

- What will happen if we do this?

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    for index in range(len(a_list)):
        a_list[index] +=1

# Now, create a list and use the function
my_list = [1, 20, 34, 8]
incrementEachElement(my_list)
print my_list
```

- **(note: no return value)**

```
[2, 21, 35, 9]
```

WHY IS THIS WORKING?

Pass-by-reference vs. pass-by-value

- Two fundamentally different function calling strategies:
- **Pass-by-Value:**
  - The value of the argument is copied into a local variable inside the function
  - C, Scheme, C++
- **Pass-by-reference:**
  - The function receives an implicit reference to the variable used as argument, rather than a copy of its value
  - Perl, VB, C++
- **So, how does Python pass arguments?**

Python passes arguments by reference

(almost)

- So … this will work!

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    for index in range(len(a_list)):
        a_list[index] +=1
```

```
>>> my_list = [1, 20, 34, 8]
>>> incrementEachElement(my_list)
>>> my_list
[2, 21, 35, 9]
>>> incrementEachElement(my_list)
>>> my_list
[3, 22, 36, 10]
```

Python passes arguments by reference

(almost)

- How about this?

```
def addQuestionMark(word):
    print "word inside function (1):", word
    word = word + "?"
    print "word inside function (2):", word

my_word = "really"
addQuestionMark(my_word)
print "word after function:", my_word
```

Python passes arguments by reference

(almost)

- How about this?

```
def addQuestionMark(word):
    print "word inside function (1):", word
    word = word + "?"
    print "word inside function (2):", word

my_word = "really"
addQuestionMark(my_word)
print "word after function:", my_word
```

```
word inside function (1): really
word inside function (2): really?
word after function: really
```

- Remember:
  1. Strings/numbers are immutable
  2. The assignment command often creates a new object

Passing by reference: the bottom line

- **You can (and should) use this option when**:
  - Handling large data structures
  - “In place” changes make sense

- **Be careful** (a double-edged sword):
  - Don’t lose the reference!
  - Don’t change an argument by mistake

- When we learn about objects and methods we will see yet an additional way to change variables

Required Arguments

- How about this?

```
def printMulti(text, n):
    for i in range(n):
        print text
```

```
>>> printMulti(“Bla”,4)
Bla
Bla
Bla
Bla
```

- What happens if I try to do this:

```
>>> printMulti("Bla")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: printMulti() takes exactly 2
arguments (1 given)
```

Default Arguments

- Python allows you to define defaults for various arguments:

```
def printMulti(text, n=3):
    for i in range(n):
        print text
```

```
>>> printMulti("Bla",4)
Bla
Bla
Bla
Bla
```

```
>>> printMulti("Yada")
Yada
Yada
Yada
```

Default Arguments

- This is very useful if you have functions with numerous arguments/parameters, most of which will rarely be changed by the user:

```
def runBlast(fasta_file, costGap=10, E=10.0, desc=100,
   max_align=25, matrix="BLOSUM62", sim=0.7, corr=True):
   <runBlast code here>
```

- You can now simply use:

```
>>> runBlast("my_fasta.txt")
```

- Instead of:

```
>>> runBlast("my_fasta.txt",10,10.0,100,25,"BLOSUM62",0.7,
True)
```

Keyword Arguments

- You can still provide values for specific arguments using their label:

```
def runBlast(fasta_file, costGap=10, E=10.0, desc=100,
  max_align=25, matrix="BLOSUM62", sim=0.7, corr=True):
    <runBlast code here>
    …


>>> runBlast("my_fasta.txt", matrix="PAM40")
```

Modules

Modules

- Recall your makeDict function:

```
def makeDict(fileName):
    myFile = open(fileName, "r")
    myDict = {}
    for line in myFile:
        fields = line.strip().split("\t")
        myDict[fields[0]] = float(fields[1])
    myFile.close()
    return myDict
```

- This is in fact a very useful function which you may want to use in many programs!
- So are other functions you wrote (e.g., makeMatrix)

Modules

- A module is a file that contains a collection of **related** functions.
- You have already used several built-in modules:
  - e.g.: sys, math
- Python has numerous standard modules
  - Python Standard Library: (http://docs.python.org/library/)

- **It is easy to create and use your own modules:**
  - **JUST PUT YOUR FUNCTIONS IN A SEPARATE FILE!**

Importing Modules

- To use a module, you first have to import it into your namespace
- To import the entire module:

```
import module_name
```

my_prog.py

```
import utils
import sys

Dict1 = utils.makeDict(sys.argv[1])
Dict2 = utils.makeDict(sys.argv[2])

Mtrx = utils.makeMatrix("blsm.txt")

…
```

utils.py

```
# This function makes a dictionary
def makeDict(fileName):
   myFile = open(fileName, "r")
   myDict = {}
   for line in myFile:
      fields = line.strip().split("\t")
      myDict[fields[0]] = float(fields[1])
   myFile.close()
   return myDict

# This function reads a 2D matrix
def makeMatrix(fileName):
    < ... >
```

The dot notation

- Why did we use `utils.makeDict()` instead of just `makeDict()`?

- Dot notation allows the Python interpreter to organize and divide the namespace

Sample problem #1

- Write a function that calculates the first n elements of the Fibonacci sequence.
  - Reminder: In the Fibonacci sequence of numbers, each number is the sum of the previous two numbers, starting with 0 and 1. This sequence begins: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, …
- The function should return these n elements as a list

Solution #1

```
# Calculate Fibonacci series up to n
def fibonacci(n):
    fib_seq = [0, 1];
    for i in range(2,n):
        fib_seq.append(fib_seq[i-1] + fib_seq[i-2])

    return fib_seq[0:n]     # Why not just fib_seq?

print fibonacci(10)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

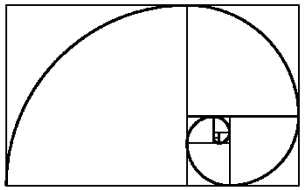
Sample problem #2

- Make the following improvements to your function:

1. Add two **optional** arguments that will denote alternative starting values (instead of 0 and 1).
   - fibonacci(10) → [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
   - fibonacci(10,4) → [4, 1, 5, 6, 11, 17, 28, 45, 73, 118]
   - fibonacci(10,4,7) →[4, 7, 11, 18, 29, 47, 76, 123, 199, 322]

2. Return, in addition to the sequence, also the ratio of the last two elements you calculated (how would you return it?).

3. Create a module “my_math” and include your function in this module. Import this module into another program and use the function.



Solution #2

my_math.py

```
# Calculate Fibonacci series up to n
def fibonacci(n, start1=0, start2=1):
    fib_seq = [start1, start2];
    for i in range(2,n):
        fib_seq.append(fib_seq[i-1]+fib_seq[i-2])

    ratio = float(fib_seq[n-1])/float(fib_seq[n-2])
    return [fib_seq[0:n], ratio]
```

my_prog.py

```
import my_math
seq, ratio = my_math.fibonacci(1000)
print "first 10 elements:",seq[0:10]
print "ratio:", ratio
# Will print:
# first 10 elements: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
# ratio: 1.61803398875

fib = my_math.fibonacci # creating a local name
print fib(5,12,14)
# Will print:
# [[12, 14, 26, 40, 66], 1.65]
```

Challenge problem

- Write your own sort function!
- Sort elements in ascending order.
- The function should sort the input list **in-place** (i.e. do not return a new sorted list as a return value; the list that is passed to the function should itself be sorted after the function is called).
- As a return value, the function should return the number of elements that were in their appropriate ("sorted") location in the original list.
- You can use any sorting algorithm. Don't worry about efficiency right now.

Challenge solution 1

```
def swap(a_list, k, l):
        temp = a_list[k]
        a_list[k] = a_list[l]
        a_list[l] = temp

def bubbleSort(a_list):
    n = len(a_list)
    a_list_copy = [] # note: why don't we use assignment
    for item in a_list: a_list_copy.append(item)

    # bubble sort
    for i in range(n):
        for j in range(n-1):
            if a_list[j] > a_list[j+1]:
                swap(a_list, j, j+1) # note: in place swapping

    # check how many are in the right place
    count = 0
    for i in range(n):
        if a_list[i] == a_list_copy[i]: count += 1
    return count
```

This is the actual sorting algorithm. Simple!

```
>>> ls = [1, 3, 2, 15, 7, 4, 8, 12]
>>> print bubbleSort(ls)
2
>>> print ls
[1, 2, 3, 4, 7, 8, 12, 15]
```

Alternative challenge solution 1

```
def swap(a_list, k, l):
        temp = a_list[k]
        a_list[k] = a_list[l]
        a_list[l] = temp

def bubbleSort(a_list):
    n = len(a_list)
    a_list_copy = [] # note: why don't we use assignment
    for item in a_list: a_list_copy.append(item)

    # bubble sort
    for i in range(n):
        for j in range(n-1-i):
            if a_list[j] > a_list[j+1]:
                swap(a_list, j, j+1) # note: in place swapping

    # check how many are in the right place
    count = 0
    for i in range(n):
        if a_list[i] == a_list_copy[i]: count += 1
    return count
```

Why is this better?
Why is this working?

```
>>> ls = [1, 3, 2, 15, 7, 4, 8, 12]
>>> print bubbleSort(ls)
2
>>> print ls
[1, 2, 3, 4, 7, 8, 12, 15]
```

FYI: Variable-length arguments

- You may want to allow a function to process more arguments than you specified while defining it (can you think of an example?)
- These arguments are called variable-length arguments and are not named in the function definition.

```
def printinfo( arg1, *vartuple ):
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Importing functions

- To import specific functions:
  ```
  from module_name import function_name
  ```

- To import all the functions in a module:
  ```
  from module_name import *
  ```

- If functions (rather than modules) are imported, there is no need to use the dot notation

- Be careful when using the import * command. It can easily lead to namespace conflicts.