

Lecture 4

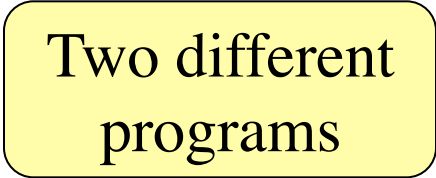
Defining Functions

Academic Integrity Quiz

- Reading quiz about the course AI policy
 - Go to <http://www.cs.cornell.edu/courses/cs11110/>
 - 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

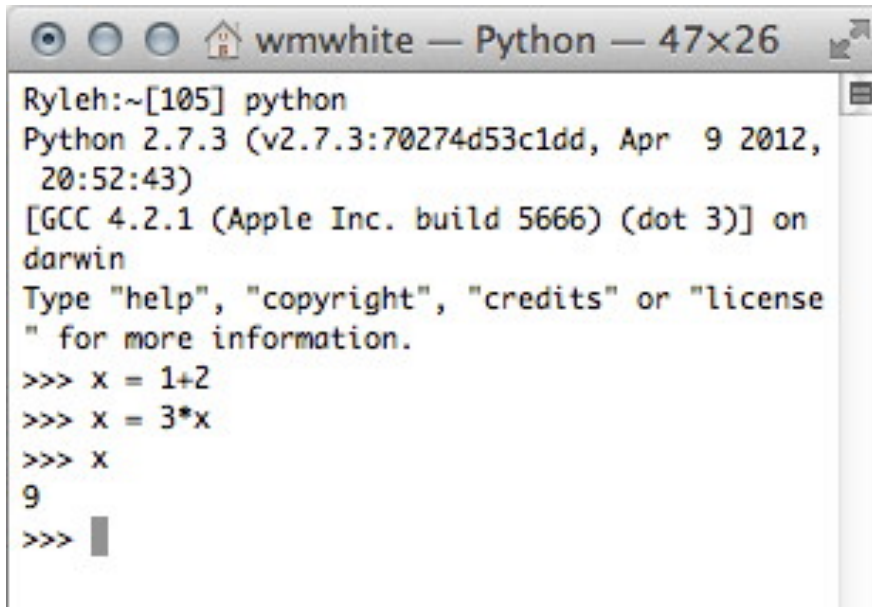
Recall: Modules

- Modules provide extra functions, variables
 - **Example:** math provides `math.cos()`, `math.pi`
 - Access them with the `import` command
- Python provides a lot of them for us
- **This Lecture:** How to make modules
 - Komodo Edit to *make* a module
 - Python to *use* the module



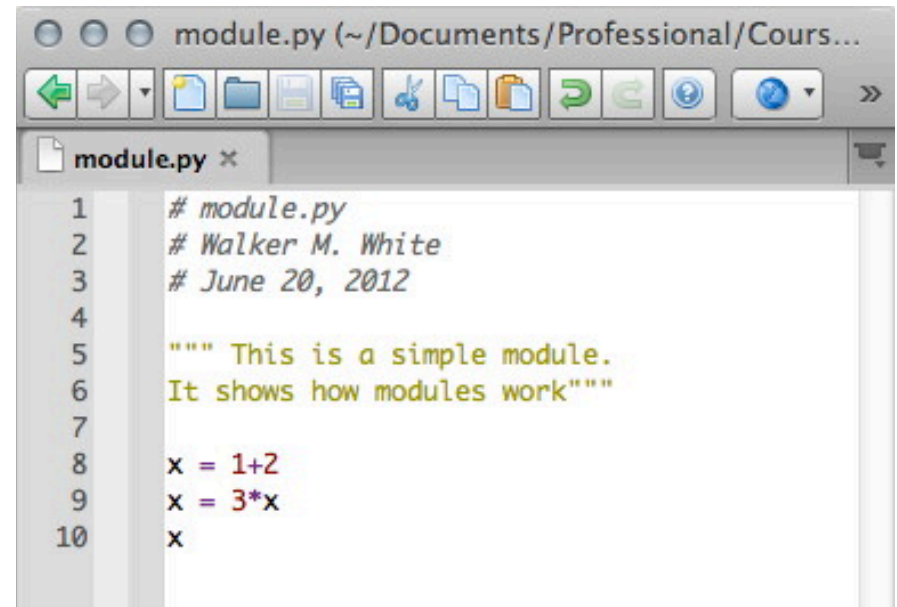
Two different programs

Python Shell vs. Modules



```
wmwhite — Python — 47x26
Ryleh:~[105] python
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr 9 2012,
20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on
darwin
Type "help", "copyright", "credits" or "license
" for more information.
>>> x = 1+2
>>> x = 3*x
>>> x
9
>>>
```

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



```
module.py (~/Documents/Professional/Cours...
module.py x
1 # module.py
2 # Walker M. White
3 # June 20, 2012
4
5 """ This is a simple module.
6 It shows how modules work"""
7
8 x = 1+2
9 x = 3*x
10 x
```

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

Using a Module

Module Contents

```
# module.py
```

Single line comment
(not executed)

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

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

```
x = 1+2
```

```
x = 3*x
```

Commands
Executed on import

```
x
```

Not a command.
import **ignores this**

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>
```

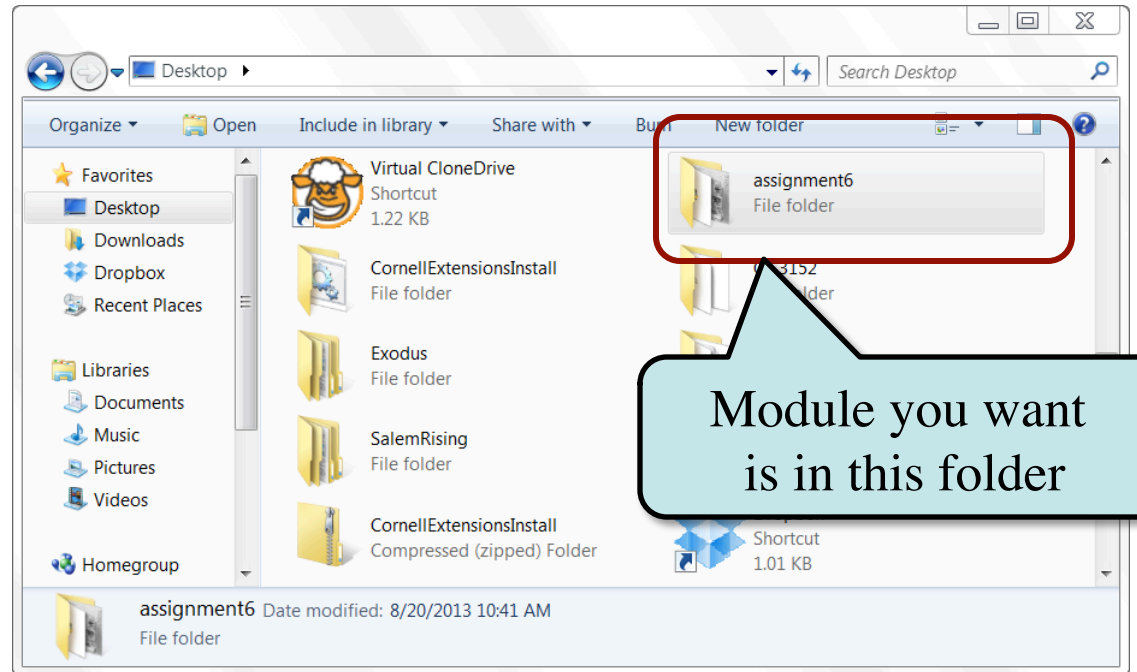
```
NameError: name 'x' is not defined
```

```
>>> module.x
```

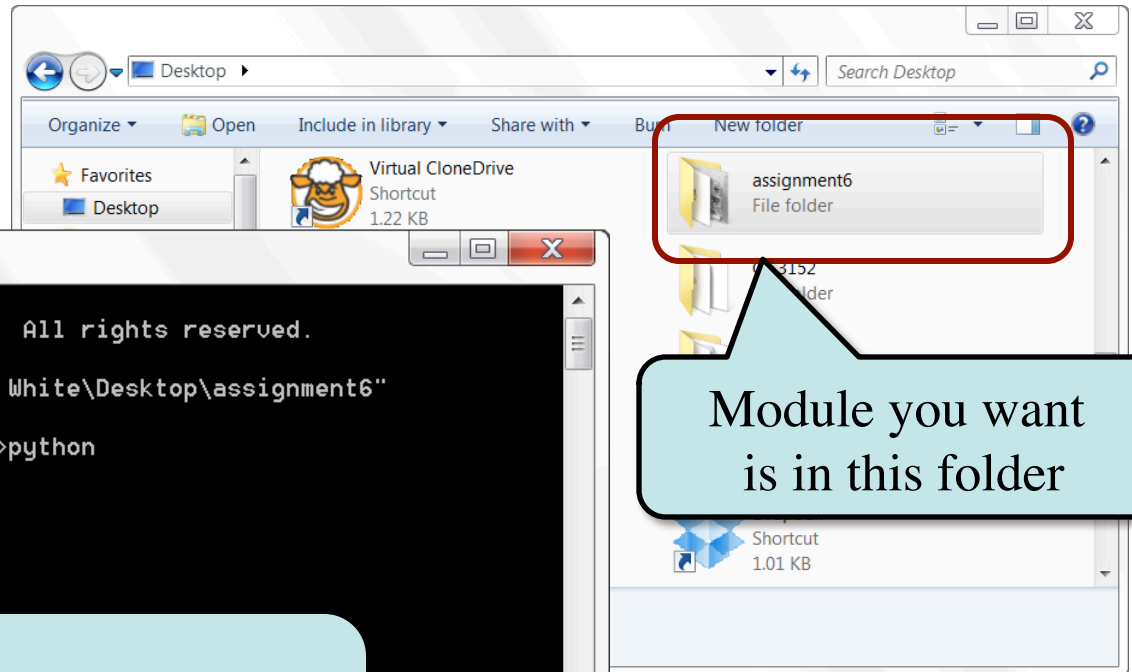
```
9
```

```
>>> help(module)
```

Modules Must be in Working Directory!



Modules Must be in Working Directory!



```
Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Walker White>cd "C:\Users\Walker White\Desktop\assignment6"
C:\Users\Walker White\Desktop\assignment6>python
```

Have to navigate to folder
BEFORE running Python

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
Header

Function Definition

- Defines what function **does**

```
def greet(n):
```

```
    print 'Hello '+n+'!'
```

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):
```

Function Header

```
    """Prints a greeting to the name n
```

```
    Precondition: n is a string
    representing a person's name"""
```

```
    print 'Hello '+n+'!'
```

```
    print 'How are you?'
```

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
```

Print vs. Return

Print

- Displays a value on screen
 - Used primarily for **testing**
 - Not useful for calculations

```
def print_plus(n):
```

```
| print (n+1)
```

```
>>> x = plus_one(2)
```

```
3
```

```
>>>
```

Return

- Defines a function's value
 - Important for **calculations**
 - But does not display anything

```
def return_plus(n):
```

```
| return (n+1)
```

```
>>> x = plus_one(2)
```

```
>>>
```

Print vs. Return

Print

- Displays a value on screen
 - Used primarily for **testing**
 - Not useful for calculations


```
def print_plus(n):
```

```
    print (n+1)
```

```
>>> x = plus_one(2)
```

```
3
```

```
>>>
```

x 

Nothing here!

Return

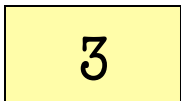
- Defines a function's value
 - Important for **calculations**
 - But does not display anything

```
def return_plus(n):
```

```
    return (n+1)
```

```
>>> x = plus_one(2)
```

```
>>>
```

x 

Functions and Modules

- Purpose of modules is **function definitions**
 - Function definitions are written in module file
 - Import the module to call the functions
- Your Python workflow (right now) is

1. Write a function in a module (a .py file)
2. Open up the command shell
3. Move to the directory with this file
4. Start Python (type python)
5. Import the module
6. Try out the function

Aside: Constants

- Modules often have variables outside a function
 - We call these global variables
 - Accessible once you import the module
- Global variables should be **constants**
 - Variables that never, ever change
 - Mnemonic representation of important value
 - **Example:** `math.pi`, `math.e` in `math`
- In this class, constant names are **capitalized!**
 - So we can tell them apart from non-constants

Module Example: Temperature Converter

```
# temperature.py
```

```
"""Conversion functions between fahrenheit and centigrade"""
```

```
# 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

Example from Previous Lecture

```
def second_in_list(s):
```

```
    """Returns: second item in comma-separated list
```

```
    The final result does not have any whitespace on edges
```

```
    Precondition: s is a string of items separated by a comma."""
```

```
    startcomma = s.index(',')
```

```
    tail = s[startcomma+1:]
```

```
    endcomma = tail.index(',')
```

```
    item = tail[:endcomma].strip()
```

```
    return item
```

See commalist.py

Recall: The Python API

The image shows a screenshot of the Python documentation for the `math.ceil(x)` function. Three callout boxes highlight key parts of the documentation:

- Function name:** `math.ceil(x)`
- Number of arguments:** 1 (the parameter `x`)
- What the function evaluates to:** "Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`."

The screenshot also shows the following text from the documentation:

so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all

representation functions

9.1. numbers — Numeric abstract base classes

Next topic
9.3. cmath — Mathematical functions for complex numbers

This Page
Report a Bug
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

math.ceil(x)
Return the ceiling of `x` as a float, the

math.copysign(x, y)
Return `x` with the sign of `y`. On a platform where `float` has IEEE 754 floating point arithmetic, `copysign(x, y)` returns `x` with the sign of `y`.
New in version 2.6.

math.fabs(x)
Return the absolute value of `x`.

math.factorial(x)
Return `x` factorial. Raises `ValueError` if `x` is not a non-negative integer.
New in version 2.6.

math.floor(x)
Return the floor of `x` as a float, the

- This is a **specification**
 - Enough info to use func.
 - But not how to implement
- Write them as **docstrings**

Anatomy of a Specification

```
def greet(n):
```

```
    """Prints a greeting to the name n
```

```
    Greeting has format 'Hello <n>!'
    Followed by a conversation starter.
```

```
    Precondition: n is a string
    representing a person's name"""
```

```
    print 'Hello '+n+'!'
    print 'How are you?'
```

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

Anatomy of a Specification

```
def to_centiGrade(x):
```

```
    """Returns: x converted to centigrade
```

```
    Value returned has type float.
```

```
    Precondition: x is a float measuring
    temperature in fahrenheit"""
```

```
return 5*(x-32)/9.0
```

“Returns” indicates a fruitful function

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`