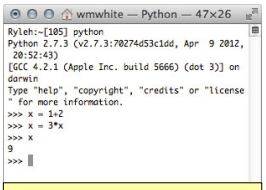
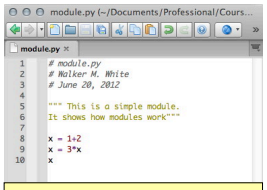
Academic Integrity Quiz

- Reading quiz about the course AI policy
  - Go to http://www.cs.cornell.edu/courses/cs1110/
  - Click **Academic Integrity** in side bar
  - Read and take quiz in CMS
- Right now, missing ~100 enrolled students
  - If you do not take it, you must drop the class
- Will grade and return by Friday
  - If you missed questions, you will retake

Python Shell vs. Modules





- Launch in command line
- Type each line separately
- Python executes as you type

- **Write in a text editor**
  - We use Komodo Edit
  - But anything will work
- Run module with `import`

Using a Module

Module Contents

```
# module.py

""" This is a simple module.
It shows how modules work"""

x = 1+2
x = 3*x
x
```

"**Module data**" must be prefixed by module name

Prints **docstring** and module contents

Python Shell

```
>>> import module
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> module.x
9
>>> help(module)
```

We Write Programs to Do Things

- Functions are the **key doers**

Function Call

- Command to **do** the function

```
greet('Walker')
```

argument to assign to n

Function Definition

- Defines what function **does**

```
def greet(n):
    print 'Hello '+n+'!'
```

Function **Header**

declaration of **parameter** n

Function **Body** (indented)

- **Parameter**: variable that is listed within the parentheses of a method header.
- **Argument**: a value to assign to the method parameter when it is called

Anatomy of a Function Definition

name | parameters

```
def greet(n):
    """Prints a greeting to the name n

    Precondition: n is a string
    representing a person's name"""
    print 'Hello '+n+'!'
    print 'How are you?'
```

Function **Header**

Docstring **Specification**

Statements to execute when called

The vertical line indicates indentation

Use vertical lines when you write Python on **exams** so we can see indentation

Procedures vs. Fruitful Functions

Procedures

- Functions that **do** something
- Call them as a **statement**
- Example: `greet('Walker')`

Fruitful Functions

- Functions that give a **value**
- Call them in an **expression**
- Example: `x = round(2.56,1)`

Historical Aside

- Historically "function" = "fruitful function"
- But now we use "function" to refer to both

The `return` Statement

- Fruitful functions require a **return statement**
- **Format**: return <*expression*>
  - Provides value when call is used in an expression
  - Also stops executing the function!
  - Any statements after a **return** are ignored
- **Example**: temperature converter function

```
def to_centigrade(x):
    """Returns: x converted to centigrade"""
    return 5*(x-32)/9.0
```

Module Example: Temperature Converter

```
# temperature.py
"""Conversion functions between fahrenheit and centrigrade"""

# Functions
def to_centigrade(x):
    """Returns: x converted to centigrade"""
    return 5*(x-32)/9.0

def to_fahrenheit(x):
    """Returns: x converted to fahrenheit"""
    return 9*x/5.0+32

# Constants
FREEZING_C = 0.0  # temp. water freezes
...
```

Style Guideline: Two blank lines between function definitions

Print vs. Return

Print	Return
• Displays a value on screen ▪ Used primarily for **testing** ▪ Not useful for calculations	• Defines a function's value ▪ Important for **calculations** ▪ But does not display anything

```
def print_plus(n):
    print (n+1)
>>> x = plus_one(2)
3
>>>
```

x

Nothing here!

```
def return_plus(n):
    return (n+1)
>>> x = plus_one(2)
>>>
```

x 3

Anatomy of a Specification

```
def greet(n):
    """Prints a greeting to the name n

    Greeting has format 'Hello <n>!'

    Precondition: n is a string
    representing a person's name"""
    print 'Hello '+n+'!'
```

One line description, followed by blank line

More detail about the function. It may be many paragraphs.

Precondition specifies assumptions we make about the arguments

Preconditions

- Precondition is a promise
  - If precondition is true, the function works
  - If precondition is false, no guarantees at all
- Get **software bugs** when
  - Function precondition is not documented properly
  - Function is used in ways that violates precondition

```
>>> to_centigrade(32)
0.0
>>> to_centigrade(212)
100.0
>>> to_centigrade('32')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "temperature.py", line 19 ...
TypeError: unsupported operand type(s)
for -: 'str' and 'int'
```

Precondition violated

Global Variables and Specifications

- Python *does not support* docstrings for variables
  - Only functions and modules (e.g. first docstring)
  - help() shows "data", but does not describe it
- But we still need to document them
  - Use a single line comment with #
  - Describe what the variable means
- **Example**:
  - `FREEZING_C = 0.0  # temp. water freezes in C`
  - `BOILING_C = 100.0  # temp. water boils in C`