## Previously

**Functions (with a bit more structure):**

- Definition and call syntax
- Parameters and arguments
- Variable *scope*
- Recursion

## Today

**Modules**

- Python files as scripts **(or not)**
- The `import` statement
- Python scripts and `__main__`

## Python code quantities

**Often measured in SLoC (*source lines of code*)**

- labs: maybe as much as 200 lines?
- `engi1020`: 1031 lines
- qutebrowser: 40k lines
- Python standard library: 500k lines
- no limit to amount of code you can write!

Large, complex pieces of software like Web browsers and operating systems have ________________ of lines of code. We don't write all of those lines in one source file!

## Recall: organizing your writing

**How would you organize:**

- a text?
- a letter?
- an essay?
- a report?
- a book?

## Organizing Python

**We've seen:**

- expressions (like phrases)
- statements (like sentences)
    - "simple" statements: assignment, `pass`, `return`, ...
    - "compound" statements: `if`, `while`, `for`, `def`, ...
- functions (like paragraphs)

"Simple" and "compounds" statements are a distinction that's made by the full Python grammar, which is the "last word" on the syntactic rules of the langauge. However, you're definitely not expected to understand all of these grammatical rules after finishing just one introductory course!

## Today

**Modules**

- Like *chapters* and *parts*
- Organize larger chunks of code into hierarchies
- Multiple files working together
- The `import` statement (in more detail / with more background than we've seen before!)

## Python files

**So far, we've written Python *scripts:***

- files containing statements
- statements executed one at a time

**Can also use Python files as *modules:***

- files containing statements
- statements executed one at time

***What's the difference?***

## How we run Python code

**Python scripts:**

- open a file
- click "Run" (or run from the command line)

**Python modules**

- run when we *import* them
- "import"... where have we seen that before?

## `import` *statement*

**Makes names from a module available for use:**

```
from math import *
y = sin(0)
```

```
import math
y = math.sin(0)
```

```
import central
m = central.mean([1, 2, 3])
```

But how does this actually work?

## Importing modules

**When we `import` a module:**

- Python *interpreter* looks for a file with that name + `'.py'`
- interpreter *executes* its statements
- result: *module* with global names accessible with `.`

```
import math
y = math.sin(x)
z = math.cos(y)
```

Where the Python interpreter finds a module can be a bit complicated. One set of places it looks is the list of directories contained in `sys.path`:

```
>>> import sys
>>> print(sys.path)
['/Users/jon/Documents/Teaching/1020/website/content/lectures/17', '/Applications/Thonny.app/Contents/Frameworks/Python.framework/Versions/3.7/lib/python37.zip', '/Applications
```

For our purposes, the most important of these is the first entry: the ________________________.

## `import` *syntax*

```
import goodstuff
goodstuff.greet("Jon")
```

```
import goodstuff as stuff
stuff.boots_filled = True
```

```
from goodstuff import scald
if scald:
    print("Got 'er scald!")
```

```
from goodstuff import greet as hello
hello("world")
```

## Aside

`dir` tells us names in a module (or other things we'll see later):

```
>>> import engi1020.arduino.api
>>> dir(engi1020.arduino.api)
[#...
 'analog_read',
 'analog_write',
 'buzzer_frequency',
 #...
 'temp_humid_getTemp']
```

## Python modules

**So how can we write/use our own modules?**

1. Write a Python file (just like we've been doing)
2. Save it with a *valid identifer* name + `.py`
3. `import` it from a script *in the same directory*
4. Refer to its *attributes* (global names)

## Python modules that are scripts

**One potential problem:**

```
def add(x, y):
    return x + y

# Some test code:
test_x = input('x?')
test_y = input('y?')
result = add(test_x, test_y)
```

**What's the problem?**

If you submit something like this to Gradescope as an assignment, the autograder will try to `import` your code and it will ________________ waiting for user input. A module should not do this kind of computation __________________. So what can we do instead if we want to write test code (which is a good idea)?

## Separating test code

### Separation of concerns

Separate modules for code that does different things:

`central.py` : implementation of assignment 2 (**individual**)

`test.py` : *tests* for assignment 2 (can be **shared**)

```
import central

result = central.mean([1, 2, 3])
expected = 2.0
if result != expected:
    print("mean returned '" + result + "'; expected '" + expected + "'")
```

## Python module `__name__`

**A special variable containing the module's name**

**In scripts: will be `'__main__'`**

```
def add(x, y):
    return x + y


if __name__ == '__main__':
    # Run the tests:
    test_x = input('x?')    # etc.
```

## Summary

**Modules**

- Python files as scripts **(or not)**
- The `import` statement
- Python scripts and `__main__`