CS 1
Introduction to Computer Programming

Lecture 1: September 28, 2015

Orientation and basics

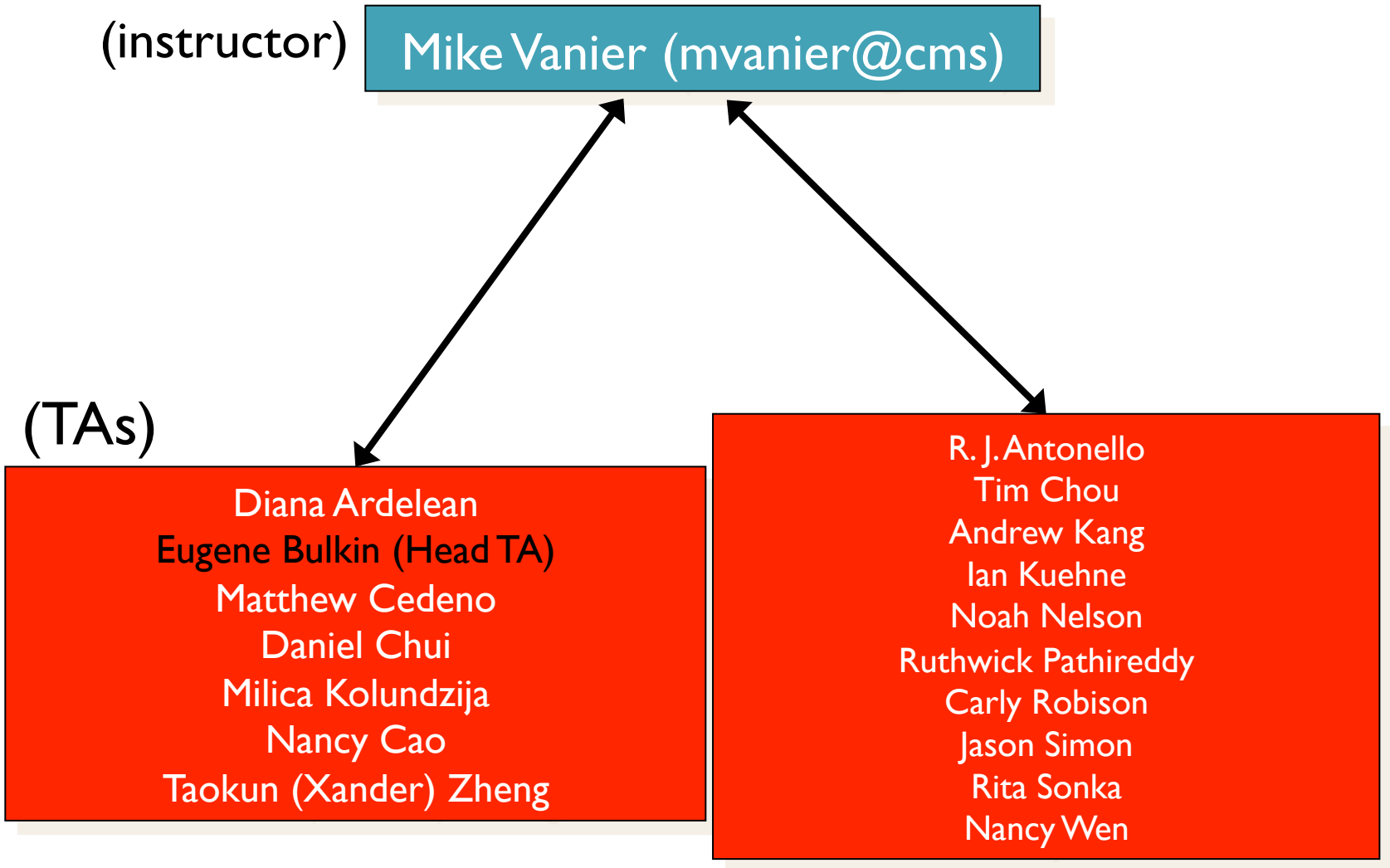
[add cards signed at end of class]

Outline

- [15] Administrative details
- [15] Overview of course
- [20] Introduction to Python

Administrative details

People


(instructor)
Mike Vanier (mvanier@cms)
(TAs)
Diana Ardelean
Eugene Bulkin (Head TA)
Matthew Cedeno
Daniel Chui
Milica Kolundzija
Nancy Cao
Taokun (Xander) Zheng
R. J. Antonello
Tim Chou
Andrew Kang
Ian Kuehne
Noah Nelson
Ruthwick Pathireddy
Carly Robison
Jason Simon
Rita Sonka
Nancy Wen

Website

- http://moodle.caltech.edu
  - Follow links to CS001
  - Guest login password (if you need it) is *nohtyp* (python spelled backwards)
    - Should be able to login as a student without this
  - "Enrolment key" is the same: *nohtyp*
  - Make sure you enroll in the course so you'll receive course-related email etc.
  - Let me know if there are any problems!

Website

- The website will contain:
  - Slides from lectures
  - Assignments
  - Exams
  - Useful links
  - Supporting documents
  - Admin stuff
  - Surveys/questionnaires
  - etc.

Website

- When you first get onto the website, read:
  - Action items for new students
  - Collaboration policies
  - CMS cluster do's and don'ts
- Browse the "Course policies/FAQ" document (long!)
- After getting a CMS cluster account (see the action items page for this), fill in the lab section signup sheet

csman

- Grades and lab submissions are *not* handled through the moodle website but through the csman system
- Grading website is: csman.cms.caltech.edu
- Homework is submitted to csman, retrieved by graders, graded and comments uploaded back there

Organization

- Core lectures: 2x50min/ week
  - Introduce new material
  - Punctuate with interludes (midpoint)
  - Movie clips
- Supplemental lectures: 1x50min/week
  - Cover additional material
- Staffed lab: 4d x 8 hrs
  - TAs, assigned times
- Weekly assignments
  - Allow rework (1 wk)
- Exams
  - midterm, final
- Point system

Friday lectures

- Friday lectures will happen roughly every other week, starting with this week
- Held in this room
- Cover less critical (but still important!) material
  - problem-solving skills, debugging
  - some additional topics

Attendance

- Attendance is *optional*
- We hope it will be worth your while to come to lectures, but we will not force you to
- Similarly, attendance at lab sections is optional

Your Part

- Go to lectures and sections
- Ask your TA for help
- Learn from your mistakes
  - (rework gives you a chance to do so)
- Tell us what makes things hard
  - Fill out feedback forms, talk to ombudspeople, TAs, instructors…

Point System

• Homework (0 - 3) x 7	21
• Midterm	6
• Final	18
• **Possible**:	45
• Pass:	28

Grading Homework

- Set has multiple parts (A through F, e.g.)
- Score for entire set is *minimum* of the section scores
- Each part receives integer in [0, 3]:

0	*incorrect (worthy of no credit)*
1	*insufficient (not passing quality, demonstrates significant bugs which must be addressed)*
2	*good (demonstrates mastery of key idea, may have a few minor bugs)*
3	*excellent (masters all important parts)*

Time on Assignments

- Taking excessive time on a problem is a **symptom** that you missed an important concept
  - It's possible to do things the **wrong** way (missing what we're teaching you)
    - …but it probably will take you a lot more time
- Time hint on assignments
  - suggested upper bound
  - if takes much longer → you should see a TA
  - Don't spend hours alone on a problem!

Rework Philosophy

- Programs that are 1% wrong are **wrong**

  …as far as the computer is concerned
- So: we allow rework on assignments!
- Point: we want you to *master* the material
  - not just make things work by trial and error!

Collaboration policy

- You are welcome to collaborate informally on your assignments with other individuals who are taking or have taken the class, but you must write all of your own code!
- Honor code violations include:
  - Copying another student's code verbatim or nearly verbatim
  - Taking another student's file and modifying it to make it look different (changing variable names etc.)
  - Consulting a printed or electronic version of another student's code and referring to it while you write your own code
  - Having someone dictate code to you while you type
- Basically, you have to do your own work!

Collaboration policy

- Getting advice on solution methods (algorithms etc.) from other students is OK as long as you don't copy code from other students
- Letting other students copy from *you* is just as much of an Honor Code violation as copying from someone else
- Copying from homework submissions in past years is also an Honor Code violation (of course)

Collaboration policy

- The "50 foot rule": If you want to help someone with their CS 1 homework, your own code should be ≥50 feet away (*i.e.* not visible, not accessible)
  - *"Help them with your brain, not with your code."*
  - Do *not* show your code to other people, email it, dictate it over the phone, put it up on a pastebin site, etc. etc.
- This applies also when helping students to debug their code!
  - don't go and look at your code and come back etc.

Collaboration policy

- Consider this: when helping other students with their code, you are acting as an unofficial TA
- A good TA will not show a student what they (the TA) did to solve the problem, he/she will help the student work through this on their own
- *If you can't do this, don't help other students!*
  - that's why we have official TAs!
- Failure to abide by these guidelines may result in a trip to the Board of Control (BoC)

Lab

- No formal lab sections this week
  - this is "week 0"; next week is "week 1"
- Assignment 1 will not be out until next Monday
- Computer lab is in Annenberg 104
  - need card access to get in (should get this automatically)

Linux tutorial lecture

- First Friday lecture!
- One-hour Linux tutorial
- Useful if you intend to use computers in computer lab (or just for your own information)

Your Action Items

- Visit web site
  - Make sure you can get in successfully
  - "Enroll" in the web site
  - Read anything that's there
  - Check back once/day at least
- Sign up for CMS account (via the web)
  - instructions will be on website "Administrative Information" page
  - also gives you csman access
  - Fill in lab section signup sheet after this is done

Adding the class

- I allow adds up to the end of the second week *only*
- Bring add cards to class, I'll sign them at the end of class
- Don't try to add the day before add day!
  - Unless you like rejection!

Course conflicts

- We want CS 1 to be a core course, but so far it isn't, so other courses sometimes schedule lectures overlapping with CS 1
- Many students report course conflicts with Friday lectures especially
- Good news! I'm willing to sign most course conflict cards
  - BUT: you are responsible for reading the lecture slides yourself and keeping up with the material!

Overview of the course

Goals of CS 1

- A practical introduction to computer programming
- Using the Python programming language
  - language syntax/semantics
  - program design process
  - debugging and testing

Goals of CS 1

- Cover useful/fun application areas
  - numerics, graphics, etc.
- Preparation for future courses
  - CS 2 and 3 (larger-scale programming)
  - CS 4 (deeper fundamentals)
  - CS 11 (more languages)
- Show that programming is fun!
  - (we hope)

Teaching challenges

- Any CS1 course is hard to teach because of the enormous variation in the programming experience of students taking the class
  - probably greater than in any other subject
- As example, let's look at hypothetical CS 1 student profiles

Hypothetical student #1

- "I'm interested in learning how to program, but I've never done any programming of any kind before."

Hypothetical student #2

- "I did some programming in high school. It was fun, but I didn't feel like I really understood what I was doing. I'd like to learn to program the right way."

Hypothetical student #3

- "I've been programming since I was 5 years old. I wrote my own programming language when I was 11. When I was 16, I wrote my own operating system and sold it to Google for $50M. I'm at Caltech to give me something to do between IPOs."

This course

- This course is ideal for students #1 and #2
- Student #3 will find it way too slow for his/her taste
  - should probably take CS 2 and CS 4 instead (next term)
  - but might also want to take CS 1 just to learn Python and pick up some practical skills (or if they want to become a TA)

This course

- Early lectures will be extremely easy for those who have done significant amounts of programming
- After midterm, material will get significantly harder
  - and more interesting/fun!

Choice of language

- There are literally *hundreds* of computer programming languages
  - dozens in common use
- For example:
  - C, C++, Objective-C, Java, C#, D, Go, Rust, Swift
  - Perl, Python, Ruby, Visual Basic, PHP
  - Matlab, Mathematica, Fortran
  - Scheme, Ocaml, Erlang, Scala, Haskell

Choice of language

- We will use the Python programming language in this course because
  - It has a straightforward syntax and semantics
  - It is a general purpose language
  - It has a huge variety of application domains
  - It is heavily used in science and industry
    - Google uses Python extensively
    - Many Caltech labs use Python
  - Lots of useful libraries exist
  - It's fun to program in!

One language to rule them all?

- Will Python be the only computer language you'll ever need?
- Almost certainly not! ☹
- Almost all serious programmers will have to know multiple languages to be productive

One language to rule them all?

- Scientific programmers
  - will probably know C, C++, (Fortran), Python, Mathematica, Matlab, maybe Java
- Web programmers
  - will probably know Java and/or C#, Perl or Python or Ruby or PHP, plus HTML, CSS, etc.
- Functional programmers
  - One or more of: Scheme, Lisp, Ocaml, Erlang, Haskell, Scala
- Other interesting languages: Rust, D, Go …

One language to rule them all?

- Whatever your goals, Python will probably be useful and is a great foundation on which to learn new languages
- Languages have many similarities, so once you've learned one, learning another is generally much easier

Introduction to Python

Caltech CS 1: Fall 2015

What is Python?

- Python is a computer programming language
- Named after *"Monty Python's Flying Circus"*
- Designed by Guido van Rossum starting in 1991
  - and continuing to this day (new versions)
- Most recent version is Python 3.5
- We will use Python 2.7.10 in this course
  - mostly because of greater availability of code modules

Writing programs in Python

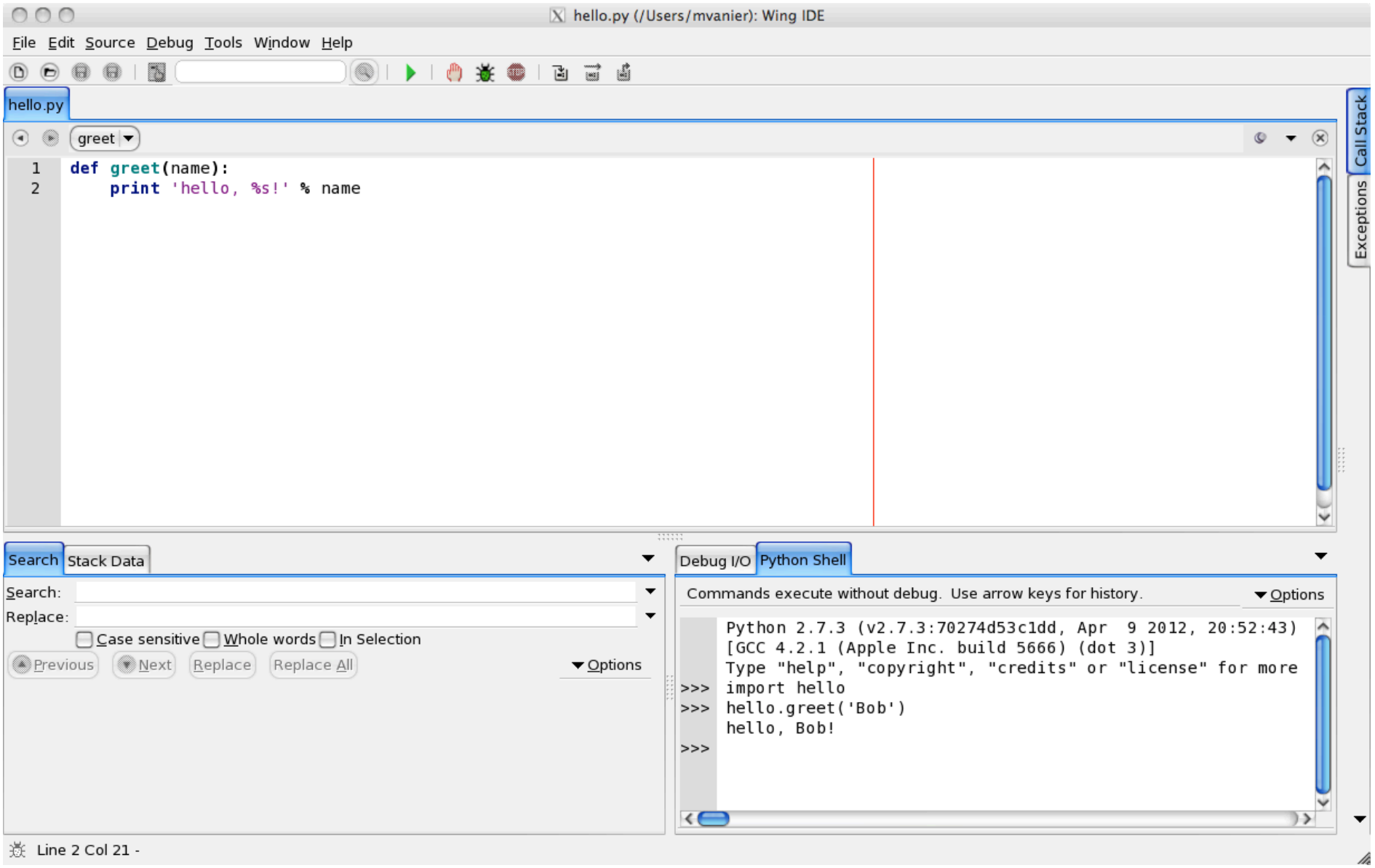
1. Write a program in a text editor (this is called *source code*)
2. Save the source code to a file on the computer's hard drive
   - (normally with a filename that ends in "`.py`")
   - *e.g.* "`myprogram.py`"
3. Execute the program by running the `python` program on the file

Writing programs in Python

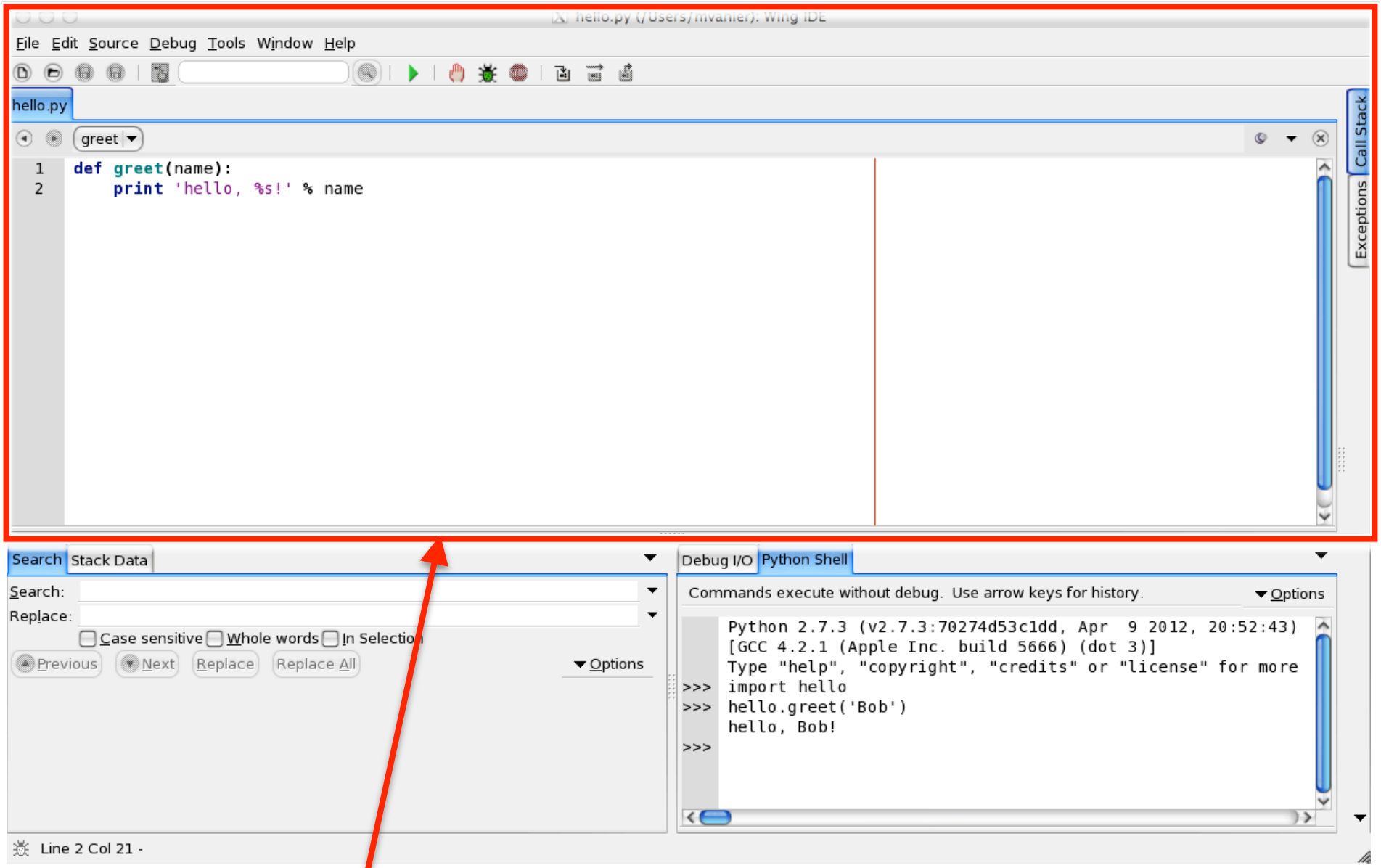
- The text editor you write the code in can be a separate program
  - and Python can be invoked from a terminal window
- Or the text editor can be "integrated" into a full program development environment (called an "IDE" or Integrated Development Environment)

Writing programs in Python

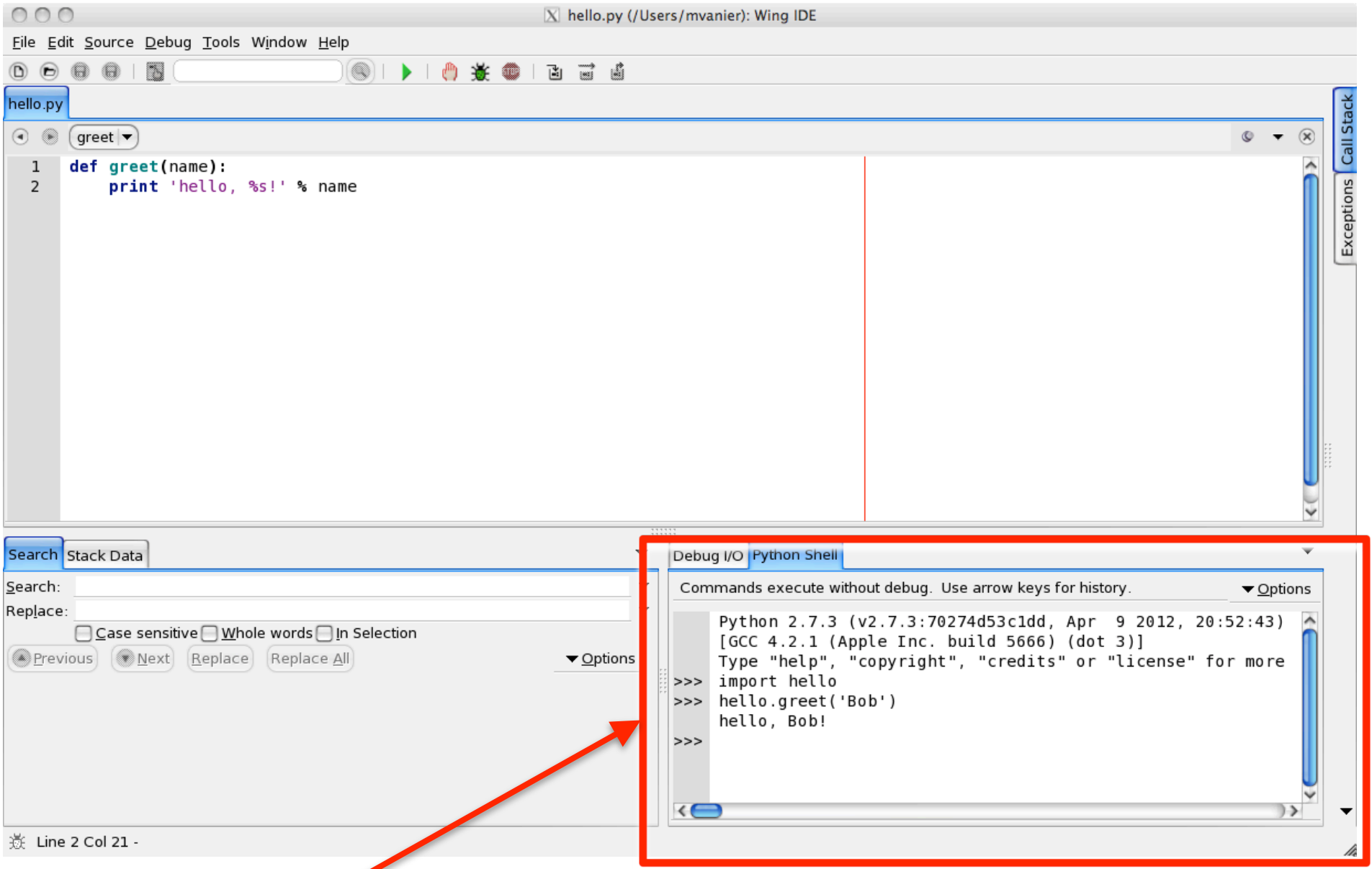
- For this course, we will use the WingIDE development environment
- Includes:
  - text editor
  - debugger
  - Python interpreter
  - links to documentation


hello.py (/Users/mvanier): Wing IDE
File Edit Source Debug Tools Window Help
hello.py
greet
1 def greet(name):
2 print 'hello, %s!' % name
Call Stack
Exceptions
Search
Stack Data
Search:
Replace:
Case sensitive
Whole words
In Selection
Previous
Next
Replace
Replace All
Options
Debug I/O
Python Shell
Commands execute without debug. Use arrow keys for history.
Options
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr 9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
Type "help", "copyright", "credits" or "license" for more
>>> import hello
>>> hello.greet('Bob')
hello, Bob!
>>>
Line 2 Col 21 -

WingIDE


File Edit Source Debug Tools Window Help
hello.py
greet
1 def greet(name):
2 print 'hello, %s!' % name
Call Stack
Exceptions
Search
Stack Data
Search:
Replace:
Case sensitive
Whole words
Previous
Next
Replace
Replace All
Options
Debug I/O
Python Shell
Commands execute without debug. Use arrow keys for history.
Options
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr 9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
Type "help", "copyright", "credits" or "license" for more
>>> import hello
>>> hello.greet('Bob')
hello, Bob!
>>>
Line 2 Col 21 -

Editor



Python shell

Writing programs in Python

- The *editor* is where you write your code
- The *Python shell* is where you can interactively experiment with Python code
- Note: Using WingIDE is not the only way to write programs in Python!
  - but very convenient, especially when debugging

The Python shell

- The Python "shell" is just an interactive interpreter of Python code
- It prints a *prompt* (>>>) and waits for you to enter Python source code
- Then it evaluates your code, prints the result, and prints another prompt, etc.
- We will use this a lot in our examples

Python as a calculator

```
>>> 1 + 1
2
>>> 2.2 * 3.4
7.48
>>> 1 + 2 * 3
7
>>> (1 + 2) * 3
9
```

The >>> prompt

- The >>> prompt is not part of the Python language
  - it's just the way that the Python shell tells you that it's waiting for more input
  - When you write Python code in files, there is no >>> and results are not automatically printed

Arithmetic expressions

- Arithmetic expressions contain numbers (operands) combined with symbols (operators) which compute values given the numbers
- Operators: `+` `-` `*` `/` etc.
- Numbers can be integers (no decimal point) or floating-point (with decimals)
  - floating-point is an approximation to real numbers

Operator precedence

- What does `1 + 2 * 3` mean?
- It could mean
  - `1 + (2 * 3)`
  - `(1 + 2) * 3`
- Computer languages have *precedence rules* to determine meaning of ambiguous cases
- Here, `*` has *higher precedence* than `+`, so the first meaning is correct

Operator precedence

- What does `1 + 2 * 3` mean?
- It could mean
  - `1 + (2 * 3)` Correct!
  - `(1 + 2) * 3`
- Computer languages have *precedence rules* to determine meaning of ambiguous cases
- Here, `*` has *higher precedence* than `+`, so the first meaning is correct

Operator precedence

- In general, `+` and `-` have lower precedence than `*` and `/`
  - and `=` is lower than either of them
- The `**` (exponentiation) operator is even higher precedence than `*` and `/`

```
>>> 2 * 3 ** 4
162
```

- Use parentheses to force a different order of evaluation if you need it

```
>>> (2 * 3) ** 4
1296
```

Variables and assignment

- Often, we want to give names to quantities
- In Python, use the **=** (assignment) operator to do this:

```
>>> pi = 3.1415926535897931
```

- From here on, **pi** stands for **3.1415...**

```
>>> 4.0 * pi
12.566370614359172
```

Variables and assignment

- Names assigned to can be reassigned:

```
>>> a = 10
>>> a
10
>>> a = 20
>>> a
20
```

Variables and assignment

- Not any sequence of letters is a valid name:

```
a = 10
b1 = 20
this_is_a_name = 30
&*%$2foo? = 40
```

- The first three are OK, the last not

Variables and assignment

- Names of variables ("identifiers") can only consist of the letters `a`-`z`, `A`-`Z`, the digits `0`-`9`, and the underscore (`_`)
- Can have as many letters as you like
- Identifiers also cannot start with a digit
  - avoids confusion with numbers
- Identifiers can't contain spaces!
- Case of letters is significant!
  - `Foo` is a different identifier than `foo`

Variables and assignment

- Can have expressions on the right-hand side of assignment statements:

```
>>> x = 5 * 3
>>> x
15
```

- The expression is terminated by the end of the line

Variables and assignment

- Can use results of previous assignments in subsequent ones:

```
>>> y = x * 5
>>> y
75
>>> z = x + y
>>> z
90
```

Variables and assignment

- Can use results of previous assignments in subsequent ones:

```
>>> z = z + 10
>>> z
100
```

- Note: expressions like `z = z + 10` are perfectly legal!

Variables and assignment

- Evaluation rule for assignment statements:
  1. Evaluate the right-hand side
  2. Assign the resulting value to the variable on the left-hand side
- This explains why `z = z + 10` works:
  - previously, `z` was `90`
  - evaluate `z + 10` → `100`
  - assign `100` to `z` (new value)
- Variables can vary!

Types

- Data in programming languages is subdivided into different "types":
  - integers: `0  -43  1001`
  - floating-point numbers: `3.1415  2.718`
  - boolean values: `True  False`
  - strings: `'foobar'  'hello, world!'`
  - and many others

Types

- Names for types
  - integers:
    - called "`int`" in Python
  - floating-point numbers:
    - called "`float`" in Python
  - boolean values:
    - called "`bool`" in Python
  - strings:
    - called "`str`" in Python

Types

- In Python, variables can hold data of any type:

```
a = 'foobar'
b1 = 10.3245
c_45 = 13579
some_boolean = True
```

Types

- In Python, the *same* variable can hold data of different types at different times:

```
>>> a = 'foobar'
>>> a
'foobar'
>>> a = 3.1415926
>>> a
3.1415926
```

Functions

- A function takes some *input data* and transforms it into *output data*
- Functions must be *defined* and then *called* with the appropriate arguments
- A few functions are built-in to Python
  - *e.g.* **`abs`**, **`max`**, **`min`**
  - ... so we don't have to define them ourselves

Functions

- Examples of function calls:

```
>>> abs(-5)
5
>>> min(5, 3)
3
>>> max(5, 3)
5
```

Functions

- Anatomy of a function call:

```
max(5, 3)
```

Functions

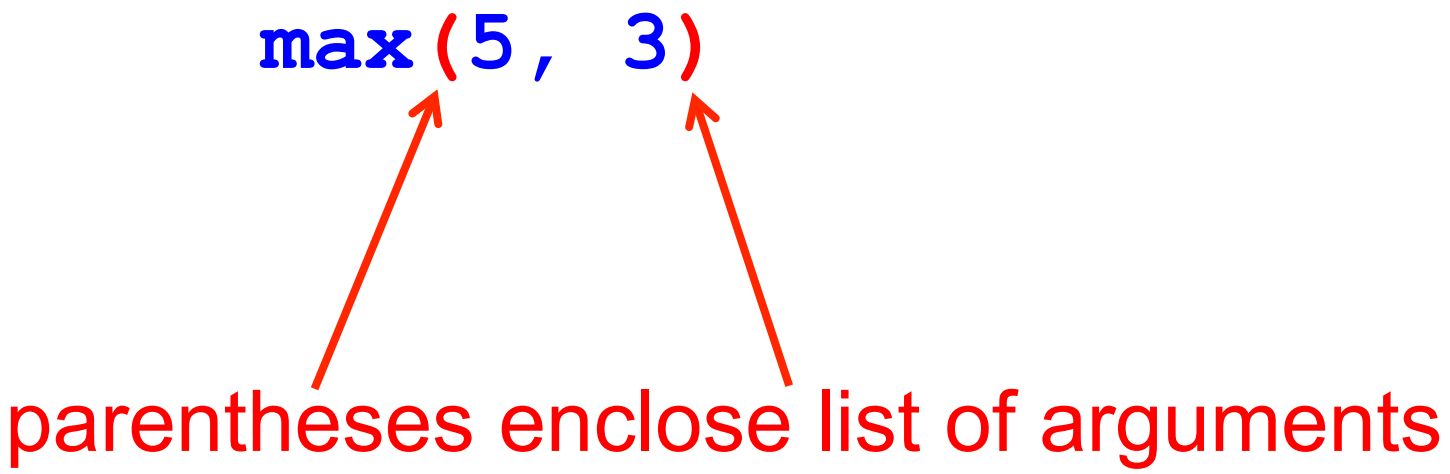
- Anatomy of a function call:

```
max(5, 3)
```

name of function

Functions

- Anatomy of a function call:



Functions

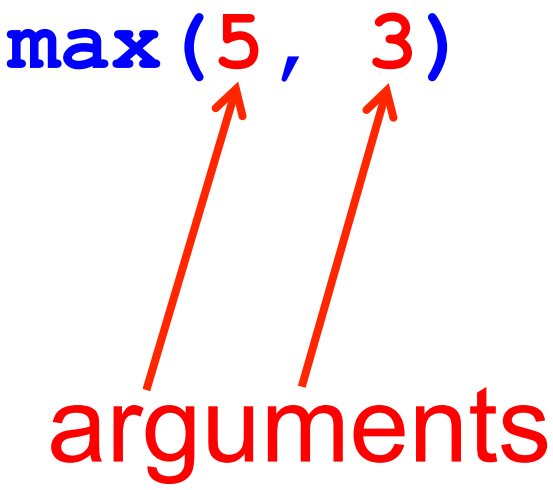
- Anatomy of a function call:

```
max(5, 3)
```

commas separate arguments

Functions

- Anatomy of a function call:



Functions

- Can have expressions as arguments:

```
>>> max(5 + 3, 8 - 6)
8
```

- Evaluation rule:
  1. Evaluate all argument expressions to get values
  2. Then evaluate the function using those values

Functions

- Can have expressions as arguments:
- `max(5 + 3, 8 - 6)`
- → `max(8, 2)`
- → `8`

Functions

- Can have function calls in expressions:
- `2 * max(5 + 3, 8 - 6) - 4`
- → `2 * max(8, 2) - 4`
- → `2 * 8 - 4`
- → `16 - 4`
- → `12`

Functions

- Can have function calls as arguments to other functions:

```
>>> max(max(5, 3), min(8, 6))
6
>>> min(2 + max(5, 3), 10)
7
```

- Evaluation rule:
  - same as before!

Function definitions

- A function *call* is done when you want to compute a particular value using that function
- If the function doesn't exist yet, you have to *define* it
- Python has a particular *syntax* to define functions
  - "syntax" means the way the language is written

Function definitions

- Example function definition in Python:

```
def double(x):
    return x * 2
```

Function definitions

- Example function definition in Python:

```
def double(x):
    return x * 2
```

- `def` is a *keyword* (reserved word) that introduces a function definition

Function definitions

- Example function definition in Python:

```
def double(x):
    return x * 2
```

- `double` is the name of the function we are defining

Function definitions

- Example function definition in Python:

```
def double(x):
    return x * 2
```

- Parentheses enclose the list of *formal arguments* to the function
  - Here, there is just one: **x**
- A colon (**:**) *must* follow the argument list!

Function definitions

- Example function definition in Python:

```
def double(x):
    return x * 2
```

- Indented lines below the `def` are the *body* of the function
  - can be just one line, or many
  - indenting is *not* optional!

Function definitions

- Example function definition in Python:

```
def double(x):
    return x * 2
```

- `return` statement is used to return the result of the function to the caller
  - `return` is another keyword in Python
  - here, `x * 2` is evaluated and returned as the result of the call to `double`

Function definitions

- Using our function definition:

```
def double(x):
    return x * 2
>>> double(42)
84
```

Function definitions

- When entering function definitions interactively into python, it looks like this:

```
>>> def double(x):
...     return x * 2
>>> double(42)
84
```

- Usually, function definitions are written directly into a file instead

Function definitions

- When entering function definitions interactively into python, it looks like this:

```
>>> def double(x):
...     return x * 2
```

- `...` is Python's *secondary prompt*
- Indicates that you're writing a function body
- Goes back to regular prompt when you're done

More function definitions

- Functions can have more than one argument:

```
def f(x, y):
    return (x * x + y * y)

>>>  f(2, 3)
13
>>>  f(2 + 1, 8 - 2)
45
```

More function definitions

- Functions can have *local variables*:

```
def f2(x, y):
    v1 = x * x
    v2 = y * y
    res = v1 + v2
    return res
```

- Here, **`v1`**, **`v2`**, and **`res`** are all local variables

More function definitions

- Functions can have *local variables*:

```
def f2(x, y):
    v1 = x * x
    v2 = y * y
    res = v1 + v2
    return res
```

- Local variables only exist for the duration of the function

More function definitions

- Body of function executed one line at a time:

```
def f2(x, y):
    v1 = x * x
    v2 = y * y
    res = v1 + v2
    return res
```

body of function

More function definitions

- Body of function executed one line at a time:

```
def f2(x, y):
    v1 = x * x
    v2 = y * y
    res = v1 + v2
    return res
```

- `x` is the value passed in when function is called

More function definitions

- Body of function executed one line at a time:

```
def f2(x, y):
    v1 = x * x
    v2 = y * y
    res = v1 + v2
    return res
```

- Ditto for `y`

More function definitions

- Body of function executed one line at a time:

```
def f2(x, y):
    v1 = x * x
    v2 = y * y
    res = v1 + v2
    return res
```

More function definitions

- Body of function executed one line at a time:

```
def f2(x, y):
    v1 = x * x
    v2 = y * y
    res = v1 + v2
    return res
```

More function definitions

- Body of function executed one line at a time:

```
def f2(x, y):
    v1 = x * x
    v2 = y * y
    res = v1 + v2
    return res
```

- Then result is returned

Local variables are local!

```
def f2(x, y):
    ... <as before> ...
    return res
>>> f2(10, 20)
500
>>> res
Traceback (most recent call last):
  File "<string>", line 1, in <string>
NameError: name 'res' is not defined
```

Comments

- Comments are lines in a source code file that are "notes to the reader"
  - Python just ignores them
- Comments start with a **#** and go to the end of the line:

```
# This is a comment.
```

- Comments are one way to document your code
  - we'll see others as we go along

Wrap-up

- Action items:
  - Go to web site and enroll in it
  - Read important sections
  - Fill out lab section preference questionnaire when it's available
- Next time:
  - strings and string processing