Functions

Functions have two parts: function definitions and call sites.

A definition starts out like this

```
def my_function(some_argument, another_argument):
    ____________
    ____________
    ____________
    ____________
    return ____
```

notice the colon

notice the indentation (usually four spaces — shortcut is tab key)

As with while loops and for loops, the statements that belong to the function are established with the : and the indentation.

```
other code
more code
not part of function defn
```

Call sites look like this

```
my_function(2, 3)
my_function(90, 95)
```

The function definition doesn't exist until you send it to the kernel. It is not enough for it to exist in the notebook. If you edit it, send it again.

More Realistic function Example

```
def power_of_two (exponent):
    _____
    _____
    _____
    return the_answer
```

Example call sites

```
really_big_number = power_of_two (20)
medium_number = power_of_two (10)
small_number = power_of_two (3)
```

```
really_big_number, medium_number, small_number
```

1048576 1024 8

Here is an implementation of power_of_two

```
def power_of_two( exponent):
    i=0
    the_answer = 1
    while  i<exponent:
        the_answer = 2* the_answer
        i= i+1
    return  the_answer
```

Variable Scope

The best way to illustrate this is with an example

```
i=5
# the same function definition
def power_of_two(exponent):
    i=0
    the_answer = 1
    while i<exponent:
        the_answer = 2* the_answer
        i= i+1
    return the_answer

power_of_two(8)
```

i ← what is i now??

A totally reasonable guess is 8. However it is 5! The variable inside the function "shadows" the variable in the outer scope.

More on Scope

Inside a function, the variables that are defined outside the function can be read.

That's how we read k_over_m inside the acceleration function.

However if you were to do something whacky like change k_over_m inside the function, once the function is exited, the change no longer exists.

Because of the possibility of confusion, avoid "shadowing" variables in a code cell. It is easy enough to pick new variable names.

Variable Lifetime

If it isn't obvious from the preceding, I should emphasize that a new variable i comes into existence each time power_of_two executes. i is not remembered by the kernel between executions.

Pass-by-Value Example

Here is another implementation of power_of_two

```
def power_of_two(exponent):
    the_answer = 1
    while exponent > 0:
        the_answer = 2 * the_answer
        exponent = exponent - 1
    return the_answer
```

```
exp = 5
a_number = power_of_two(exp)
```

Will exp be decremented to zero??

A reasonable answer is yes, but in fact, the function gets its own copy of exp. So the exp in the outer scope is unchanged.

Pass-by-Reference Example

So you have had a nice pass-by-value example and you think "everything is making sense."

What will this code do?

```
roster = ["Jack", "Jill"]

define add_to_roster(roster, player):
    roster.append(player)
add_to_roster("Hill")

roster
```

↖ Is this copy of roster — in outer scope changed? — if you've been following, you guess "no". But that is wrong ☹

Lists are an example of something that is pass-by-reference.

roster is now ["Jack", "Jill", "Hill"]

About all I'm going to say to explain this is that it is what you want. A list often contains millions of items. It would be wasteful if a copy was made when passing them.