Functions

Built-in functions

You've used several functions already

```
>>> len("ATGGTCA")
7
>>> abs(-6)
6
>>> float("3.1415")
3.1415000000000002
>>>
```

What are functions?

A function is a code block with a name

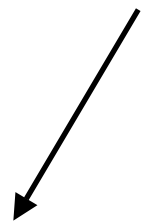
```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

Functions start with ‘def’

```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

Then the name

This function is named ‘hello’



```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

The list of parameters

The parameters are always listed in parenthesis.
There are no parameters in this function
so the parameter list is empty.

```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

(I'll cover parameters in more detail soon)

A colon

A function definition starts a new code block.
The definition line must end with a colon (the ":")
Just like the 'if', and 'for' statements.

```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

The code block

These are the statements that are run when the function is called. They can be any Python statement (print, assignment, if, for, open, ...)

```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

Calling the function

When you "call" a function you ask Python to execute the statements in the code block for that function.

```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

Which function to call?

Start with the name of the function.
In this case the name is "hello"

```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

List any parameters

The parameters are always listed in parenthesis.
There are no parameters for this function
so the parameter list is empty.

```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

And the function runs

```
>>> def hello():
...     print "Hello, how are you?"
...
>>> hello()
Hello, how are you?
>>>
```

Arguments and Parameters

(Two sides of the same idea)

Most of the time you don't want the function to do the same thing over and over. You want it to run the same algorithm using different data.

Hello, <insert name here>

Say “Hello” followed by the person’s name

In maths we say “the function is *parameter*ized by the person’s name”

```
>>> def hello(name):
...     print "Hello", name
...
>>> hello("Andrew")
Hello Andrew
>>>
```

Change the function definition

The function now takes one parameter. When the function is called this parameter will be accessible using the variable named `name`

```
>>> def hello(name):
...     print "Hello", name
...
>>> hello("Andrew")
Hello Andrew
>>>
```

Calling the function

The function call now needs one argument.
Here I'll use the string "Andrew".

```
>>> def hello(name):
...     print "Hello", name
...
>>> hello("Andrew")
Hello Andrew
>>>
```

And the function runs

The function call assigns the string “Andrew” to the variable “name” then does the statements in the code block

```
>>> def hello(name):
...     print "Hello", name
...
>>> hello("Andrew")
Hello Andrew
>>>
```

Multiple parameters

Here's a function which takes two parameters and subtracts the second from the first.

Two parameters in the definition

```
>>> def subtract(x, y):
...     print x-y
...
>>> subtract(8, 5)
3
>>>
```

Two parameters in the call

Returning values

Rarely do functions only print.

More often the function does something and the results of that are used by something else.

For example, `len` computes the length of a string or list then returns that value to the caller.

subtract doesn't return anything

By default, a function returns the special value `None`

```
>>> def subtract(x, y):
...   print x-y
...
>>> x = subtract(8, 5)
3
>>> print x
None
>>>
```

The `return` statement

The return statement tells Python to exit the function and return a given object.

```
>>> def subtract(x, y):
...     return x-y
...
>>> x = subtract(8, 5)
>>> print x
3
>>>
```

You can return anything (list, string, number, dictionary, even a function).

Making a function

Yes, we're going to count letters again.

A solution yesterday's #1 (except for the raw_input)

```
seq = "ATGCATGATGCATGAAAGGTCG"
counts = {}
for base in seq:
  if base not in counts:
    counts[base] = 1
  else:
    counts[base] = counts[base] + 1

for base in counts:
  print base, "=", counts[base]
```

Identify the function

I'm going to make a function which counts bases.
What's the best part to turn into a function?

```
seq = "ATGCATGATGCATGAAAGGTCG"
counts = {}
for base in seq:
  if base not in counts:
    counts[base] = 1
  else:
    counts[base] = counts[base] + 1

for base in counts:
  print base, "=", counts[base]
```

Identify the input

In this example the sequence can change.
That makes `seq` a good choice as a parameter.

```
seq = "ATGCATGATGCATGAAAGGTCG"
counts = {}
for base in seq:
  if base not in counts:
    counts[base] = 1
  else:
    counts[base] = counts[base] + 1

for base in counts:
  print base, “=”, counts[base]
```

Identify the algorithm

This is the part of your program
which does something.

```
seq = "ATGCATGATGCATGAAAGGTCG"
counts = {}
for base in seq:
  if base not in counts:
    counts[base] = 1
  else:
    counts[base] = counts[base] + 1

for base in counts:
  print base, “=”, counts[base]
```

Identify the output

The output will use the data computed by your function...

```
seq = "ATGCATGATGCATGAAAGGTCG"
counts = {}
for base in seq:
  if base not in counts:
    counts[base] = 1
  else:
    counts[base] = counts[base] + 1

for base in counts:
  print base, “=”, counts[base]
```

Identify the return value

... which helps you identify the return value

```
seq = "ATGCATGATGCATGAAAGGTCG"
counts = {}
for base in seq:
  if base not in counts:
    counts[base] = 1
  else:
    counts[base] = counts[base] + 1

for base in counts:
  print base, “=”, counts[base]
```

Name the function

First, come up with a good name for your function.

It should be descriptive so that when you or someone else sees the name then they have an idea of what it does.

Good names

count_bases
count_letters
countbases

Bad names

do_count
count_bases_in_sequence
CoUnTbAsEs
QPXT

Start with the ‘def’ line

The function definition starts with a ‘`def`’

```
def count_bases(seq):
```

It is named ‘`count_bases`’

It takes one parameter, which will be accessed using the variable named ‘`seq`’

Remember, the def line ends with a colon

Add the code block

```
def count_bases(seq):
    counts = {}
    for base in seq:
        if base not in counts:
            counts[base] = 1
        else:
            counts[base] = counts[base] + 1
```

Return the results

```
def count_bases(seq):
  counts = {}
  for base in seq:
    if base not in counts:
      counts[base] = 1
    else:
      counts[base] = counts[base] + 1
  return counts
```

Use the function

```
def count_bases(seq):
  counts = {}
  for base in seq:
    if base not in counts:
      counts[base] = 1
    else:
      counts[base] = counts[base] + 1
  return counts

input_seq = “ATGCATGATGCATGAAAGGTCG”
results = count_bases(input_seq)
for base in results:
  print base, “=”, counts[base]
```

Use the function

```
def count_bases(seq):
  counts = {}
  for base in seq:
    if base not in counts:
      counts[base] = 1
    else:
      counts[base] = counts[base] + 1
  return counts

input_seq = “ATGCATGATGCATGAAAGGTCG”
results = count_bases(input_seq)
for base in results:
  print base, “=”, counts[base]
```

Notice that the variables for the parameters and the return value don’t need to be the same

Interactively

```
>>> def count_bases(seq):
...   counts = {}
...   for base in seq:
...     if base not in counts:
...       counts[base] = 1
...     else:
...       counts[base] = counts[base] + 1
...   return counts
...
>>> count_bases("ATATC")
{'A': 2, 'C': 1, 'T': 2}
>>> count_bases("ATATCQGAC")
{'A': 3, 'Q': 1, 'C': 2, 'T': 2, 'G': 1}
>>> count_bases("")
{}
>>>
```

(I don't even need a variable name - just use the values directly.)

Functions can call functions

```
>>> def gc_content(seq):
...     counts = count_bases(seq)
...     return (counts["G"] + counts["C"]) / float(len(seq))
...
>>> gc_content("CGAATT")
0.333333333333
>>>
```

Functions can be used (almost) anywhere

In an 'if' statement

```
>>> def polyA_tail(seq):
...   if seq.endswith("AAAAAA"):
...      return True
...   else:
...     return False
...
>>> if polyA_tail("ATGCTGTCGATGAAAAAAA"):
...     print "Has a poly-A tail"
...
Has a poly-A tail
>>>
```

Functions can be used (almost) anywhere

In an ‘for’ statement

```
>>> def split_into_codons(seq):
...   codons = []
...   for i in range(0, len(seq)-len(seq)%3, 3):
...     codons.append(seq[i:i+3])
...   return codons
...
>>> for codon in split_into_codons("ATGCATGCATGCATGCATGC"):
...   print "Codon", codon
...
Codon ATG
Codon CAT
Codon GCA
Codon TGC
Codon ATG
Codon CAT
>>>
```

Exercise A

Make a function to add two numbers.
Use the following as a template for your program

```
def add(a, b):
    # ... your function body goes here

print "2+3 =", add(2, 3)
print "5+9 =", add(5, 9)
```

The output from your program should be

```
2+3 = 5
5+9 = 14
```

Exercise B

Modify your program from Exercise A to add three numbers. Use the following as a template for your new program

```
def add3  # you must finish this line
  # then fill in the body

print "2+3+4 =", add(2, 3, 4)
print "5+9+10 =", add(5, 9, 10)
```

Exercise C

Use the count_bases function defined earlier to reimplement Exercise 1 from yesterday. (That was the one which asked for a sequence using raw_input then printed the result.)

Exercise D

Use the count_bases function defined earlier to reimplement Exercise 2 from yesterday. (That was the one which printed count statistics for every sequence in a data file.)

Exercise E

Last Friday's Exercise 5 ("if and files") asked you to write a program which counts the number of sequences with certain properties (eg, the number of sequences with length > 2000 and %GC > 50%). Redo that exercise but this time use a function for each of the tests. The code should look something like:

```
... define the functions first ...

num_over_1000 = 0  # initialize counters

for line in open(...):
  seq = line.rstrip()
  if is_over_1000(seq):
    num_over_1000 = num_over_1000 + 1
  ... other if cases ...

... print the results ...
```

Exercise F

Look at your DNA to protein translation program.

Identify the parts that could be turned into a function. Modify your program accordingly. When finished, ask me to review what you did.