Python 101
Functions

DIOGO SILVA

Day overview:

Functions

1. Purpose
2. The basic recipe and calling a function
3. Arguments
4. Variable scopes
5. Returning values from a function
6. Lambda (anonymous) function

I/O Input output

1. Input from user/keyboard
2. Reading files
3. Writing files
4. Closing files

Purpose

Functions (http://docs.python.org/tutorial/controlflow.html#defining-functions) - pieces of code that are written one time and reused as much as desired within the program. They:

- Are the simplest callable object in python
- Perfom single related actions that can handle repetitive tasks
- Significantly reduce code redundancy and complexity, while providing a clean structure
- Decompose complex problems into simpler pieces

Purpose

- Supose you have a protein sequence and want to find out the frequency of the "**W**" amino acid and all its positions in the sequence.

```
aa_sequence = "mgagkvikckaafwagkplwegevappkakapca"
position_list = []
sequence_length = float(len(aa_sequence))
for i in range (sequence_length):
    if aa_sequence[i] == "w":
        position_list.append(str(i))

p_count = float(aa_sequence.count("w"))
p_frequency = p_count/sequence_length
print "The aa 'w' has a frequency of %s and is found in the following sites: %s" % (p_frequency," ".join(position_list))
```

```
The aa 'w' has a frequency of 0.058823529411764705 and is
found in the following sites: 13 19
```

Purpose

- Now you may want to know the same information about, say "**P**". You would need to re-write your entire code again for "**P**"...

```
aa_sequence = "mgagkvikckaafwagkplwegevappkakapca"
position_list = []
sequence_length = float(len(aa_sequence))
for i in range (sequence_length):
    if aa_sequence[i] == "p":
        position_list.append(str(i))

p_count = float(aa_sequence.count("p"))
p_frequency = p_count/sequence_length
print "The aa 'p' has a frequency of %s and is found in the following sites: %s" % (p_frequency," ".join(position_list))
```

```
The aa 'p' has a frequency of 0.11764705882352941 and is
found in the following sites: 17 25 26 31
```

And 19 more times to accomodate all other amino acids!!

Purpose

- Using a function, the problem can be easily solved like this:

```
aa_sequence = "mgagkvikckaafwagkplwegevappkakapca"
def aa_statistics(sequence,aa):
    sequence_length,aa_positions = len(sequence),[]
    aa_frequency = (lambda
count,length:float(count)/float(length))
    for i in range (sequence_length):
        if sequence[i] == aa:
            aa_positions.append(str(i))
    print (aa_frequency(sequence.count(aa),sequence_length),aa_positions)
```

With only 7 lines of code, we are now able to provide the required information for all amino acids and for any input sequence.

The basic recipe

- The basic steps when defining a function:

```
def name ():
    "Documentation string of the function"
    [statements]

```

1. "def" - Functions must start with the "**def**" keyword.
2. "**name**" - The name of the function must not contain special characters or whitespaces

3. "**()**" - Parenthesis enclose input parameters or arguments
4. ":" - The code block within every function starts with a **colon** and is **indented**
5. Documentation [optional] - It is good practice to document your function
6. "statements" - The actual code block of your function

Function calling

- After a function is defined, it represents nothing more than an idle piece of code, unless called. It is only when we call a function that the statements inside the function body are executed.

```python
def print_me ():
    "This function prints something"
    print "Hello World"

```

Arguments

A function can be created without arguments,

```python
def print_me():
    "Example of a simple function without arguments"
    print "Hello World"

print_me()

```

```
Hello World
```

or using the following types of arguments:

- **Required arguments**
- **Default arguments**
- **Variable length arguments**

Arguments

Required arguments

```python
def aa_frequency (sequence,aa):
    "This function takes exactly two arguments"
    sequence_length = len(sequence)
    aa_frequency = 
float(sequence.count(aa))/float(sequence_length)
    print aa_frequency
```

- When calling for a function with required arguments, the **exact** same number of arguments must be specified, no more and no less.

```python
def aa_frequency (sequence,aa): #folded
aa_frequency ("AWKLCVPAMAKNENAW","K")

```

```
0.125
```

Arguments

Required arguments

- It is also possible to provide previously named variables as arguments

```python
H_sapiens_aa = "AWKLCVPAMAKNENAW"
def aa_frequency (sequence,aa): #folded
aa_frequency (H_sapiens_aa,"K")

```

```
0.125
```

- If you specify a different number of arguments, however

```python
H_sapiens_aa = "AWKLCVPAMAKNENAW"
def aa_frequency (sequence,aa): #folded
```

```
aa_frequency (H_sapiens_aa,"K","G")
```

```
TypeError: aa_statistics() takes exactly 2 arguments (3 given)
```

Arguments

Variable length arguments

- Placing an asterisk (*) before the variable name will store the arguments in a tuple (http://docs.python.org/tutorial/datastructures.html#tuples-and-sequences)

```
def concatenate (*sequences):
    " This one can take a variable number of arguments, even 0"
    concatenated_sequences = ""
    for i in sequences: # You can iterate over the tuple,
        concatenated_sequences += i
    if len(sequences) >= 2:
        first_sequences = sequences[:2] # and slice its items
        print concatenated_sequences, first_sequences

concatenate("GTCCG","AGTCG","AGTAG","AGTGA")
concatenate() # In this case the tuple "sequences" is empty
```

```
GTCCGAGTCGAGTAGAGTGA ('GTCCG', 'AGTCG')
```

Arguments

Default arguments

- Arguments can also have default values, by assigning those values to the argument keyword with the assign ("=") symbol.

```
def codon_count (Sequence,
StopCodon="TAA",StartCodon="ATG"):
    stop_count = Sequence.count(StopCodon)
    start_count = Sequence.count(StartCodon)
    print stop_count, start_count
```

- The function will assume the default value if the argument keyword is not specified when calling the function.

```
def codon_count (Sequence,
StopCodon="TAA",StartCodon="ATG"): #folded

H_sapiens =
"AGCTAGTCGTAGCATGATTAACGTAGGCTATACTACTAAATGRC"
codon_count (H_sapiens)
```

```
2 2
```

Arguments

Using argument keywords

- When calling a function, the order of the arguments can be changed by using the argument's keyword and the assign ("=") symbol.

```
H_sapiens =
"AGCTAGTCGTAGCATGATTAACGTAGGCTATACTACTAAATGRC"
def codon_count (Sequence,
StopCodon="TAA",StartCodon="ATG"): #folded

codon_count (StopCodon="UAG", Sequence=H_sapiens)
```

```
0 2
```

- Note that this is necessary if you would like to change only the second default argument, and leave the first with the default value

```
H_sapiens =
"AGCTAGTCGTAGCATGATTAACGTAGGCTATACTACTAAATGRC"
def codon_count (Sequence,
StopCodon="TAA",StartCodon="ATG"): #folded
```

```
codon_count (H_sapiens,StartCodon="ATT")
```

```
2 1
```

Arguments

Considerations when combining different argument types

- **Default** arguments should come after **required** arguments

```
def name (required,required,
(...),default=value,default=value,(...)):
    [...code block...]
```

- **Variable length** arguments should be used only once and be always last. There is also no point in using them with **default** arguments.

```
def name (required,required,(...),*varible_length):
    [...code block...]
```

Namespaces or scope of variables

When writting a program, it is extremely important to know the difference between the **local** and **global** scope of the variables

Glogal variables

- Variables defined outside functions or other objects (i.e., classes) are **global** variables - they are accessible throughout most of the program, even by functions.

```
sequence = "ACGTGTGC"
def print_me():
    print sequence
```

```
print_me()

```

```
ACGTGTGC
```

- To change the contents of a **global** variable in a function, we can use the global keyword

```
sequence = "ACGTGTGC"
def print_me():
    global sequence
    sequence = "TTTTTTT"
    print sequence

print_me()
print sequence # Because of the global keyword, the
global variable was changed
```

```
TTTTTTT
TTTTTTT
```

Namespaces or scope of variables

Local variables

- By default, all variables defined inside a function (including argument keywords) are **local** variables - they are not accessible by the whole program, only within the function where they are declared.

```
def print_me():
    sequence = "ACGTGA"
    print sequence

print sequence

```

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

- Note that without the global keyword, global variables are overwritten by local variables with the same name defined in a function

```
sequence = "TTTTT"
def print_me():
    sequence = "AAAAAA"
    print sequence

print_me()

```

```
AAAAAA
```

Return

The ***return*** keyword is used to return values from a function, which can then be assigned to new variables that are accessible to the whole program

```
H_sapiens_lc1 =
"AGCTAGTCGTAGCATGATTAACGTAGGCTATACTACTAAATGRC"
H_sapiens_lc2 =
"CGTAGTCGTAGTTTGCAGTGCGCTGATCGTAGTCGATGCTGTGT"

def concatenate (*sequences):
    concatenated_sequence = ""
    for i in sequences:
        concatenated_sequence += i
    return concatenated_sequence

new_sequence = concatenate(H_sapiens_lc1,H_sapiens_lc2)
    # And now we can use the output of a function, as
the input of another

def codon_count (Sequence,
StopCodon="TAA",StartCodon="ATG"):
    stop_count = Sequence.count(StopCodon)
    start_count = Sequence.count(StartCodon)
    print stop_count, start_count

codon_count (new_sequence)
```

```
2 3
```

Return

Returning multiple values

- Functions can return multiple values

```
def codon_count (Sequence, StopCodon="TAA",StartCodon="ATG"):
    stop_count = Sequence.count(StopCodon)
    start_count = Sequence.count(StartCodon)
    return stop_count, start_count # Returns a tuple with two items
    # OR
    # return [stop_count, start_count] -> Returns a list with two items
```

- And these values can be assigned to multiple variables

```
H_sapiens = "AGCTAGTCGTAGCATGATTAACGTAGGCTATACTACTAAATGRC"
def codon_count (Sequence, StopCodon="TAA",StartCodon="ATG"): #folded

stop,start = codon_count(H_sapiens)
print stop,start

start = codon_count(H_sapiens)[1] # You can even select the variable(s) you want
print start
```

```
2 2
2
```

Return

Functions always return something

If a function does not contain the return keyword, it will return *None*

```
def print_me():
    a = 2+2

print_me() == None

```

```
True
```

Lambda (anonymous) functions

Lambda (http://docs.python.org/tutorial/controlflow.html#lambda-forms) is an anonymous (unnamed) function that is used primarily to write very short functions that are a hassle to define in the normal way. Where a regular function would do:

```
def add(a,b):
    print a+b

add(4,3)

```

```
7
```

a lambda function:

```
print (lambda a,b: a+b)(4,3)

```

```
7
```

The lambda function can be used elegantly with other functional parts of the Python language, like map() (http://docs.python.org/library/functions.html#map). In this example we can use it to convert a list of RNA sequences into DNA sequences:

```
RNA = ["AUGAUU","AAUCGAUCG","ACUAUG","ACUAUG"]
DNA = map(lambda sequence: sequence.replace("U","T"),
RNA)
```

```
print DNA
```

```
["ATGATT","AATCGATCG","ACTATG","ACTATG"]
```

Wrap up

So, we have covered thus far:

- How to define functions using the ***def*** keyword
- How to call a function
- The three main types of arguments a function can take: Required , variable length and arguments
- The local and global scope of variables
- The usage of the ***return*** keyword to return values from functions
- Lambda functions