The Python Language
(Part 5)

Copyright © 2021 by
Robert M. Dondero, Ph.D.
Princeton University

1

Objectives

- We will cover:
  - A subset of Python...
  - That is appropriate for COS 333...
  - Through example programs

2

Objectives

Continued from Part 4

[see slide]

Agenda

- **Prelim: character and string encodings**
- Files
- The "with" statement
- Arrays
- Associative arrays
- Iterable classes
- Variadic functions/methods

3

Character Encodings

Examples:

Encoding	Fixed/Variable Width	Bytes
ASCII	Fixed	1 (7 bits)
Latin-1	Fixed	1
UCS-2	Fixed	2
UCS-4	Fixed	4
UTF-8	Variable	1, 2, 3, or 4
UTF-16	Variable	2 or 4

4

Character Encodings

As you know, any modern computer deals with all data as bytes
    1 byte = 8 bits
To deal with **characters**, any computer must use a **character encoding**
A character encoding maps each character to the byte(s) which represent it

There are many character encodings

[see slide]

ASCII is a fixed-width encoding in which each character is mapped to 1 byte
…

Character Encodings

Character	ASCII	Latin-1	UCS-2	UCS-4	UTF-8	UTF-16
space	20	20	0020	00000020	20	0020
!	21	21	0021	00000021	21	0021
0	30	30	0030	00000030	30	0030
A	41	41	0041	00000041	41	0041
a	61	61	0061	00000061	61	0061
a with grave		e0	00e0	000000e0	e0	00e0
Greek small pi			03c0	000003c0	cf80	03c0
Double prime			2033	00002033	e280b3	2033
Aegean number 2000				00010123	f09084a3	d800dd23

5

Character Encodings

- In Python:

  - `'abc'`
    - Object of class `str`
    - Internal encoding is
      - **Latin-1** if possible, or if not…
      - **UCS-2** if possible, or if not…
      - **UCS-4**
    - But best to treat as an ADT

6

Character Encodings

[see slide]

It's best to pretend you don't know how str objects are stored
It's best to treat str as an abstract data type

Character Encodings

- In Python:

  - `b'\x61\x62\x63'`
    - Object of class `bytes`
    - Array of bytes
    - No internal encoding

Character Encodings

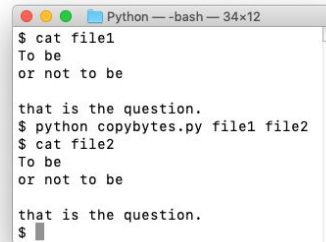Python also has a related data type; bytes

[see slide]

We'll use bytes objects when doing network programming (soon)

Agenda

- Prelim: character and string encodings
- **Files**
- The "with" statement
- Arrays
- Associative arrays
- Iterable classes
- Variadic functions/methods

Files

- See **copybytes.py**
  - The job:
    - Accept names of input and output files as command-line args
    - Read some bytes from input file; write those bytes to output file; repeat until end of input file

```
$ cat file1
To be
or not to be

that is the question.
$ python copybytes.py file1 file2
$ cat file2
To be
or not to be

that is the question.
$
```

9

Files

[see slide]

Notes:

    Command-line arguments
            sys.argv is a list
                    sys.argv[0] is the name of the program
                    sys.argv[1] is the first command-line arg
                    sys.argv[2] is the second command-line arg
                    …
            len() function is defined in __builtin__ module
                    len(somelist) => somelist.__len__() => number of elements in
                    somelist

    File I/O
            in_file = open(in_file_name, mode='rb')
                    in_file.read(MAX_BYTE_COUNT) reads up to
                    MAX_BYTE_COUNT bytes from in_file, returns bytes object
            out_file = open(out_file_name, mode='wb')
                    out_file.write(byte_array) writes bytes from bytes object b to
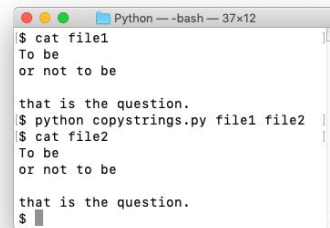   out_file

            in_file.close()

This program: to be polite; other programs: necessary

Files

- See **copystrings.py**
  - The job:
    - Accept names of input and output files as command-line args
    - Read a line (as a string) from input file; write the line (as a string) to output file, repeat until end of input file

```
$ cat file1
To be
or not to be

that is the question.
$ python copystrings.py file1 file2
$ cat file2
To be
or not to be

that is the question.
$
```

10

Files

[see slide]

Notes

File I/O

in_file = open(inFileName, mode='r', encoding='utf-8')
in_file contains strings encoded as UTF-8
in_file .readline() reads bytes comprising 1 line from in_file ,
decodes to str object, returns str object

out_file = open(outFileName, mode='w', encoding='utf-8')
out_file should contain strings encoded as UTF-8
out_file.write(line) *encodes* line to bytes and writes to out_file

Implicit call of readline() via for statement!
for statement can iterate over any Iterable object – even a file!
More on that later

Files

- copybytes.py vs copystrings.py
  - copybytes.py: faster
  - copystrings.py: more flexible

11

Files

copybytes.py vs. copystrings.py

     copybytes.py: faster
        No decoding or encoding

     copystrings.py: more flexible
        Data can be manipulated as Strings
           Sorted, concatenated, …

Agenda

- Prelim: character and string encodings
- Files
- **The "with" statement**
- Arrays
- Associative arrays
- Iterable classes
- Variadic functions/methods

The "with" Statement

- Consider copystrings.py
- **Problem:**

```
in_file = open(…)
…
…                Exception thrown =>
…                File never closed
in_file.close()
```

13

The "with" Statement

[see slide]

If an exception is thrown in the area between open() and close(), then close() is never called

In copystrings.py: no problem
		The process will exit soon anyway

In other programs: maybe a problem

In general, whenever a process opens a file it should close the file, no matter what

The "with" Statement

- **Solution 1:**
  - `try…finally` statement

```
in_file = open(…)
try:
    …
    …          If this is entered...
    …
finally:                Then this certainly
    in_file.close()     will be executed "on
                        the way out"
```

14

The "with" Statement

[see slide]

If the try statement is entered, then
in_file.close() certainly will be executed on the way out
whether or not the statements within the try statement throw an exception

The "with" Statement

- See **copystringsfinally.py**
  - The job
    - Same as copystrings.py, but…
    - Cleans up after exceptions properly…
    - By using `try…finally`

15

The "with" Statement

[see slide]

Awkward; error prone; hard to read
Common!

The "with" Statement

- **Solution 2:**
  - `with` statement

```
with open(…) as in_file:
    …
    …
    …
```

If this is entered, then
in_file.close() certainly will
be executed on the way out

The "with" Statement

[see slide]

New to Python 2.6 and 3.0
If the with statement is entered, then
f.close() certainly will be executed automatically on the way out
whether or not the statements within the with statement throw an exception

- See **copystringswith.py**
  - The job
    - Same as copystrings.py, but…
    - Cleans up after exceptions properly…
    - By using `with`

The "with" Statement

[see slide]

Less awkward; less error prone, easier to read

Generalizing:
When you open a file, you should make sure you close it
As we'll see…
When you open a network socket, you should make sure you close it
When you open a DB connection, you should make sure you close it
When you open a DB cursor, you should make sure you close it
When you acquire a lock on an object, you should make sure you release it

In those situations, you should use the with statement

Agenda

- Prelim: character and string encodings
- Files
- The "with" statement
- **Arrays**
- Associative arrays
- Iterable classes
- Variadic functions/methods

18

Arrays

- Generic term: **array**
- Python term: *list*
  - A dynamically expanding (doubling) array of object references

Arrays

Generic term: **array**
> Wikipedia: "A data structure consisting of a collection of elements…, each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula."

Python term: **list**
> Python offers a list class
> Not a linked list!
> A dynamically expanding (doubling) array of object references

Arrays

- See **linesort1.py**
  - The job:
    - Read lines from stdin
    - Sort lines in lexicographical order
    - Write lines to stdout

Arrays

CPU time: 0.20 sec

- See **linesort1.py** (cont.)

21

Arrays

[see slide]

Notes:

`list`: dynamically sized heterogeneous array

Has `sort()` method

Can iterate over a file or list

Performance:

Python: CPU time to sort Bible.txt: 0.20 sec!!!

Faster than expected

Java: Time to sort Bible.txt: 0.40 sec

Arrays

CPU time: 0.59 sec

- See **linesort2.py**
  - The job:
    - Same as linesort1.py
    - Uses custom mergesort

22

Arrays

[see slide]

Notes:

       Uses custom mergesort

       Performance:

              CPU time to sort Bible.txt: 0.59 sec

                     Much slower than linesort1.py

                     Slower than Java version

Aside: Python Efficiency

- **Question**
  - Why is linesort1.py so fast?

23

Aside: Python Efficiency

- **Answer**
  - `sort()` method in `list` class
    was written in C

- **Generalizing**
  - Need efficiency =>
    use standard functions/methods

24

Aside: Python Efficiency

Answer

      sort() method in list class was written in C, not Python
            And is present at run-time in machine language!
      All of the Python standard functions and classes were written in C
      That's why the standard functions and methods are so fast

Generalizing

      Don't count on Python handling **your** functions/methods so efficiently!
      If you want/need speed, try to avoid doing work using your own
      functions/methods
      Constrain yourself to using the standard functions/methods as much as you can

Commentary

      Python is awkward for a CS1 + CS2 course sequence
      Will Python ever replace Java as the most popular language for CS1 courses?

Agenda

- Prelim: character and string encodings
- Files
- The "with" statement
- Arrays
- **Associative arrays**
- Iterable classes
- Variadic functions/methods

25

Associative Arrays

- Generic term: **associative array**
- Python term: ***dict***
  - An associative array implemented as a hash table

26

Associative Arrays

Generic term: **associative array**
> Wikipedia: "An abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection"
> Typical implementation: red-black tree, hash table, …

Python term: **dict**
> Python offers a dict class
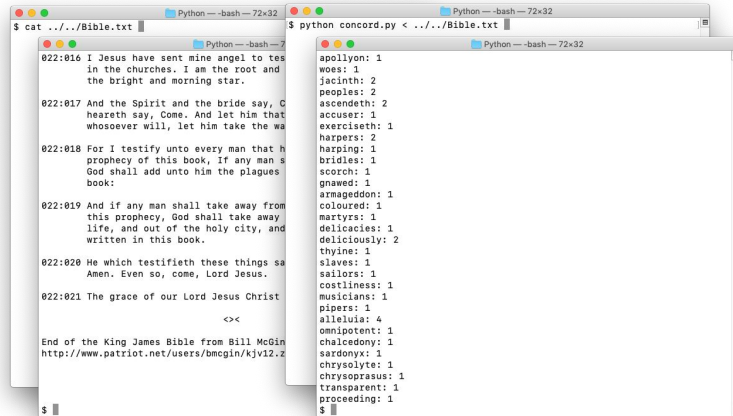> A dict object is a "dictionary"
> An associative array implemented as a hash table

Associative Arrays

- See **<u>concord.py</u>**
  - The job:
    - Read words from `stdin`
    - Write a *concordance* to `stdout`
      - Each word and its occurrence count

Associative Arrays

- See **<u>concord.py</u>** (cont.)

Associative Arrays

CPU time: 0.39 sec

- See **concord.py** (cont.)

```
● ● ●                    📁 Python — -bash — 72×22
$ python concord.py < ../../Bible.txt | sort -n -k 2 -r | head -20
the: 63937
and: 51699
of: 34624
to: 13569
that: 12913
in: 12670
he: 10420
shall: 9838
unto: 8997
for: 8973
i: 8854
his: 8474
a: 8178
lord: 7965
they: 7376
be: 7013
is: 6993
him: 6659
not: 6596
them: 6431
$
```

30

Associative Arrays

[see slide]

Notes:
    `dict` object
        An associative array
        `in` operator
        Indexing using `[]` operator
        Iteration using `for` statement

    Regular expressions
        `[a-z]`
            Used to find a lower-case letter
        `[a-z]+`
            Used to find a sequence of 1 or more lower-case letters
    Raw strings
        `r'[a-z]+'`
        Backslash is not interpreted as an escape character

    Performance
        Python: Time to handle Bible.txt: 0.39 sec
        Java: Time to handle Bible.txt: 0.47sec

Generalizing...

Agenda

- Prelim: character and string encodings
- Files
- The "with" statement
- Arrays
- Associative arrays
- **Iterable classes**
- Variadic functions/methods

30

Iterable Classes

- Python iterable classes
  - `str`
    - An immutable sequence of characters
  - `bytes`
    - An immutable sequence of bytes
  - `list`
    - A sequence of object references
  - `tuple`
    - An immutable sequence of object references
  - `set`
    - A group of object references that contains no duplicates
  - `dict`
    - An associative array implemented as a hash table
  - file
    - A persistent sequence of bytes

Iterable Classes

Creating iterable objects

```
str
    str1 = 'hi'
    str2 = "hi"
    str3 = r'hi'

bytes
    bytes1 = b'hi'
    bytes2 = b"hi"
    bytes3 = rb'hi'

list
    list1 = [obj1, obj2, ...]

tuple
    tuple1 = (obj1, obj2, ...)
    tuple2 = (obj1, )              ← hack
```

32

Iterable Classes

[see slide]

Shows the syntax for creating iterable objects

Iterable Classes

Creating iterable objects (cont.)

```
set
    set1 = {obj1, obj2, ...}
    # tests for object ref equality

dict
    dict1 = {keyobj1:valueobj1, keyobj2:valueobj2, ...}

file
    fileobj = open('filename', mode='somemode')
        # somemode: r, rb, w, wb, ...
```

Iterable Classes

[see slide]

Shows the syntax for creating iterable objects

Iterable Classes

Some str Methods	
`str1 = str2.__add__(str3)`	`# str1 = str2 + str3`
`bool1 = str1.__eq__(str2)`	`# bool1 = str1 == str2`
`bool1 = str1.__ne__(str2)`	`# bool1 = str1 != str2`
`bool1 = str1.__lt__(str2)`	`# bool1 = str1 < str2`
`bool1 = str1.__gt__(str2)`	`# bool1 = str1 > str2`
`bool1 = str1.__le__(str2)`	`# bool1 = str1 <= str2`
`bool1 = str1.__ge__(str2)`	`# bool1 = str1 >= str2`
`int1 = str1.__len__()`	`# int1 = len(str1)`
`str1 = str2.__getitem__(int1)`	`# str1 = str2[int1]`
`bool1 = str1.__contains__(str2)`	`# bool1 = str2 in str1`

34

Iterable Classes

For reference

[see slide]

Iterable Classes

Some str Methods (cont.)
`bool1 = str1.startswith(str2)`
`bool1 = str1.endswith(str2)`
`bool1 = str1.isspace()`
`bool1 = str1.isalnum()`
`bool1 = str1.isalpha()`
`bool1 = str1.isdecimal()`
`bool1 = str1.isdigit()`
`bool1 = str1.islower()`
`bool1 = str1.isnumeric()`
`bool1 = str1.isupper()`
`str1 = str2.upper()`
`str1 = str2.lower()`

35

Iterable Classes

For reference

[see slide]

Iterable Classes

Some str Methods (cont.)
`list1 = str1.split(str2)`
`str1 = str2.replace(str3, str4)`
`str1 = str2.strip()`
`str1 = str2.lstrip()`
`str1 = str2.rstrip()`
`int1 = str1.find(str2)`
`int1 = str1.rfind(str2)`
`str1 = str2.join(list1)` (See note)
`bytes1 = str1.encode(encoding)`

Note:
```
'/'.join(['hello', 'there', 'world']) =>
'hello/there/world'
```

36

Iterable Classes

For reference

[see slide]

Iterable Classes

Some list Methods	
`list1 = list2.__add__(list3)`	`# list1 = list2 + list3`
`bool1 = list1.__contains__(obj1)`	`# `**`bool1 = obj1 in list1`**
`list1.__delitem__(int1)`	`# del(list1[int1])`
`obj1 = list1.__getitem__(int1)`	`# `**`obj1 = list1[int1]`**
`list1.__iadd__(list2)`	`# list1 += list2`
`int1 = list1.__len__()`	`# `**`int1 = len(list1)`**
`list1.__setitem__(int1, obj1)`	`# `**`list1[int1] = obj1`**

37

Iterable Classes

For reference

[see slide]

Iterable Classes

Some list Methods (cont.)
list1.append(obj1)
list1.clear()
list1 = list2.copy()
int1 = list1.index(obj1)
list1.insert(obj1, int1)
obj1 = list1.pop()
list1.remove(obj1)
list1.reverse()
list1.sort()

Iterable Classes

For reference

[see slide]

Iterable Classes

Some dict Methods
`bool1 = dict1.__contains__(obj1) # `**`bool1 = obj1 in dict1`**
`dict1.__delitem(obj1)            # del(dict1[obj1])`
`obj1 = dict1.__getitem__(obj2)   # `**`obj1 = dict1[obj2]`**
`int1 = dict1.__len__()           # `**`int1 = len(dict1)`**
`dict1.__setitem__(obj1, obj2)    # `**`dict1[obj1] = obj2`**
`dict1.clear()`
`dict1 = dict2.copy()`
`list1 = dict1.keys()`
`list1 = dict1.items()`
`list1 = dict1.values()`

39

Iterable Classes

For reference

[see slide]

Iterable Classes

Some file Methods
file1.close()
file1.flush()
str1 = file1.read()
bool1 = file1.readable()
str1 = file1.readline()
list1 = file1.readlines()
fool1 = file1.writable()
file1.write(str1)
file1.writelines(list1)

40

Iterable Classes

For reference

[see slide]

Iterable Classes

For more information:

```
$ python
>>> help(str)
>>> help(bytes)
>>> help(tuple)
>>> help(list)
>>> help(set)
>>> help(dict)
>>> quit()
$
```

41

Agenda

- Prelim: character and string encodings
- Files
- The "with" statement
- Arrays
- Associative arrays
- Iterable classes
- **Variadic functions/methods**

42

Variadic Functions/Methods

- *Variadic* function/method:
  - A function/method that can be called with a variable number of arguments
  - Example: `printf()` in C
    - `printf("hello");`
    - `printf("The answer is %d", 5);`
    - `printf("The answers are %d and %d, 5, 10);`

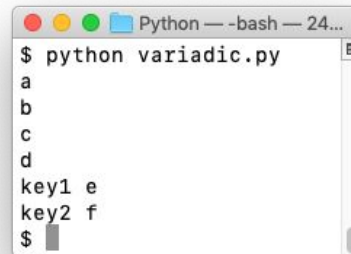# Variadic Functions/Methods

- **Question**
  - How to define variadic functions/methods in Python?
- **Answer 1**
  - Default parameter values
    - Already described
- **Answer 2**
  - `*args` and `**kwargs`…

44

Variadic Functions/Methods

- See **variadic.py**
  - The job:
    - Nonsensical
    - Uses `*args`
      and `**kwargs`

```
$ python variadic.py
a
b
c
d
key1 e
key2 f
$
```

45

Variadic Functions/Methods

- See **variadic.py** (cont.)
  - Notes:
    - Argument/parameter matching

Parameter	Referenced Object
i	'a'
j	'b'
args	['c', 'd']
kwargs	{'key1':'e', 'key2':'f'}

Variadic Functions/Methods

[see slide]

The body of main() simply prints the parameter values to demonstrate that argument/parameter matching the occurred

Easy alternative: pass a list and/or a dict

But this mechanism does make the code more concise
And is used often in standard functions/methods
So it's worth knowing

Python Commentary

From r@google.com Sun Jan 1 16:18:21 2006
Date: Sun, 1 Jan 2006 13:18:08 -0800
From: Rob 'Commander' Pike <r@google.com>
To: Brian Kernighan <bwk@CS.Princeton.EDU>

python is a very easy language. i think it's actually a
good choice for some things. awk is perfect for a line
or two, python for a page or two. both break down
badly when used on larger examples, although python
users utterly refuse to admit its weaknesses for
large-scale programming, both in syntax and efficiency.

 -rob

What do **you** think?

Python Commentary

[see slide]

Could sell this on EBay!

I'm a big fan of C, Python, and Java
But I like them for different jobs!
 C: systems programming
 Python: application programming, small applications (5 programmers, a few months)
 Java: application programming, big applications (25 programmers, a few years)
Rhetorically: What do you think?

Summary

- We have covered these aspects of Python:
  - Files
  - Arrays
  - Associative arrays
  - Iterable classes
  - Variadic functions/methods

48

Summary

Summary of this lecture

[see slide]

Summary

- We have covered:
  - Subset of Python...
  - That is appropriate for COS 333...
  - Through example programs

49

Summary

Summary of the Python sequence of lectures

[see slide]

Appendices

- **Appendix 1**: Regular expressions
- **Appendix 2**: The Python Debugger

Appendix 1:
Regular Expressions

Regular Expressions

- Used widely
  - Java (string manipulation)
  - Python (string manipulation)
  - Unix `grep` command (file searching)
  - Bash shell (filename wildcards)
  - SQL `like` clauses (querying databases)
    - See upcoming *Databases* lectures
  - …

52

Regular Expressions

```
RE                  Matches
thing               thing anywhere in string
^thing              thing at beginning of string
thing$              thing at end of string
^thing$             string that contains only thing
^                   any string, even empty
^$                  empty string
.                   non-empty, i.e. the first char in string
thing.$             thing plus any char at end of string
thing\.$            thing. at end of string
\\thing\\           \thing\ anywhere in string
[tT]hing            thing or Thing anywhere in string
thing[0-9]          thing followed by one digit
thing[^0-9]         thing followed by a non-digit
thing[0-9][^0-9]    thing followed by digit, then non-digit
thing1.*thing2      thing1 then any (or no) text then thing2
^thing1.*thing2$    thing1 at beginning and thing2 at end
```

Thanks to Prof. Brian Kernighan

53

Regular Expressions

- What do these match?
  - `a.*e.*i.*o.*u`
    - Try with `grep` command and /usr/share/dict/words file
  - `^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*$`
    - Try with `grep` command and /usr/share/dict/words file

Thanks to Prof. Brian Kernighan

54

Regular Expressions

- Implementations vary
  - See *Mastering Regular Expressions* (Jeffrey Friedl) book
    - ~500 pages!
- In Python…

55

Regular Expressions

```
RE      Matches
X       the character X, except for metacharacters
\X      the character X, where X is a metacharacter
.       any character except \n
            (use DOTALL as argument to compile() to match \n too)
^       start of string
$       end of string
XY      X followed by Y
X*      zero or more cases of X (X*? is the same, but non-greedy)
X+      one or more cases of X (X+? is the same, but non-greedy)
X?      zero or one case of X (X?? is the same, but non-greedy)
[...]   any one of ...
[^...]  any character other than ...
[X-Y]   any character in the range X through Y
X|Y     X or Y
(...)   ..., and indicates a group
```

Precedence: * + ? higher than concatenation, which is higher than |

56

Regular Expressions

```
RE  Matches
\t  tab
\v  vertical tab
\n  newline
\r  return
\f  form feed
\a  alert
\e  escape
\\  backslash
\A  empty string at start of given string
\b  empty string, but only at start or end of a word
\B  empty string, but not at start or end of a word
\d  a digit
\D  a non-digit
\s  a white space character, that is, [\t\n\r\f\v]
\S  a non-white space character
\w  an alphanumeric character, that is, [a-zA-Z0-9_]
\W  a non-alphanumeric character
\Z  the empty string at the end of the given string
```
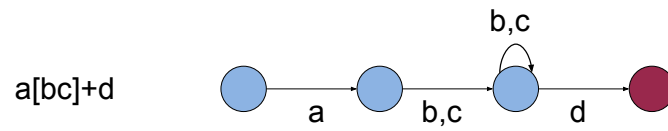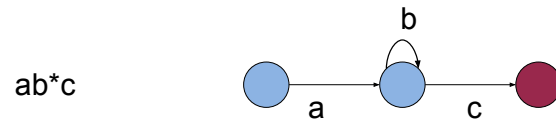
57

Regular Expressions

- What kinds of strings do these regular expressions match?
  - `[-+]?([0-9]+\.?[0-9]*|\.[0-9]+)([Ee][-+]?[0-9]+)?`
  - `/\*.*?\*/`   (use with `DOTALL`)
    - Why the question mark?
    - Why `DOTALL`?
- Commentary: Regular expressions are write-only!!!

58

Regular Expressions

- Some theory:
  - Regular expressions have the same power as **deterministic finite state automata (DFAs)**
  - A regular expression defines a **regular language**
  - A DFA also defines a regular language

59

Regular Expressions

ab*c

a[bc]+d

Appendix 2:
The Python Debugger

The Python Debugger

- `pdb` debugger is bundled with Python

- To use `pdb`:
  - `$ python -m pdb` *file*`.py`

The Python Debugger

Some `pdb` Commands
- `help`
- `break` *functionOrMethod*
- `break` *filename*:*linenum*
- `run`
- `list`
- `next`
- `step`
- `continue`
- `print` *expr*
- `where`
- `quit`

Note similarities with `gdb`

63

The Python Debugger

- Common commands have abbreviations: h, b, r, l, n, s, c, p, w, q
- Blank line means repeat the same command
- Beware: Cannot easily read from `stdin`

64