basic functions syntax overview:

```
def functionname([parameter1 [= defaultvalue1] [, parameter2 [= defaultvalue2] [, ...]]]):
 functionbody
 [return returnvalue]
```

why functions?

- to tackle a harder problem in smaller and simpler pieces that are individually testable
- so you "don't repeat yourself!" -- you can call a function over and over with different parameters!

simple function example to add two numbers (or even strings):

```
>>> def add(a, b):
...  return a+b
...
>>> add(2, 3)  # 2 maps to "a" and 3 maps to "b" and the function returns 2+3
5
>>> add(20, 35)  # 20 maps to "a" and 35 maps to "b" and the function returns 20+35
55
>>> add("cat", "dog")
'catdog'
>>>
```

what if we want to add numbers but our computer is incapable of any operations other than adding 1, subtracting 1, and comparing to 1? N.B. this only works for integers!

```
>>> def add(a, b):
...  while (b >= 1):
...   a += 1  # same as a = a+1; we reuse variable a to hold our answer
...   b -= 1  # same as b = b-1
...  return a
...
>>> add(2, 3)
5
>>> add(20, 35)
55
>>>
```

notice:

- we have two implementations of the same interface (add(a, b) returns the sum of a+b)
- they both return the same results (yay)
- the first is faster and depends on more underlying functionality (adding arbitrary numbers)
- the second is slower but only depends on being able to add, subtract, or compare to 1!

nested functions (we can build a "mul" function using only our simple "add" function, as well as (again) subtracting or comparing to 1):

```
>>> def add(a, b):
...  while (b >= 1):
...   a += 1  # same as a = a+1; we reuse variable a to hold our answer
...   b -= 1  # same as b = b-1
...  return a
...
>>> def mul(a, b):
...  c = 0  # this will be our answer
...  while (b >= 1):
...   c = add(c, a)
...   b -= 1  # same as b = b-1
...  return c
...
>>> mul(57, 84)
4788
>>>
```

notice:

- both functions have private (local) variables "a" and "b" that do not interfere with each other

default parameters:

```
>>> def add(a, b = 1):  # if no value is specified for b, we'll pretend it is 1
...  return a+b
...
>>> add(5, 6)
11
>>> add(4)
5
>>> add(5, 6, 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() takes from 1 to 2 positional arguments but 3 were given
>>>
```

locals/globals (by default, variables used in a function are private to that function unless they are declared global)

recursion (a function can call itself for a mind-bender):

```
>>> def add(a, b):
...  if (b == 1):
...   return a+1
...  return add(a+1, b-1)
...
>>> add(2, 3)
5
>>> add(20, 35)
55
>>>
```

notice:

- we now have three implementations of the same interface (add(a, b) returns the sum of a+b)
- tail recursion (a function calling itself from its return path) can always be simplified into iteration (i.e., looping)