CS1110 note-taking aid, Thu Jan 31, 2013

Announcements

Unable to enroll because of section conflicts? Check Student Center regularly to see if space opens up.

Install Python, Komodo Edit, and the "Run Python Module" button. The first assignment is coming next week.

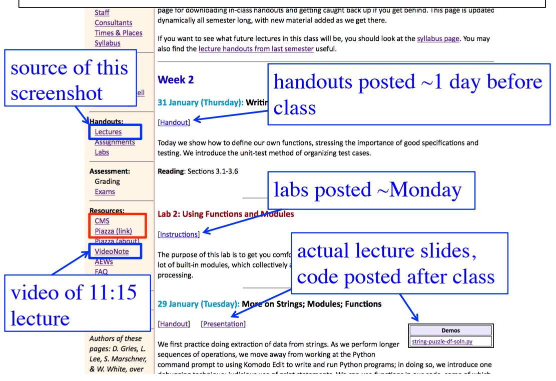
No reading for next time: Our treatment of *objects* differs significantly from the book's.

iClickers that need to be registered: See Texts on webpage for instructions.

Remote ID	Remote ID
#005BBEE5	#24594934
#01C99058	#35040534
#0C7B93E4	#3813FED5
#0CABFC5B	#389DB316
#0D2D1838	#3A0B4879
#11756105	#0BB03D86
#19C3964C	#0D425A15
#25E34187	#33CB52AA
#33005F6C	#3412EDCB
#33D032D1	#3503EADC
#3684EC5E	#37165677
#369912BD	#39D3846E
#36CFBF46	#0C8E1795
#39241409	#14D15F9A
#39425823	#22E9F13A
#3962A1FA	#0C79F184
#3667FEAF	#0CC572BB
	#31FA25EE
	#321B321B
	#327F4409

Slides by D. Gries, L. Lee, S. Marschner, W. White

http://www.cs.cornell.edu/courses/cs1110/2013sp



Authors of these pages: D. Gries, L. Lee, S. Marschner, & W. White, over

page for downloading in-class handouts and getting caught back up if you get behind. This page is updated dynamically all semester long, with new material added as we get there.

If you want to see what future lectures in this class will be, you should look at the syllabus page. You may also find the lecture handouts from last semester useful.

Week 2

31 January (Thursday): Writin

[Handout]

Today we show how to define our own functions, stressing the importance of good specifications and testing. We introduce the unit-test method of organizing test cases.

Reading: Sections 3.1-3.6

Lab 2: Using Functions and Modules

[Instructions]

The purpose of this lab is to get you comfo
lot of built-in modules, which collectively a
processing.

29 January (Tuesday): More on Strings; Modules; Functions

[Handout] [Presentation]

We first practice doing extraction of data from strings. As we perform longer sequences of operations, we move away from working at the Python command prompt to using Komodo Edit to write and run Python programs; in doing so, we introduce one

Users Want Functions

Given: info contains a comma-separated string with last name, difficulty, execution, and penalty.

- *Example:* info = 'RAISMAN, 6.7, 9.1,0'

Goal: store the difficulty as a string, with no extra spaces or punctuation, in variable df

Users (including other programmers) want to write things like:

```
raisman_df = gym.dscore('RAISMAN, 6.7, 9.1,0')
ponor_df    = gym.dscore(' PONOR , 6.2 ,  9.0 , 0')
```

The function dscore is in module (file) gym.

When *called*, it *returns* a value that the user can utilize as they wish.

Anatomy of a Function Definition (I)

In file gym, we define dscore as follows.

header → **def** dscore(info):

declaration of *parameter* (variable) named "info"

specification:

```
    """Returns: difficulty score, as a float, represented in info.

    Precondition: info is a string with commas separating its
    component values: last name, difficulty score, execution
    score, penalty."""
```

body (indented):

```
    startcomma = info.index(',')
    tail = info[startcomma+1:] # part of
    endcomma = tail.index(',')
    return float(tail[:endcomma].strip())
```

after **return**, the expression whose value results from the function call

Parameters: Variables Holding Input Values

```
def dscore(info):
    """Returns: difficult

    Precondition: info i
    component values:
    score, penalty."""

    startcomma = info.index(',')
    tail = info[startcomma+1:] # part of info after 1st ,
    endcomma = tail.index(',')
    return float(tail[:endcomma].strip())
```

When you call a function, you supply *arguments*: input values.

ex: gym.dscore('Raisman, 6.7, 9, 0')

These values are stored in the function's corresponding *parameters*: variables used within the function.

Anatomy of a Specification: User Documentation

Single summary line, followed by blank line. (More detail can be added in separate paragraphs)

```
def dscore(info):
    """Returns: difficulty score, as a float, represented in info.

    Precondition: info is a string with commas separating its
    component values: last name, difficulty score, execution
    score, penalty."""

    startcomma = info.index(',')
    tail = info[startcomma+1:] # part of info after 1st ,
    [...]
```

Precondition: assumptions about the argument values

CS1110 note-taking aid, Thu Jan 31, 2013

A Specification is a Contract

Preconditions are a promise that:

- if the arguments satisfy the preconditions, the function works as described in the specification;
- but, if the user's arguments violate the precondition, all bets are off.

```
>>> gym.dscore('R; 6.7, 9,0')
"I'm sorry Dave, I'm afraid I can't do that"
```

So write these contracts carefully!

Common sources of **software errors:**

- Preconditions not documented properly
- Functions used in ways that violate preconditions

Testing Program "Correctness"

- **Bug**: Error in a program. (Always expect them!)
- **Debugging**: Process of finding bugs and removing them.
- **Testing**: Process of analyzing, running program, looking for bugs.
- **Test case**: A set of input values, together with the expected output.

Get in the habit of writing test cases for a function from the function's specification —even *before* writing the function's body.

```
def number_vowels(w):
    """Returns: number of vowels in word w.

    Precondition: w string w/ at least one letter and only letters"""
    pass  # nothing here yet!
```

Organizing Test Cases: Unit Tests

- A unit test is a module that tests another module
  - It imports the other module (so it can access it)
  - It imports the cunittest module (provided by us)
  - It defines one or more test procedures
    - Evaluate the function(s) on the test cases
    - Compare the result to the expected value
  - It has special code that calls the test procedures
- The test procedures use the cunittest function

```
def assert_equals(expected,received):
    """Quit program if expected and received differ"""
```

Example unit test: last_name_first(n)

```
# test procedure
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    unittest.assert_equals('White, Walker',
                           last_name_first('Walker White'))
    unittest.assert_equals('White, Walker',
                           last_name_first('Walker     White'))

# Application code
if __name__ == '__main__':
    test_last_name_first()
    print 'Module name is working correctly'
```

Expected is the literal value.

Received is the expression.

Quits Python if not equal

Message will print out only if no errors.

Aside: Application Code

Applications often have "application code"

- Code not executed if imported; only if run as app/ Komodo "Run Python Module" button
- Indented under the line

```
if __name__ == '__main__':
```

Debugging with Print Statements

Print statements expose the values of variables, so you can check if they have the value you expect.

```
print 'in this solution, df is :' + df + ':'
```

Don't leave these in your finished code! They reduce readability.