The story so far

- Computation
- Expressions
- Variables
- Function definition
- Later: objects, modules, more types, numbers...

- Flow control:
  - `if` statements
  - loops: `while`, `for` and iteration

Today

Functions (with a bit more structure):

- Semantics
- Syntax (both call *and* definition)
- Parameters and arguments

Don't repeat yourself!

What's wrong with this code?

- technically nothing **iff** we know how many characters are in `name`
- but we **don't** always know
- also... ick!

```
count = 0

if name[0] == 'e':
    count += 1

if name[1] == 'e':
    count += 1

if name[2] == 'e':
    count += 1

if name[3] == 'e':
    count += 1

# ...
```


This code, in addition to being inflexible, offends our programmer's sense of aesthetics.
Programmers are _____: we don't like to repeat ourselves!

More repetition

What's wrong here?

- lexical ordering of words
- comparing word A to B is fine
- comparing A to B and B to C is repetitive
- what if we need to compare lots of words in a game?

```
for i in range(word_lenth):
    if word_a[i] < word_b[i]:
        print(word_a, '<', word_b)
        break

    if word_b[i] < word_a[i]:
        print(word_b, '<', word_a)
        break

for i in range(word_lenth):
    if word_b[i] < word_c[i]:
        print(word_b, '<', word_c)
        break

    if word_c[i] < word_b[i]:
        print(word_c, '<', word_b)
        break
```

Functions

A way of creating *abstractions*

A procedure that we can:

- define once
- use many times

Help us stay DRY (don't repeat yourself)

Function call

We've been using this for a while:

```
print('hello')
```

A function call is an *expression* that *evaluates* to something:

```
a = analog_read(2)
```

A call can take multiple *arguments:*

```
digital_write(4, True)
```

Function definition

- `def` keyword
- function name ("valid"?)
- parameters
- docstring*
- function *body*

```
def any_valid_name(x, y, z):
    """Example of function documentation.

    It's common to start a function with a
    description in a Python "docstring".
    """

    # we can return any expression, or None
    return x + y * z
```

* A *triple-quoted* string can have **multiple lines** in it, and, it's safe to use either single (') or double (") quotes without causing confusion ("is this the end of the string?").

Parameters vs arguments

Parameters: *variables* initialized by arguments

```
def abs(x):
    if x >= 0:
        positive = x
    else:
        positive = -x

    return positive
```

Arguments: *values* passed into functions

```
abs(-2)
```

Function docstrings

- description of function
- like a comment: for people, not the computer
- often *triple-quoted**

```
def any_valid_name(x, y, z):
    """Example of function documentation.

    It's common to start a function with a
    description in a Python "docstring".
    """

    # we can return any expression, or None
    return x + y * z
```

* A *triple-quoted* string can have **multiple lines** in it, and, it's safe to use either single (') or double ('') quotes without causing confusion ("is this the end of the string?").

Function body

- one or more statements
- can be *any* statement
- can be `pass`

```
def any_valid_name(x, y, z):
    """Example of function documentation.

    It's common to start a function with a
    description in a Python "docstring".
    """

    # we can return any expression, or None
    return x + y * z
```


A function body can include any kind of statement: assignment, `if` statement, loop...

A function body must have _____ one statement in it. However, that statement can be `pass`, which means "do nothing".

Return value

- a function's "output"
  - input: arguments to parameters
  - output: `return` value
- can be any value
- can be `None`

```
def any_valid_name(x, y, z):
    """Example of function documentation.

    It's common to start a function with a
    description in a Python "docstring".
    """

    # we can return any expression, or None
    return x + y * z
```


A function with no `return` statement implicitly returns **`None`**.