

---

---

Python Concepts  
The Misunderstood Programming Language

---

---

CUSTOMIZED FOR INFORMATION TECHNOLOGY INSTITUTE

EGYPT, 2013

COLLECTED BY

**HAITHAM A. EL-GHAREEB**  
*The University of Mansoura  
Egypt*

HELGHAREEB@MANS.EDU.EG  
[HTTP://WWW.HELGHAREEB.ME](http://WWW.HELGHAREEB.ME)  
2013



-= To my Family =-  
*Haitham A. El-Ghareeb, Ph.D.*

March, 2013



# Acknowledgment

First and foremost, I thank ALLAH for giving me incentive to go on and finish this work, and every other work have been finished and giving me glimpses of heavenly hope in the darkest times.

I would like to thank Eng. Ayman Lotfy for the cooperation, generosity, and support.

My Family, you are the people whom without I could not have continued the journey to the day I witness this book alive. Thanks for always being there to support me, every single moment, with everything you can, when I was about to lose hope and trust in me being able to continue through the darkness, trying to reach the goal through long nights of mess, you were there, through long distances you were pushing me forward by your support, encouragement, guidance, and help. Thanks to ALLAH, I am really blessed with amazing family like you. I Love You all!

*Haitham A. El-Ghareeb*  
March, 2013



# Contents

<b>Acknowledgment</b>	<b>v</b>
<b>Preface</b>	<b>xvii</b>
<b>1 Teach yourself Programming in 10 Years</b>	<b>1</b>
1.1 Why is everyone in such a rush? . . . . .	1
1.2 Teach Yourself Programming in Ten Years . . . . .	2
1.3 So You Want to be a Programmer . . . . .	3
<b>2 Programming Languages Are Not The Same</b>	<b>5</b>
2.1 Basis for Comparing Programming Languages . . . . .	5
2.1.1 Object Orientation . . . . .	5
2.1.2 Static vs. Dynamic Typing . . . . .	6
2.1.3 Generic Classes . . . . .	6
2.1.4 Inheritance . . . . .	7
2.1.5 Feature Renaming . . . . .	8
2.1.6 Method Overloading . . . . .	8
2.1.7 Operator Overloading . . . . .	8
2.1.8 Higher Order Functions & Lexical Closures . . . . .	9
2.1.9 Garbage Collection . . . . .	9
2.1.10 Uniform Access . . . . .	10
2.1.11 Class Variables/Methods . . . . .	11
2.1.12 Reflection . . . . .	11
2.1.13 Access Control . . . . .	11
2.1.14 Design by Contract . . . . .	11
2.1.15 Multithreading . . . . .	12
2.1.16 Regular Expressions . . . . .	12
2.1.17 Pointer Arithmetic . . . . .	12
2.1.18 Language Integration . . . . .	12
2.1.19 Built-In Security . . . . .	13
2.2 Comparing based on Performance and Optimization . . . . .	13
<b>3 Python Tutorial</b>	<b>15</b>
<b>4 Advanced Python</b>	<b>167</b>
<b>5 Object Oriented Python Concepts</b>	<b>297</b>

<b>6 Python Magic</b>	<b>317</b>
<b>7 Python GUI</b>	<b>351</b>
<b>8 Network Programming</b>	<b>367</b>
<b>9 Python Facts</b>	<b>385</b>
9.1 Zip . . . . .	385
9.2 List Comprehensions Support Conditions . . . . .	385
9.3 Expanding arrays to function arguments . . . . .	385
9.4 Ordered and Named Function Arguments . . . . .	386
9.5 Functions as objects . . . . .	386
9.6 Mutability . . . . .	386
9.7 Change mutable lists using another name . . . . .	387
9.8 Factorial Function . . . . .	387
9.9 String formatting using dictionaries . . . . .	387
9.10 Never Define One Line Functions . . . . .	387
9.11 Number from Digits . . . . .	388
9.12 Generator Functions . . . . .	388
9.13 Parsing REST API . . . . .	388
9.14 Sets for finding unions and intersections . . . . .	389
9.15 Locals and strings . . . . .	389
9.16 Encoding & Decoding . . . . .	389
9.17 Set Comprehensions . . . . .	389
9.18 dict() and zip() make for a powerful combination . . . . .	390
9.19 Bools inherit from int . . . . .	390
9.20 The Zen of Python . . . . .	390
<b>10 Python Glossary</b>	<b>391</b>
10.1 List . . . . .	391
10.2 Slice . . . . .	391
10.3 Variables . . . . .	392
10.4 Functions . . . . .	392
10.5 Tuples . . . . .	393
10.6 List Comprehensions . . . . .	393
10.7 Sets . . . . .	393
10.8 Dictionaries . . . . .	393
10.9 Strings . . . . .	394
10.10 The len() Function . . . . .	394
10.11 Single Line Comments . . . . .	394
10.12 Multi-line Comments . . . . .	394
10.13 Print . . . . .	395
10.14 The range() Function . . . . .	395
10.15 For Loops . . . . .	395
10.16 While Loops . . . . .	396
10.17 The str() Function . . . . .	396
10.18 »> . . . . .	396

10.19 . . . . .	397
10.20 2to3 . . . . .	397
10.21 abstract base class . . . . .	397
10.22 argument . . . . .	397
10.23 attribute . . . . .	398
10.24 BDFL . . . . .	398
10.25 bytecode . . . . .	398
10.26 class . . . . .	398
10.27 classic class . . . . .	398
10.28 coercion . . . . .	398
10.29 complex number . . . . .	399
10.30 context manager . . . . .	399
10.31 CPython . . . . .	399
10.32 decorator . . . . .	399
10.33 descriptor . . . . .	400
10.34 dictionary . . . . .	400
10.35 docstring . . . . .	400
10.36 duck-typing . . . . .	400
10.37 EAFP . . . . .	400
10.38 expression . . . . .	400
10.39 extension module . . . . .	401
10.40 file object . . . . .	401
10.41 file-like object . . . . .	401
10.42 finder . . . . .	401
10.43 floor division . . . . .	401
10.44 function . . . . .	401
10.45 __future__ . . . . .	402
10.46 garbage collection . . . . .	402
10.47 generator . . . . .	402
10.48 generator expression . . . . .	402
10.49 GIL . . . . .	403
10.50 global interpreter lock . . . . .	403
10.51 hashable . . . . .	403
10.52 IDLE . . . . .	403
10.53 immutable . . . . .	403
10.54 integer division . . . . .	404
10.55 importer . . . . .	404
10.56 interactive . . . . .	404
10.57 interpreted . . . . .	404
10.58 iterable . . . . .	404
10.59 iterator . . . . .	405
10.60 key function . . . . .	405
10.61 keyword argument . . . . .	405
10.62 lambda . . . . .	405
10.63 LBYL . . . . .	406
10.64 list . . . . .	406

10.65list comprehension . . . . .	406
10.66loader . . . . .	406
10.67mapping . . . . .	406
10.68metaclass . . . . .	406
10.69method . . . . .	407
10.70method resolution order . . . . .	407
10.71MRO . . . . .	407
10.72mutable . . . . .	407
10.73named tuple . . . . .	407
10.74namespace . . . . .	407
10.75nested scope . . . . .	408
10.76new-style class . . . . .	408
10.77object . . . . .	408
10.78parameter . . . . .	408
10.79positional argument . . . . .	409
10.80Python 3000 . . . . .	409
10.81Pythonic . . . . .	409
10.82__slots__ . . . . .	410
10.83sequence . . . . .	410
10.84slice . . . . .	410
10.85special method . . . . .	410
10.86statement . . . . .	410
10.87struct sequence . . . . .	410
10.88type . . . . .	411
10.89view . . . . .	411
10.90virtual machine . . . . .	411
10.91Zen of Python . . . . .	411
<b>11 Project</b>	<b>413</b>
<b>12 List Methods</b>	<b>433</b>
<b>13 String Formattig</b>	<b>435</b>
<b>14 Built-in Modules</b>	<b>439</b>
<b>I Python Labs</b>	<b>443</b>
<b>15 Python Syntax</b>	<b>445</b>
15.1 Variables and Data Types . . . . .	445
15.1.1 Variables . . . . .	445
15.1.2 Data Types . . . . .	446
15.1.3 You've Been Reassigned . . . . .	446
15.2 Whitespace and Statements . . . . .	447
15.2.1 What's a Statement? . . . . .	447
15.2.2 Whitespace Means Right Space . . . . .	447

15.2.3 A Matter of Interpretation . . . . .	448
15.3 Comments . . . . .	448
15.3.1 Single Line Comments . . . . .	448
15.3.2 Multi-Line Comments . . . . .	448
15.4 Math Operations . . . . .	449
15.4.1 Arithmetic Operators . . . . .	449
15.4.2 Subtraction . . . . .	449
15.4.3 Multiplication . . . . .	449
15.4.4 Division . . . . .	450
15.4.5 Exponentiation . . . . .	450
15.4.6 Modulo . . . . .	450
15.5 Review . . . . .	450
15.5.1 Bringing It All Together . . . . .	450
15.6 Project: Tip Calculator . . . . .	451
15.6.1 Your Favorite Meal . . . . .	451
15.6.2 The Tax . . . . .	451
15.6.3 The Tip . . . . .	452
15.6.4 Reassign in a Single Line . . . . .	452
15.6.5 Second Verse, Same as the First . . . . .	452
<b>16 Strings and Console Output</b> . . . . .	<b>453</b>
16.1 Strings . . . . .	453
16.1.1 Step One: Strings . . . . .	453
16.1.2 Step Two: Things . . . . .	453
16.1.3 Step Three: Escape! . . . . .	454
16.1.4 Access by Offset . . . . .	454
16.2 String Methods . . . . .	455
16.2.1 Four Methods to the Madness . . . . .	455
16.2.2 lower() . . . . .	455
16.2.3 upper() . . . . .	455
16.2.4 str() . . . . .	456
16.2.5 Dot Notation . . . . .	456
16.3 Print . . . . .	456
16.3.1 Printing with String Literals . . . . .	457
16.3.2 Printing with Variables . . . . .	457
16.4 Advanced Printing . . . . .	457
16.4.1 String Concatenation . . . . .	457
16.4.2 Explicit String Conversion . . . . .	458
16.4.3 String Formatting with %, Part 1 . . . . .	458
16.4.4 String Formatting with %, Part 2 . . . . .	459
16.5 Review . . . . .	459
16.5.1 INSTRUCTIONS . . . . .	459
16.6 Project: Date and Time . . . . .	460
16.6.1 The datetime Library . . . . .	460
16.6.2 Getting the Current Date and Time . . . . .	460
16.6.3 Extracting Information . . . . .	460

16.6.4 Hot Date . . . . .	461
16.6.5 Pretty Time . . . . .	461
16.6.6 Grand Finale . . . . .	462
<b>17 Conditionals and Control Flow</b>	<b>463</b>
17.1 Introduction to Control Flow . . . . .	463
17.1.1 Go With the Flow . . . . .	463
17.2 Comparators . . . . .	464
17.2.1 Compare Closely! . . . . .	464
17.2.2 Compare... Closelier . . . . .	465
17.2.3 How the Tables Have Turned . . . . .	465
17.3 Boolean Operators . . . . .	466
17.3.1 To Be and/or Not to Be . . . . .	466
17.3.2 And . . . . .	467
17.3.3 Or . . . . .	467
17.3.4 Not . . . . .	468
17.3.5 This and That (or This, But Not That!) . . . . .	468
17.3.6 Mix 'n Match . . . . .	469
17.4 If, Else and Elif . . . . .	470
17.4.1 Conditional Statement Syntax . . . . .	470
17.4.2 If You're Having... . . . . .	470
17.4.3 Else Problems, I Feel Bad for You, Son... . . . . .	471
17.4.4 I Got 99 Problems, But a Switch Ain't One . . . . .	472
17.5 Review . . . . .	472
17.5.1 The Big If . . . . .	472
<b>18 Project: PygLatin</b>	<b>473</b>
18.1 PygLatin Part 1 . . . . .	473
18.1.1 Break It Down . . . . .	473
18.1.2 Ahoy! (or Should I Say Ahoyay!) . . . . .	474
18.1.3 Input! . . . . .	474
18.1.4 Check Yourself! . . . . .	474
18.1.5 Check Yourself... Some More . . . . .	475
18.1.6 Pop Quiz! . . . . .	475
18.2 PygLatin Part 2 . . . . .	476
18.2.1 Ay B C . . . . .	476
18.2.2 Word Up . . . . .	476
18.2.3 E-I-E-I-O . . . . .	477
18.2.4 I'd Like to Buy a Vowel . . . . .	477
18.2.5 Almost Oneday! . . . . .	478
18.2.6 Testing, Testing, is This Thing On? . . . . .	478
<b>19 Functions</b>	<b>481</b>
19.1 Introduction to Functions . . . . .	481
19.1.1 Documentation: a PSA . . . . .	481
19.1.2 Ample Examples . . . . .	482
19.2 Function Syntax . . . . .	483

19.2.1	Function Junction . . . . .	483
19.2.2	Call and Response . . . . .	484
19.2.3	No One Ever Does . . . . .	484
19.2.4	Splat! . . . . .	485
19.2.5	Functions Calling Functions . . . . .	486
19.2.6	Practice Makes Perfect . . . . .	486
19.3	Importing Modules . . . . .	487
19.3.1	I Know Kung Fu . . . . .	487
19.3.2	Generic Imports . . . . .	487
19.3.3	Function Imports . . . . .	488
19.3.4	Universal Imports . . . . .	488
19.3.5	Here Be Dragons . . . . .	489
19.4	Built-In Functions . . . . .	490
19.4.1	On Beyond Strings . . . . .	490
19.4.2	max() . . . . .	490
19.4.3	min() . . . . .	491
19.4.4	abs() . . . . .	491
19.4.5	type() . . . . .	491
19.5	Review . . . . .	492
19.5.1	Review: Functions . . . . .	492
19.5.2	Review: Modules . . . . .	492
19.5.3	Review: Built-In Functions . . . . .	493
<b>20</b>	<b>Taking a Vacation</b>	<b>495</b>
20.1	A Review of Function Creation . . . . .	495
20.1.1	Before We Begin . . . . .	495
20.1.2	Finding Your Identity . . . . .	495
20.1.3	Call Me Maybe? . . . . .	495
20.1.4	Function and Control Flow . . . . .	496
20.1.5	Problem Solvers . . . . .	496
20.2	Planes, Hotels and Automobiles . . . . .	496
20.2.1	Planning Your Trip . . . . .	496
20.2.2	Getting There . . . . .	497
20.2.3	Transportation . . . . .	497
20.2.4	Pull it Together . . . . .	497
20.2.5	Hey, You Never Know! . . . . .	498
20.2.6	Plan Your Trip! . . . . .	498
20.3	Return to Base . . . . .	498
20.3.1	Coming Back Home . . . . .	498
20.3.2	Gotta Give Me Some Credit . . . . .	499
20.3.3	At a Bare Minimum . . . . .	499
20.3.4	Something of Interest . . . . .	499
20.3.5	Paying Up . . . . .	500
20.3.6	Run It . . . . .	500

<b>21 Lists and Dictionaries</b>	<b>503</b>
21.1 Lists . . . . .	503
21.1.1 Introduction to Lists . . . . .	503
21.1.2 Access by Index . . . . .	504
21.1.3 New Neighbors . . . . .	504
21.2 List Capabilities and Functions . . . . .	505
21.2.1 Late Arrivals & List Length . . . . .	505
21.2.2 List Slicing . . . . .	505
21.2.3 Slicing Lists and Strings . . . . .	506
21.2.4 Maintaining Order . . . . .	506
21.2.5 For One and All . . . . .	507
21.2.6 More with 'for' . . . . .	507
21.3 Dictionaries . . . . .	508
21.3.1 This Next Part is Key . . . . .	508
21.3.2 New Entries . . . . .	508
21.3.3 Changing Your Mind . . . . .	509
21.3.4 It's Dangerous to Go Alone! Take This . . . . .	510
<b>22 Project: A Day at the Supermarket</b>	<b>511</b>
22.1 Looping with Lists and Dictionaries . . . . .	511
22.1.1 BeFOR We Begin . . . . .	511
22.1.2 This is KEY! . . . . .	512
22.1.3 Control Flow and Looping . . . . .	512
22.1.4 Lists + Functions . . . . .	513
22.1.5 String Looping . . . . .	513
22.2 Owning a Store . . . . .	513
22.2.1 Your Own Store! . . . . .	513
22.2.2 Investing in Stock . . . . .	514
22.2.3 Keeping Track of the Produce . . . . .	514
22.2.4 Something of Value . . . . .	515
22.3 Shopping Trip! . . . . .	515
22.3.1 Shopping at the Market . . . . .	515
22.3.2 Making a Purchase . . . . .	515
22.3.3 Stocking Out . . . . .	516
22.3.4 Let's Check Out! . . . . .	517
<b>23 Lists and Functions</b>	<b>519</b>
<b>24 Loops</b>	<b>521</b>
<b>25 Exam Statistics</b>	<b>523</b>
<b>26 Advanced Topics in Python</b>	<b>525</b>
<b>27 Introduction to Classes</b>	<b>527</b>
<b>28 File Input and Output</b>	<b>529</b>

<b>A Resources</b>	<b>531</b>
--------------------	------------



# Preface



# Chapter 1

# Teach yourself Programming in 10 Years

By: Peter Norvig

## 1.1 Why is everyone in such a rush?

Walk into any bookstore, and you'll see how to Teach Yourself Java in 7 Days alongside endless variations offering to teach Visual Basic, Windows, the Internet, and so on in a few days or hours. I did the following power search at Amazon.com: pubdate: after 1992 and title: days and (title: learn or title: teach yourself) and got back 248 hits. The first 78 were computer books (number 79 was Learn Bengali in 30 days). I replaced "days" with "hours" and got remarkably similar results: 253 more books, with 77 computer books followed by Teach Yourself Grammar and Style in 24 Hours at number 78. Out of the top 200 total, 96% were computer books. The conclusion is that either people are in a big rush to learn about computers, or that computers are somehow fabulously easier to learn than anything else. There are no books on how to learn Beethoven, or Quantum Physics, or even Dog Grooming in a few days. Felleisen et al. give a nod to this trend in their book How to Design Programs, when they say "Bad programming is easy. Idiots can learn it in 21 days, even if they are dummies.

Let's analyze what a title like Learn C++ in Three Days could mean:

- Learn: In 3 days you won't have time to write several significant programs, and learn from your successes and failures with them. You won't have time to work with an experienced programmer and understand what it is like to live in a C++ environment. In short, you won't have time to learn much. So the book can only be talking about a superficial familiarity, not a deep understanding. As Alexander Pope said, a little learning is a dangerous thing.
- C++: In 3 days you might be able to learn some of the syntax of C++ (if you already know another language), but you couldn't learn much about how to use the language. In short, if you were, say, a Basic programmer, you could learn to write programs in the style of Basic using C++ syntax, but you couldn't learn what C++ is actually good (and bad) for. So what's the point?

Alan Perlis once said: "A language that doesn't affect the way you think about programming, is not worth knowing". One possible point is that you have to learn a tiny bit of C++ (or more likely, something like JavaScript or Flash's Flex) because you need to interface with an existing tool to accomplish a specific task. But then you're not learning how to program; you're learning to accomplish that task.

- in Three Days: Unfortunately, this is not enough, as the next section shows.

## 1.2 Teach Yourself Programming in Ten Years

Researchers (Bloom (1985), Bryan & Harter (1899), Hayes (1989), Simmon & Chase (1973)) have shown it takes about ten years to develop expertise in any of a wide variety of areas, including chess playing, music composition, telegraph operation, painting, piano playing, swimming, tennis, and research in neuropsychology and topology. The key is deliberative practice: not just doing it again and again, but challenging yourself with a task that is just beyond your current ability, trying it, analyzing your performance while and after doing it, and correcting any mistakes. Then repeat. And repeat again. There appear to be no real shortcuts: even Mozart, who was a musical prodigy at age 4, took 13 more years before he began to produce world-class music. In another genre, the Beatles seemed to burst onto the scene with a string of #1 hits and an appearance on the Ed Sullivan show in 1964. But they had been playing small clubs in Liverpool and Hamburg since 1957, and while they had mass appeal early on, their first great critical success, Sgt. Peppers, was released in 1967. Malcolm Gladwell reports that a study of students at the Berlin Academy of Music compared the top, middle, and bottom third of the class and asked them how much they had practiced: Everyone, from all three groups, started playing at roughly the same time - around the age of five. In those first few years, everyone practised roughly the same amount - about two or three hours a week. But around the age of eight real differences started to emerge. The students who would end up as the best in their class began to practise more than everyone else: six hours a week by age nine, eight by age 12, 16 a week by age 14, and up and up, until by the age of 20 they were practising well over 30 hours a week. By the age of 20, the elite performers had all totalled 10,000 hours of practice over the course of their lives. The merely good students had totalled, by contrast, 8,000 hours, and the future music teachers just over 4,000 hours. So it may be that 10,000 hours, not 10 years, is the magic number. (Henri Cartier-Bresson (1908-2004) said "Your first 10,000 photographs are your worst," but he shot more than one an hour.) Samuel Johnson (1709-1784) thought it took even longer: "Excellence in any department can be attained only by the labor of a lifetime; it is not to be purchased at a lesser price." And Chaucer (1340-1400) complained "the lyf so short, the craft so long to lerne." Hippocrates (c. 400BC) is known for the excerpt "ars longa, vita brevis", which is part of the longer quotation "Ars longa, vita brevis, occasio praecipit, experimentum periculosum, iudicium difficile", which in English renders as "Life is short, [the] craft long, opportunity fleeting, experiment treacherous, judgment difficult." Although in Latin, ars can mean either art or craft, in the original Greek the word "techne" can

only mean "skill", not "art".

## 1.3 So You Want to be a Programmer

Here's my recipe for programming success:

1. Get interested in programming, and do some because it is fun. Make sure that it keeps being enough fun so that you will be willing to put in your ten years/10,000 hours.
2. Program. The best kind of learning is learning by doing. To put it more technically, "the maximal level of performance for individuals in a given domain is not attained automatically as a function of extended experience, but the level of performance can be increased even by highly experienced individuals as a result of deliberate efforts to improve." (p. 366) and "the most effective learning requires a well-defined task with an appropriate difficulty level for the particular individual, informative feedback, and opportunities for repetition and corrections of errors." (p. 20-21) The book *Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life* is an interesting reference for this viewpoint.
3. Talk with other programmers; read other programs. This is more important than any book or training course.
4. If you want, put in four years at a college (or more at a graduate school). This will give you access to some jobs that require credentials, and it will give you a deeper understanding of the field, but if you don't enjoy school, you can (with some dedication) get similar experience on your own or on the job. In any case, book learning alone won't be enough. "Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter" says Eric Raymond, author of *The New Hacker's Dictionary*. One of the best programmers I ever hired had only a High School degree; he's produced a lot of great software, has his own news group, and made enough in stock options to buy his own nightclub.
5. Work on projects with other programmers. Be the best programmer on some projects; be the worst on some others. When you're the best, you get to test your abilities to lead a project, and to inspire others with your vision. When you're the worst, you learn what the masters do, and you learn what they don't like to do (because they make you do it for them).
6. Work on projects after other programmers. Understand a program written by someone else. See what it takes to understand and fix it when the original programmers are not around. Think about how to design your programs to make it easier for those who will maintain them after you.
7. Learn at least a half dozen programming languages. Include one language that supports class abstractions (like Java or C++), one that supports functional abstraction (like Lisp or ML), one that supports syntactic abstraction

(like Lisp), one that supports declarative specifications (like Prolog or C++ templates), one that supports coroutines (like Icon or Scheme), and one that supports parallelism (like Sisal).

8. Remember that there is a "computer" in "computer science". Know how long it takes your computer to execute an instruction, fetch a word from memory (with and without a cache miss), read consecutive words from disk, and seek to a new location on disk. (Answers [here](#).)
9. Get involved in a language standardization effort. It could be the ANSI C++ committee, or it could be deciding if your local coding style will have 2 or 4 space indentation levels. Either way, you learn about what other people like in a language, how deeply they feel so, and perhaps even a little about why they feel so.
10. Have the good sense to get off the language standardization effort as quickly as possible.

With all that in mind, its questionable how far you can get just by book learning. Before my first child was born, I read all the How To books, and still felt like a clueless novice. 30 Months later, when my second child was due, did I go back to the books for a refresher? No. Instead, I relied on my personal experience, which turned out to be far more useful and reassuring to me than the thousands of pages written by experts. Fred Brooks, in his essay *No Silver Bullet* identified a three-part plan for finding great software designers:

- Systematically identify top designers as early as possible.
- Assign a career mentor to be responsible for the development of the prospect and carefully keep a career file.
- Provide opportunities for growing designers to interact and stimulate each other.

This assumes that some people already have the qualities necessary for being a great designer; the job is to properly coax them along. Alan Perlis put it more succinctly: "Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers". Perlis is saying that the greats have some internal quality that transcends their training. But where does the quality come from? Is it innate? Or do they develop it through diligence? As Auguste Gusteau (the fictional chef in Ratatouille) puts it, "anyone can cook, but only the fearless can be great." I think of it more as willingness to devote a large portion of one's life to deliberative practice. But maybe fearless is a way to summarize that. Or, as Gusteau's critic, Anton Ego, says: "Not everyone can become a great artist, but a great artist can come from anywhere." So go ahead and buy that Java/Ruby/Javascript/PHP book; you'll probably get some use out of it. But you won't change your life, or your real overall expertise as a programmer in 24 hours, days, or even weeks. How about working hard to continually improve over 24 months? Well, now you're starting to get somewhere...

# Chapter 2

## Programming Languages Are Not The Same

Programming languages are used for controlling the behavior of a machine (often a computer). Like natural languages, programming languages conform to rules for syntax and semantics. There are thousands of programming languages and new ones are created every year. Few languages ever become sufficiently popular that they are used by more than a few people, but professional programmers may use dozens of languages in a career. So, not to get lost between all those programming languages, there are main things we need to discuss in Programming Languages, and comparison between them.

### 2.1 Basis for Comparing Programming Languages

#### 2.1.1 Object Orientation

Many languages claim to be Object-Oriented. While the exact definition of the term is highly variable, there are several qualities that most will agree an Object-Oriented language should have:

- Encapsulation/Information Hiding
- Inheritance
- Polymorphism/Dynamic Binding
- All pre-defined types are Objects
- All operations performed by sending messages to Objects
- All user-defined types are Objects

A language is considered to be a "pure" Object-Oriented languages if it satisfies all of these qualities. A "hybrid" language may support some of these qualities, but not all. In particular, many languages support the first three qualities, but not the final three.

### 2.1.2 Static vs. Dynamic Typing

The debate between static and dynamic typing has raged in Object-Oriented circles for many years with no clear conclusion. Proponents of dynamic typing contend that it is more flexible and allows for increased productivity. Those who prefer static typing argue that it enforces safer, more reliable code, and increases efficiency of the resulting product.

Statically-typed language requires a very well-defined type system in order to remain as flexible as its dynamically-typed counterparts. Without the presence of genericity (templates) and multiple type inheritance, a static type system may severely inhibit the flexibility of a language. In addition, the presence of "casts" in a language can undermine the ability of the compiler to enforce type constraints.

A dynamic type system doesn't require variables to be declared as a specific type. Any variable can contain any value or object. In many cases this can make the software more flexible and amenable to change. However, care must be taken that variables hold the expected kind of object. Typically, if a variable contains an object of a different type than a user of the object expects, some sort of "message not understood" error is raised at run-time. Users of dynamically-typed languages claim that this type of error is infrequent in practice.

Statically-typed languages require that all variables are declared with a specific type. The compiler will then ensure that at any given time the variable contains only an object compatible with that type. (We say "compatible with that type" rather than "exactly that type" since the inheritance relationship enables subtyping, in which a class that inherits from another class is said to have an IS-A relationship with the class from which it inherits, meaning that instances of the inheriting class can be considered to be of a compatible type with instances of the inherited class.) By enforcing the type constraint on objects contained or referred to by the variable, the compiler can ensure a "message not understood" error can never occur at run-time. On the other hand, a static type system can hinder evolution of software in some circumstances. For example, if a method takes an object as a parameter, changing the type of the object requires changing the signature of the method so that it is compatible with the new type of the object being passed. If this same object is passed to many such methods, all of them must be updated accordingly, which could potentially be an arduous task. One must remember, though, that this ripple effect could occur even in a dynamically-typed language. If the type of the object is not what it was originally expected to be, it may not understand the messages being sent to it. Perhaps even worse is that it could understand the message but interpret it in a way not compatible with the semantics of the calling method. A statically-typed language can flag these errors at compilation-time, pointing out the precise locations of potential errors. A user of a dynamically-typed language must rely on extensive testing to ensure that all improper uses of the object are tracked down.

### 2.1.3 Generic Classes

Generic classes, and more generally parametric type facilities, refer to the ability to parameterize a class with specific data types. A common example is a stack class

that is parameterized by the type of elements it contains. This allows the stack to simultaneously be compile-time type safe and yet generic enough to handle any type of elements.

The primary benefit of parameterized types is that it allows statically typed languages to retain their compile-time type safety yet remain nearly as flexible as dynamically typed languages. Dynamically typed languages do not need parameterized types in order to support generic programming. Types are checked at run-time and thus dynamically typed languages support generic programming inherently.

#### 2.1.4 Inheritance

Inheritance is the ability for a class or object to be defined as an extension or specialization of another class or object. Most object-oriented languages support class-based inheritance, while others such as SELF and JavaScript support object-based inheritance. A few languages, notably Python and Ruby, support both class- and object-based inheritance, in which a class can inherit from another class and individual objects can be extended at run time with the capabilities of other objects.

Though there are identified and classified as many as 17 different forms of inheritance. Even so, most languages provide only a few syntactic constructs for inheritance which are general enough to allow inheritance to be used in many different ways. The most important distinction that can be made between various languages' support for inheritance is whether it supports single or multiple inheritance. Multiple inheritance is the ability for a class to inherit from more than one super (or base) class. For example, an application object called PersistentShape might inherit from both GraphicalObject and PersistentObject in order to be used as both a graphical object that can be displayed on the screen as well as a persistent object that can be stored in a database.

Multiple inheritance would appear to be an essential feature for a language to support for cases such as the above when two or more distinct hierarchies must be merged into one application domain. However, there are other issues to consider before making such an assertion.

First, we must consider that multiple inheritance introduces some complications into a programming language supporting it. Issues such as name clashes and ambiguities introduced in the object model must be resolved by the language in order for multiple inheritance and this leads to additional complexity in the language.

Next, we must distinguish between implementation inheritance and interface-/subtype inheritance. Subtype inheritance (also known loosely as interface inheritance) is the most common form of inheritance, in which a subclass is considered to be a subtype of its super class, commonly referred to as an IS-A relationship. What this means is that the language considers an object to conform to the type of its class or any of its super classes. For example, a Circle IS-A Shape, so anywhere a Shape is used in a program, a Circle may be used as well. This conformance notion is only applicable to statically typed languages since it is a feature used by the compiler to determine type correctness.

Implementation inheritance is the ability for a class to inherit part or all of its implementation from another class. For example, a Stack class that is implemented

using an array might inherit from an Array class in order to define the Stack in terms of the Array. In this way, the Stack class could use any features from the Array to support its own implementation. With pure implementation inheritance, the fact that the Stack inherits its implementation from Array would not be visible to code using the Stack; the inheritance would be strictly an implementation matter.

Returning to the issue of multiple inheritance, we can see that a language's support for multiple inheritance is not a boolean condition; a language can support one or more different forms of multiple inheritance in the same way it can support different forms of single inheritance (e.g. implementation and subtype inheritance).

### 2.1.5 Feature Renaming

Feature renaming is the ability for a class or object to rename one of its features (a term used to collectively refer to attributes and methods) that it inherited from a super class. There are two important ways in which this can be put to use:

- Provide a feature with a more natural name for its new context
- Resolve naming ambiguities when a name is inherited from multiple inheritance paths

### 2.1.6 Method Overloading

Method overloading (also referred to as parametric polymorphism) is the ability for a class, module, or other scope to have two or more methods with the same name. Calls to these methods are disambiguated by the number and/or type of arguments passed to the method at the call site. For example, a class may have multiple print methods, one for each type of thing to be printed. The alternative to overloading in this scenario is to have a different name for each print method, such as print\_string and print\_integer.

### 2.1.7 Operator Overloading

Operator overloading is the ability for a programmer to define an operator (such as +, or \*) for user-defined types. This allows the operator to be used in infix, prefix, or postfix form, rather than the standard functional form. For example, a user-defined Matrix type might provide a \* infix operator to perform matrix multiplication with the familiar notation: matrix1 \* matrix2 .

Some (correctly) consider operator overloading to be mere syntactic "sugar" rather than an essential feature, while others (also correctly) point to the need for such syntactic sugar in numerical and other applications. Both points are valid, but it is clear that, when used appropriately, operator overloading can lead to much more readable programs. When abused, it can lead to cryptic, obfuscated code. Consider that in the presence of operator overloading, it may not be clear whether a given operator is built in to the language or defined by the user. For any language that supports operator overloading, two things are necessary to alleviate such obfuscation:

All operations must be messages to objects, and thus all operators are always method calls. Operators must have an equivalent functional form, so that using the operator as a method call will behave precisely the same as using it in infix, prefix, or postfix form.

### 2.1.8 Higher Order Functions & Lexical Closures

Higher order functions are, in the simplest sense, functions that can be treated as if they were data objects. In other words, they can be bound to variables (including the ability to be stored in collections), they can be passed to other functions as parameters, and they can be returned as the result of other functions. Due to this ability, higher order functions may be viewed as a form of deferred execution, wherein a function may be defined in one context, passed to another context, and then later invoked by the second context. This is different from standard functions in that higher order functions represent anonymous lambda functions, so that the invoking context need not know the name of the function being invoked.

Lexical closures (also known as static closures, or simply closures) take this one step further by bundling up the lexical (static) scope surrounding the function with the function itself, so that the function carries its surrounding environment around with it wherever it may be used. This means that the closure can access local variables or parameters, or attributes of the object in which it is defined, and will continue to have access to them even if it is passed to another module outside of its scope.

### 2.1.9 Garbage Collection

Garbage collection is a mechanism allowing a language implementation to free memory of unused objects on behalf of the programmer, thus relieving the burden on the programmer to do so. The alternative is for the programmer to explicitly free any memory that is no longer needed. There are several strategies for garbage collection that exist in various language implementations.

#### Reference Counting

Reference counting is the simplest scheme and involves the language keeping track of how many references there are to a particular object in memory, and deleting that object when that reference count becomes zero. This scheme, although it is simple and deterministic, is not without its drawbacks, the most important being its inability to handle cycles. Cycles occur when two objects reference each other, and thus their reference counts will never become zero even if neither object is referenced by any other part of the program.

#### Mark and Sweep

"Mark and sweep" garbage collection is another scheme that overcomes this limitation. A mark and sweep garbage collector works in a two phase process, known as the mark phase and the sweep phase. The mark phase works by first starting at

the "root" objects (objects on the stack, global objects, etc.), marking them as live, and recursively marking any objects referenced from them. These marked objects are the set of live objects in program, and any objects that were not marked in this phase are unreferenced and therefore candidates for collection. In the sweep phase, any objects in memory that were not marked as live by the mark phase are deleted from memory. The primary drawback of mark and sweep collection is that it is non-deterministic, meaning that objects are deleted at an unspecified time during the execution of the program. This is the most common form of garbage collection.

## Generational

Generational garbage collection works in a similar fashion to mark and sweep garbage collection, except it capitalizes on the statistical probability that objects that have been alive the longest tend to stay alive longer than objects that were newly created. Thus a generational garbage collector will divide objects into "generations" based upon how long they've been alive. This division can be used to reduce the time spent in the mark and sweep phases because the oldest generation of objects will not need to be collected as frequently. Generational garbage collectors are not as common as the other forms.

### 2.1.10 Uniform Access

The Uniform Access Principle, states that "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation." It is described further with "Although it may at first appear just to address a notational issue, the Uniform Access principle is in fact a design rule which influences many aspects of object-oriented design and the supporting notation. It follows from the Continuity criterion; you may also view it as a special case of Information Hiding."

Say that `teach` is a feature of a class named `Professor`. For languages that do not support the Uniform Access Principle, the notation used to access `teach` differs depending on whether it is an attribute (storage) or a function (computation). For example, in Java we would use `Professor.teach` if it were an attribute, but we would use `Professor.teach()` if it were a function. Having this notational difference means that users of `Professor` are exposed to unnecessary implementation details and are tightly coupled to `Professor`. If `teach` is changed from attribute to method (or vice versa), then any users of `Professor` must also be changed.

The Uniform Access Principle seeks to eliminate this needless coupling. A language supporting the Uniform Access Principle does not exhibit any notational differences between accessing a feature regardless of whether it is an attribute or a function. Thus, in our earlier example, access to `bar` would always be in the form of `foo.bar`, regardless of how `bar` is implemented. This makes clients of `Foo` more resilient to change.

### 2.1.11 Class Variables/Methods

Class variables and methods are owned by a class, and not any particular instance of a class. This means that for however many instances of a class exist at any given point in time, only one copy of each class variable/method exists and is shared by every instance of the class.

### 2.1.12 Reflection

Reflection is the ability for a program to determine various pieces of information about an object at run-time. This includes the ability to determine the type of the object, its inheritance structure, and the methods it contains, including the number and types of parameters and return types. It might also include the ability for determining the names and types of attributes of the object. Most object-oriented languages support some form of reflection.

### 2.1.13 Access Control

Access control refers to the ability for a module's implementation to remain hidden behind its public interface. Access control is closely related to the encapsulation/information hiding principle of object-oriented languages. For example, a class `Professor` may have methods such as `name` and `email`, that return the professor's name and e-mail address respectively. How these methods work is an implementation detail that should not be available to users of the `Person` class. These methods may, for example, connect to a database to retrieve the values. The database connection code that is used to do this is not relevant to client code and should not be exposed. Language-enforced access control allows us to enforce this. Most object-oriented languages provide at least two levels of access control: public and protected. Protected features are not available outside of the class in which they are contained, except for subclasses. Some languages, notably Java and C++, provide a third level of access control known as "private". Private features are not available outside of the class in which they are declared, even for subclasses. Note, however, that this means that objects of a particular class can access the private features of other objects of that same class. Java provides a fourth level of, known as "package private" access control which allows other classes in the same package to access such features.

### 2.1.14 Design by Contract

Design by Contract (DBC) is the ability to incorporate important aspects of a specification into the software that is implementing it. The most important features of DBC are:

- Pre-conditions, which are conditions that must be true before a method is invoked
- Post-conditions, which are conditions guaranteed to be true after the invocation of a method

- Invariants, which are conditions guaranteed to be true at any stable point during the lifetime of an object

There is much more to DBC than these simple facilities, including the manner in which pre-conditions, post-conditions, and invariants are inherited. However, at least these facilities must be present to support the central notions of DBC.

### 2.1.15 Multithreading

Multithreading is the ability for a single process to process two or more tasks concurrently. (We say concurrently rather than simultaneously because, in the absence of multiple processors, the tasks cannot run simultaneously but rather are interleaved in very small time slices and thus exhibit the appearance and semantics of concurrent execution.) The use of multithreading is becoming increasingly more common as operating system support for threads has become near ubiquitous.

### 2.1.16 Regular Expressions

Regular expressions are pattern matching constructs capable of recognizing the class of languages known as regular languages. They are frequently used for text processing systems as well as for general applications that must use pattern recognition for other purposes. Libraries with regular expression support exist for nearly every language, however it has become increasingly important for a language to support regular expressions natively. This allows tighter integration with the rest of the language and allows more convenient syntax for use of regular expressions.

### 2.1.17 Pointer Arithmetic

Pointer arithmetic is the ability for a language to directly manipulate memory addresses and their contents. While, due to the inherent unsafety of direct memory manipulation, this ability is not often considered appropriate for high-level languages, it is essential for low-level systems applications. Thus, while object-oriented languages strive to remain at a fairly high level of abstraction, to be suitable for systems programming a language must provide such features or relegate such low-level tasks to a language with which it can interact. Most object-oriented languages have foregone support of pointer arithmetic in favor of providing integration with C. This allows low-level routines to be implemented in C while the majority of the application is written in the higher level language. C++ on the other hand provides direct support for pointer arithmetic, both for compatibility with C and to allow C++ to be used for systems programming without the need to drop down to a lower level language. This is the source both of C++'s great flexibility as well as much of its complexity.

### 2.1.18 Language Integration

For various reasons, including integration with existing systems, the need to interact with low level modules. It is important for a high level language (particularly inter-

interpreted languages) to be able to integrate seamlessly with other languages. Nearly every language to come along since C was first introduced provides such integration with C. This allows high level languages to remain free of the low level constructs that make C great for systems programming, but add much complexity.

### 2.1.19 Built-In Security

Built-in security refers to a language implementation's ability to determine whether or not a piece of code comes from a "trusted" source (such as the user's hard disk), limiting the permissions of the code if it does not. For example, Java applets are considered untrusted, and thus they are limited in the actions they can perform when executed from a user's browser. They may not, for example, read or write from or to the user's hard disk, and they may not open a network connection to anywhere but the originating host.

## 2.2 Comparing based on Performance and Optimization

Based on the previous basis for comparing programming languages, and because there are thousands of them, it is almost impossible to provide a comprehensive list of such features here. You, dear student, is encouraged to continue research in this point, and you are welcome to introduce such comparison between different programming languages of your choice in a report. Recent Google research [1] presents an important comparison between different programming languages. Through this important presented benchmark, differences between programming languages become clear. In this experience report, researcher encoded a well specified, compact benchmark in four programming languages, namely C++, Java, Go, and Scala. The implementations each use the languages' idiomatic container classes, looping constructs, and memory/object allocation schemes. It does not attempt to exploit specific language and run-time features to achieve maximum performance. This approach allows an almost fair comparison of language features, code complexity, compilers and compile time, binary sizes, run-times, and memory footprint. While the benchmark itself is simple and compact, it employs many language features, in particular, higher-level data structures (lists, maps, lists and arrays of sets and lists), a few algorithms (union/ find, dfs/deep recursion, and loop recognition based on Tarjan), iterations over collection types, some object oriented features, and interesting memory allocation patterns. Researcher did not explore any aspects of multi-threading, or higher level type mechanisms, which vary greatly between the languages. The benchmark points to very large differences in all examined dimensions of the language implementations. After publication of the benchmark internally at Google, several engineers produced highly optimized versions of the benchmark. Researcher found that in regards to performance, C++ wins out by a large margin. However, it also required the most extensive tuning efforts, many of which were done at a level of sophistication that would not be available to the average programmer. Scala concise notation and powerful language features allowed for the

best optimization of code complexity. The Java version was probably the simplest to implement, but the hardest to analyze for performance. Specifically the effects around garbage collection were complicated and very hard to tune. Since Scala runs on the JVM, it has the same issues. Go offers interesting language features, which also allow for a concise and standardized notation. The compilers for this language are still immature, which reflects in both performance and binary sizes.

# Chapter 3

## Python Tutorial

# Python: Introduction for Programmers

Bruce Beckles

Bob Dowling

University Computing Service

Scientific Computing Support e-mail address:  
[escience-support@ucs.cam.ac.uk](mailto:escience-support@ucs.cam.ac.uk)

1

This is the UCS one day course on Python for people who have some experience of programming in a programming language other than Python. The course assumes that you know *no* Python whatsoever – we warn all those people who already have some Python experience that they are likely to be bored on this course. Also, those people who already have considerable programming experience in several different programming languages are also likely to be bored, and would be better off just working through the notes in their own time, or looking at one of the many on-line Python tutorials.

Note that this course covers Python 2.2 to 2.6, which are the most common versions currently in use – it does **NOT** cover the recently released Python 3.0 (or 3.1) since those versions of Python are so new. Python 3.0 is significantly different to previous versions of Python, and this course will be updated to cover it as it becomes more widely used.

Also, people who do **not** already know how to program in another language – or who have minimal programming experience – and want to learn Python will probably find this course too fast/difficult for them. Such people are better off attending the UCS “Python: Introduction for Absolute Beginners” three afternoon course. For details of this course, see:

<http://training.csx.cam.ac.uk/course/python>

The official UCS e-mail address for all scientific computing support queries, including any questions about this course, is:

[escience-support@ucs.cam.ac.uk](mailto:escience-support@ucs.cam.ac.uk)



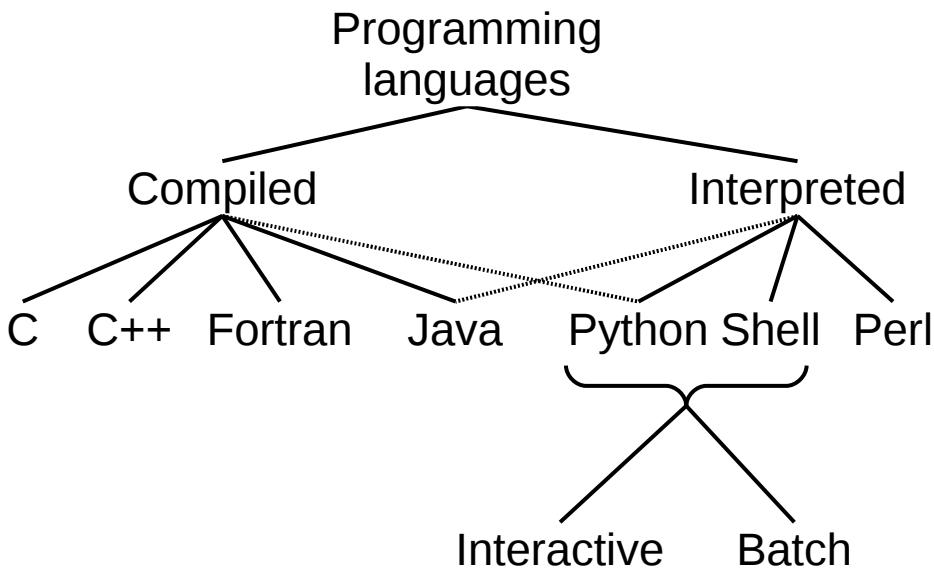
Python:  
*Introduction for Programmers*  
Bruce Beckles Bob Dowling  
University Computing Service  
Scientific Computing Support e-mail address:  
[escience-support@ucs.cam.ac.uk](mailto:escience-support@ucs.cam.ac.uk)

2

Python is named after Monty Python's Flying Circus, not the constricting snake.

There are various versions of Python in use, the most common of which are releases of Python 2.2, 2.3, 2.4, 2.5 and 2.6. (The material in this course is applicable to versions of Python in the 2.2 to 2.6 releases.)

On December 3rd, 2008, Python 3.0 was released. Python 3.0 is significantly different to previous versions of Python, is **not** covered by this course, and *breaks* backward compatibility with previous Python versions in a number of ways. As Python 3.0 and 3.1 become more widely used, this course will be updated to cover them.



3

As you probably know, programming languages split into two broad camps according to how they are used.

Compiled languages go through a “compilation” stage where the text written by the programmer is converted into machine code. This machine code is then processed directly by the CPU at a later stage when the user wants to run the program. This is called, unsurprisingly, “run time”.

Interpreted languages are stored as the text written by the programmer and this is converted into machine instructions all in one go at run time.

There are some languages which occupy the middle ground. Java, for example, is converted into a pseudo-machine-code for a CPU that doesn’t actually exist. At run time the Java environment emulates this CPU in a program which interprets the supposed machine code in the same way that a standard interpreter interprets the plain text of its program. In the way Java is treated it is closer to a compiled language than a classic interpreted language so it is treated as a compiled language in this course.

Python can create some intermediate files to make subsequent interpretation simpler. However, there is no formal “compilation” phase the user goes through to create these files and they get automatically handled by the Python system. So in terms of how we use it, Python is a classic interpreted language. Any clever tricks it pulls behind the curtains will be ignored for the purposes of this course.

So, if an interpreted language takes text programs and runs them directly, where does it get its text from? Interpreted languages typically support getting their text either directly from the user typing at the keyboard or from a text file of commands.

If the interpreter (Python in our case) gets its input from the user then we say it is running “interactively”. If it gets its input from a file we say it is running in “batch mode”. We tend to use interactive mode for simple use and a text file for anything complex.

# Interactive use

```
Unix prompt
$ python
Python 2.6 (r26:66714, Feb 3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on ...
Type "help", "copyright", "credits" or "license" ...
>>> print 'Hello, world!'
Python prompt
Hello, world!
>>> 3
3
```

4

Now that we have a Unix command line interpreter we issue the command to launch the Python interpreter. That command is the single word, “python”.

In these notes we show the Unix prompt, the hint from the Unix system that it is ready to receive commands, as a single dollar sign character (\$). On PWF Linux the prompt is actually that character preceded by some other information.

Another convention in these notes is to indicate with the use of **bold face** the text that you have to type while regular type face is used for the computer’s output.

The interactive Python interpreter starts by printing three lines of introductory blurb which will not be of interest to us.

After this preamble though, it prints a Python prompt. This consists of three “greater than” characters (>>>) and is the indication that the Python interpreter is ready for you to type some Python commands. You cannot type Unix commands at the prompt. (Well, you can type them but the interpreter won’t understand them.)

So let’s issue our first Python command. There’s a tradition in computing that the first program developed in any language should output the phrase “Hello, world!” and we see no reason to deviate from the norm here.

The Python command to output some text is “print”. This command needs to be followed by the text to be output. The text, “Hello, world!” is surrounded by single quotes (') to indicate that it should be considered as text by Python and not some other commands. The item (or items) that a command (such as `print`) needs to know what to do are called its “arguments”, so here we would say that ‘Hello, world!’ is the `print` command’s argument.

The command is executed and the text “Hello, world!” is produced. The `print` command always starts a new line after outputting its text.

You will probably not be surprised to learn that everything in Python is **case-sensitive**: you have to give the `print` command all in lower-case, “PRINT”, “pRiNT”, etc. won’t work.

Note that what the Python interpreter does is evaluate whatever it has been given and outputs the result of that evaluation, so if we just give it a bare number, e.g. 3, then it evaluates that number and displays the result:

```
$ python
Python 2.6 (r26:66714, Feb 3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on ...
Type "help", "copyright", "credits" or "license" ...

>>> print 'Hello, world!'
Hello, world!

>>> 3
3
>>>
```

To quit the Python interpreter:  
Press *control+d*

\$

Unix prompt

5

Now that we are in the Python interpreter it would be useful if we knew how to get out of it again. In common with many Unix commands that read input from the keyboard, the interpreter can be quit by indicating “end of input”. This is done with a “control+d”. To get this hold down the control key (typically marked “Ctrl”) and tap the “D” key once. Then release the control key.

Be careful to only press the “D” key once. The “control+d” key combination, meaning end of input, also means this to the underlying Unix command interpreter. If you press “control+d” twice, the first kills off the Python interpreter returning control to the Unix command line, and the second kills off the Unix command interpreter. If the entire terminal window disappears then this is what you have done wrong. Start up another window, restart Python and try again.

If you are running Python interactively on a non-Unix platform you may need a different key combination. If you type “exit” at the Python prompt it will tell you what you need to do on the current platform. On PWF Linux you get this:

```
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>>
```

It would also be useful if we knew how to use Python’s help system. We’ll look at how we access Python’s help in a few slides’ time.

# Batch use

```
#!/usr/bin/python  
print 'Hello, world!'
```

3

hello.py

\$ **python hello.py**

Hello, world!



No "3"

6

Now let us look at the file `hello.py` in the home directory of the course accounts you are using. We see the same two lines:

```
print 'Hello, world!'  
3
```

(Ignore the line starting `#`. Such lines are comments and have no effect on a Python program. We will return to them later.)

The suffix “`.py`” on the file name is not required by Python but it is conventional and some editors will put you into a “Python editing mode” automatically if the file’s name ends that way. Also, on some non-Unix platforms (such as Microsoft Windows) the “`.py`” suffix will indicate to the operating system that the file is a Python script. (Python programs are conventionally referred to as “Python scripts”).

We can run this script in batch mode by giving the name of the file to the Python interpreter at the Unix prompt:

```
$ python hello.py  
Hello, world!
```

This time we see different output. The `print` command seems to execute, but there is no sign of the 3.

We can now see the differences between interactive mode and batch mode:

- Interactive Python evaluates every line given to it and outputs the evaluation. The `print` command doesn’t evaluate to anything, but prints its argument at the same time. The integer 3 outputs nothing (it isn’t a command!) but evaluates to 3 so that gets output.
- Batch mode is more terse. Evaluation is not output, but done quietly. Only the commands that explicitly generate output produce text on the screen.  
Batch mode is similarly more terse in not printing the introductory blurb.

```
$ python
Python 2.6 (r26:66714, Feb 3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on ...
Type "help", "copyright", "credits" or "license" ...
```

```
>>> help
```

Type `help()` for interactive help, or `help(object)` for help about object.

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

Welcome to Python 2.6! This is the online help utility.

If this is your first time using Python, ...

help> help utility prompt

7

Launch the Python interpreter again as we will be using it interactively for a while.

The first thing we will do is look at Python's interactive help (which Python refers to as its "online help utility").

You may have noticed that the introductory blurb we get when we start Python suggests a number of words we might like to type to get "more information". One of those words is "help". Let's see what happens if we type "help" at the Python prompt:

```
>>> help
```

Type `help()` for interactive help, or `help(object)` for help about object.

Python tells us there are two ways we can get help from within the Python interpreter. We can either get interactive help via its online help utility by typing "`help()`", which we'll do in a moment, or we can directly ask Python for help about some particular thing (which we'll do a bit later by typing "`help('thing')`" where "`thing`" is the Python command we want to know about).

So let's try out Python's interactive help, by typing "`help()`" at the Python prompt.

This will start Python's online help utility as shown above. Note that we get an introductory blurb telling us how the utility works (as well as how to quit it (type "quit")), and the prompt changes from Python's prompt (">>>") to:

```
help>
```

Note that the online help utility will only provide useful information if the Python documentation is installed on the system. If the documentation hasn't been installed then you won't be able to get help this way. Fortunately, the complete Python documentation (exactly as given by the online help utility) is available on the web at:

<http://docs.python.org>

...in a variety of formats. It also includes a tutorial on Python.

```

help> print ← The thing on which you want help
help> quit ← Type “quit” to leave the help utility

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>> ← Back to Python prompt
>>> help('print') Note the quote marks
(' ' or "")
```

>>>

Official Python documentation (includes tutorial):  
<http://docs.python.org/>

8

Using Python's online help utility interactively is really straightforward: you just type the name of the Python command, keyword or topic you want to learn more about and press return. Let's see what happens if we ask it about the `print` command:

```
help> print
```

On PWF Linux the screen clears and we get a new screen of text that looks something like this:

---

6.6 The print statement

```

print_stmt      ::=      "print" ([expression[1] (",, expression[2])* [",,"]
| ">>" expression[3] [(",," expression[4])+ [",,"])
```

[Download entire grammar as text.\[5\]](#)

```

print evaluates each expression in turn and writes the resulting object
lines 1-10
```

(For space reasons only the first 10 lines of the help text are shown (in very small type – don't try and read this text in these notes here but rather try this out yourself on the computer in front of you).) You can get another page of text by pressing the **space bar**, and move back a page by typing “**b**” (for “back”), and you can quit from this screen by typing “**q**” (for “quit”). (If you want help on what you can do in this screen, type “**h**” (for “help”).) The program that is displaying this text is not Python but an external program known as a *pager* (because it displays text a page at a time) – this means exactly which pager is used and how it behaves depends on the underlying operating system and how it is configured. Note that on PWF Linux when you finish reading this help text and type “**q**”, the help text disappears and you get back the screen you had before, complete with the “`help>`” prompt.

When you've finished trying this out, quit the help utility by typing “quit” at the “`help>`” prompt and pressing return. Finally, we can also get help on a Python command, keyword or help topic directly, without using the online help utility interactively. To do this, type “`help('thing')`” at the Python prompt, where “*thing*” is the Python command, etc. on which you want help (note the quotes (' ) around “*' thing*”).

```
$ python  
Python 2.6 (r26:66714, Feb 3 2009, 20:52:03)  
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on ...  
Type "help", "copyright", "credits" or "license" ...
```

```
>>> print 3  
3  
>>> print 3, 5  
3 5
```

The diagram illustrates the behavior of the `print` command in Python. It shows two examples: `print 3` and `print 3, 5`. In the first example, the number `3` is enclosed in a blue box, representing a single argument. In the second example, the numbers `3` and `5` are each enclosed in separate blue boxes, representing two arguments separated by a comma. Arrows point from the text labels to the corresponding parts of the code. The label "Pair of arguments separated by a comma" points to the comma in the second example. The label "Pair of outputs no comma" points to the two separate arguments in the second example.

9

Before we go any further, we'll demonstrate a property of the `print` command.

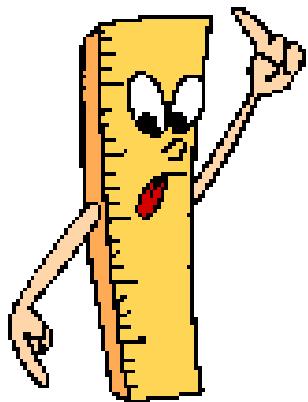
We have already seen `print` used with a single argument.

But we can also use it with two arguments, separated by a comma.

Note that `print` outputs the two arguments (separated by a single space), and not the comma. The comma separates the arguments; it is not an argument in its own right.

This is a general feature of Python commands: when the command has multiple arguments, commas are used to separate the arguments.

# Using Python for science



→ Science

↓ Quantitative

← Numbers

## Using numbers in Python

10

We are going to spend a little while talking about numbers and how computers (and in particular Python) handle them.

Why?

Well, this is a course in the “Scientific Computing” series and science is about quantifying things. That requires numbers to represent those quantities and so we need to understand how Python handles numbers.

Those of you who are already familiar with how computers handle numbers, please bear with us for a little while – despite what you may be tempted to think, this is not common knowledge to *all* programmers, especially those who have learnt to program in languages which tend to hide the subtleties of computer arithmetic from the programmer (e.g. Perl, Visual Basic).

# $\mathbb{Z}$ Integers

$$\{ \dots -2, -1, 0, \\ 1, 2, 3, \dots \}$$

11

We will start with the integers. (In case you have not met it before,  $\mathbb{Z}$  is the mathematical symbol for the integers.)

<code>&gt;&gt;&gt; 7+3</code>	<code>&gt;&gt;&gt; 7-3</code>	
10	4	
<code>&gt;&gt;&gt; 7*3</code>	<code>&gt;&gt;&gt; 7**3</code>	$7^3$ : use “**” for exponentiation
21	343	
<code>&gt;&gt;&gt; 7/3</code>	<code>&gt;&gt;&gt; -7/3</code>	integer division rounds down
2	-3	
<code>&gt;&gt;&gt; 7%3</code>	<code>&gt;&gt;&gt; -7%3</code>	remainder (mod) returns 0 or positive integer
1	2	

12

On some systems (including PWF Linux) the Python interpreter has built-in command line history. If you press the up and down arrows you can navigate backwards and forwards through your recent Python commands. You can also move left and right through the line (using the left and right arrow keys) to edit it.

Most of you will be familiar with the basic arithmetic operations. (For those who have not met these conventions before, we use the asterisk, “\*” for multiplication rather than the times sign, “×”, that you may be used to from school. Similarly we use the forward slash character, “/” for division rather than the division symbol, “÷”.)

The first thing to note, if you have not already come across it, is that division involving two integers (“integer division”) always returns an integer. Integer division rounds *strictly* downwards, so the expression “7/3” gives “2” rather than “2 1/3” and “-7/3” gives “-3” as this is the integer below “-2 1/3”. So  $(-7)/3$  does not evaluate to the same thing as  $-(7/3)$ . (Integer division is also called “floor division”).

There are also two slightly less familiar arithmetic operations worth mentioning:

- Exponentiation (raising a number to a power): the classical notation for this uses superscripts, so “7 raised to the power 3” is written as “ $7^3$ ”. Python uses double asterisks, “\*\*”, so we write “ $7^3$ ” as “7\*\*3”. You are permitted spaces around the “\*\*” but not inside it, i.e. you cannot separate the two asterisks with spaces. Some programming languages use “ $\wedge$ ” for exponentiation, but Python doesn’t – it uses “ $\wedge$ ” for a completely different operation (bitwise exclusive or (xor), which we don’t cover in this course). Python also has a function, pow(), which can be used for exponentiation, so  $\text{pow}(x, y) = x^y$ . E.g.

```
>>> pow(7, 3)
343
```

- Remainder (also called “mod” or “modulo”): this operation returns the remainder when the first number is divided by the second, and Python uses the percent character, “%” for this. “7%3” gives the answer “1” because 7 leaves a remainder of 1 when divided by 3. The remainder is always zero or positive, even when the number in front of the percent character is negative, e.g. “-7%3” gives the answer “2” because  $(3 \times -3) + 2 = -7$ .

Another function worth mentioning at this point is the abs() function, which takes a number and returns its *absolute value*. So  $\text{abs}(a) = |a|$ . (Note that pow() and abs(), in common with most functions, require parentheses (round brackets) around their arguments – the print function is a special case that doesn’t.)

```
>>> 2*2
4
>>> 4*4
16
>>> 16*16
256
>>> 256*256
65536
>>> 65536*65536
4294967296L
```



“large” integer

13

Python’s integer arithmetic is very powerful and there is no limit (except the system’s memory capacity) for the size of integer that can be handled. We can see this if we start with 2, square it, get and answer and square that, and so on. Everything seems normal up to 65,536.

If we square 65,536 Python gives us an answer, but the number is followed by the letter “L”. This indicates that Python has moved from standard integers to “long” integers which have to be processed differently behind the scenes but which are just standard integers for our purposes. Just don’t be startled by the appearance of the trailing “L”.

Note: If you are using a system with a 64-bit CPU and operating system then the 4,294,967,296 also comes without an “L” and the “long” integers only kick in one squaring later.

```
>>> 4294967296*4294967296  
18446744073709551616L  
  
>>> 18446744073709551616 *  
18446744073709551616  
340282366920938463463374607431768211456L  
  
>>> 2**521 - 1  
6864797660130609714981900799081393217269  
4353001433054093944634591855431833976560  
5212255964066145455497729631139148085803  
7121987999716643812574028291115057151L
```

No inherent limit to Python's integer arithmetic:  
can keep going until we run out of memory

14

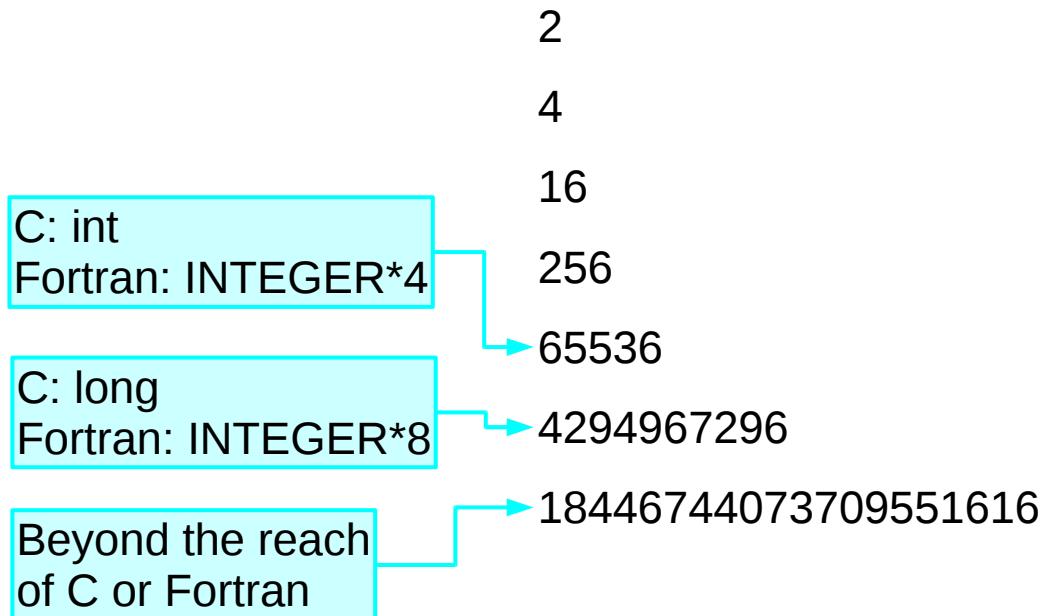
We can keep squaring, limited only by the base operating system's memory. Python itself has no limit to the size of integer it can handle.

Indeed, we can try calculating even larger integers using exponentiation, e.g. “ $2^{521} - 1$ ” ( $2^{521}-1$ ).

(In case you are wondering, “ $2^{521}-1$ ” is the 13th Mersenne prime, which was discovered in 1952 by Professor R. M. Robinson. It has 157 digits. If you are curious about Mersenne primes, the following URLs point to documents that provide good overviews of these numbers:

<http://primes.utm.edu/mersenne/>  
[http://en.wikipedia.org/wiki/Mersenne\\_prime](http://en.wikipedia.org/wiki/Mersenne_prime)

)



15

As you've probably realised, Python is quite exceptional in this regard. C and Fortran have strict limits on the size of integer they will handle. C++ and Java have the same limits as C but do also have the equivalent of Python's "long integers" as well (although they call them "big integers"; note that for C++ you need an additional library). However, in C++ and Java you must take explicit action to invoke so-called "big integers"; they are not engaged automatically or transparently as they are in Python.

(Note that more recent versions of C have a "long long" integer type which you can use to get values as large as 18,446,744,073,709,551,615.)



# Floating point numbers

16

And that wraps it up for integers.

Next we would like to move on to real numbers but here we encounter the reality of computing not reflecting reality. Subject to the size limits of the system memory we could say that the mathematical set of integers was faithfully represented by Python. We cannot say the same for real numbers. Python, and all computing systems, have an approximation to the real numbers called “floating point numbers”. (There is an alternative approximation called “fixed point numbers” but most programming languages, including Python, don’t implement that so we won’t bother with it.)

As you may know,  $\mathbb{R}$  is the mathematical symbol for the real numbers. However, since the computer’s floating-point numbers are only an approximation of the real numbers – and not a very good approximation at that – the authors are using a *crossed out*  $\mathbb{R}$  to represent them in this course.

Again, those of you who have worked with numerical code in other programming languages will already be familiar with the vagaries of floating point arithmetic – treat this as a brief revision of the relevant concepts.

Those of you who are *not* familiar with the vagaries of floating point arithmetic should have a look at the article “The Perils of Floating Point” by Bruce M. Bush, available on-line at:

<http://www.lahey.com/float.htm>

Note that all the examples in this article are in Fortran, but everything the article discusses is as relevant to Python as it is to Fortran.

```
>>> 1.0
```

```
1.0
```

Floating point number

```
>>> 0.5
```

```
0.5
```

$\frac{1}{2}$  is OK

```
>>> 0.25
```

```
0.25
```

$\frac{1}{4}$  is OK

Powers  
of two

```
>>> 0.1
```

```
0.10000000000000001
```

$\frac{1}{10}$  is not

## Usual issues with representation in base 2

17

We represent floating point numbers by including a decimal point in the notation. “1.0” is the floating point number “one point zero” and is quite different from the integer “1”. (We can specify this to Python as “1.” instead of “1.0” if we wish.)

Even with simple numbers like this, though, there is a catch. We use “base ten” numbers but computers work internally in base two. The floating point system can cope with moderate integer values like 1·0, 2·0 and so on, but has a harder time with simple fractions. Fractions that are powers of two (half, quarter, eighth, etc.) can all be handled exactly correctly. Fractions that aren’t, like a tenth for example, are approximated internally. We see a tenth (0·1) as simpler than a third (0·33333333...) only because we write in base ten. In base two a tenth is the infinitely repeating fraction 0·00011001100110011... Since the computer can only store a finite number of digits, numbers such as a tenth can only be stored approximately. So whereas in base ten, we can exactly represent fractions such as a half, a fifth, a tenth and so on, with computers it’s only fractions like a half, a quarter, an eighth, etc. that have the privileged status of being represented exactly.

We’re going to ignore this issue in this introductory course and will pretend that numbers are stored internally the same way we see them as a user.

Python provides a number of functions that perform various mathematical operations (such as finding the positive square root of a positive number) in one of its libraries – the `math` module (Python calls its libraries “*modules*”; we’ll meet modules a little later). You can find out what functions are in this module by typing “`help('math')`” at the Python prompt, or from the following URL:

<http://docs.python.org/library/math.html>

Most of the functions in the `math` module can be used on either integers or floating point numbers. In general, these functions will give their result as a floating point number even if you give them an integer as input.

```
>>> 2.0*2.0  
4.0  
>>> 4.0*4.0  
16.0  
  
...  
  
>>> 65536.0*65536.0  
4294967296.0  
>>> 4294967296.0*4294967296.0  
1.8446744073709552e+19
```

17 significant figures

18

If we repeat the successive squaring trick that we applied to the integers everything seems fine up to just over 4 billion.

If we square it again we get an unexpected result. The answer is printed as

1.8446744073709552e+19

This means  $1.8446744073709552 \times 10^{19}$ , which isn't the right answer.

In Python, floating point numbers are stored to only 17 significant figures of accuracy. Positive floating point numbers can be thought of as a number between 1 and 10 multiplied by a power of 10 where the number between 1 and 10 is stored to 17 significant figures of precision. (Recall that actually the number is stored in base 2 with a power of 2 rather than a power of 10. For our purposes this detail won't matter.)

In practice this should not matter to scientists. If you are relying on the sixteenth or seventeenth decimal place for your results you're doing it wrong!

What it *does* mean is that if you are doing mathematics with values that you think ought to be integers you should stick to the integer type, not the floating point numbers.

(Note for pedants: Python floating point numbers are really C doubles. This means that the number of significant figures of accuracy to which Python stores floating point numbers depends on the precision of the double type of the underlying C compiler that was used to compile the Python interpreter. On most modern PCs this means that you will get at least 17 significant figures of accuracy, but the exact precision may vary. Python does not provide any easy way for the user to find out the exact range and precision of floating point values on their machine.)

```
>>> 4294967296.0*4294967296.0
1.8446744073709552e+19

>>> 1.8446744073709552e+19*1.8446744073709552e+19
3.4028236692093846e+38

>>> 3.4028236692093846e+38*3.4028236692093846e+38
1.157920892373162e+77

>>> 1.157920892373162e+77*1.157920892373162e+77
1.3407807929942597e+154

>>> 1.3407807929942597e+154*1.3407807929942597e+154
inf
```

overflow

Limit at  $2^{**1023}$

19

Just as there is a limit of 17 significant figures on the precision of the number there is a limit on how large the power of 10 can get. (On most modern PCs, the limit will be about  $2^{1023}$ , although the exact limit depends on the hardware, operating system, etc.) If we continue with the squaring just four more times after 4294967296.0 we get a floating point number whose exponent is too great to be stored. Python indicates this by printing “`inf`” as the answer. We have reached “floating point overflow”, or “infinity”.

Note that sometimes Python will give you an `OverflowError` (which will normally cause your script to stop executing and exit with an error) if the result of a calculation is too big, rather than an `inf` (which allows your script to carry on, albeit with “`inf`” as the answer instead of an actual number).

# Machine epsilon

```
>>> 1.0 + 1.0e-16
```

1.0

too small to make  
a difference

```
>>> 1.0 + 2.0e-16
```

1.0000000000000002

large enough

```
>>> 1.0 + 1.1e-16
```

1.0

```
>>> 1.0 + 1.9e-16
```

1.0000000000000002

Spend the next few  
minutes using Python  
interactively to estimate  
machine epsilon – we'll  
write a Python program  
to do this for us a little  
later

20

The limit of 17 significant figures alluded to earlier begs a question. What is the smallest positive floating point number that can be added to 1·0 to give a number larger than 1·0? This quantity, known as *machine epsilon*, gives an idea of how precise the system is.

I'd like you to spend the next few minutes using Python interactively to estimate machine epsilon. This will give you an opportunity to familiarise yourself with the Python interpreter. Later today we will write a Python program to calculate an estimate of machine epsilon.

If you have not met the concept of machine epsilon before, you might like to take a look at the Wikipedia entry for machine epsilon for some references and more in-depth information:

[http://en.wikipedia.org/wiki/Machine\\_epsilon](http://en.wikipedia.org/wiki/Machine_epsilon)

# Strings

'Hello, world!'

'''Hello,  
world!'''

"Hello, world!"

""""Hello,  
world!"""

We shall now briefly look at how Python stores text.

Python stores text as “strings of characters”, referred to as “strings”.

## Single quotes

```
'Hello, world!'
```

Single quotes around  
the string

## Double quotes

```
"Hello, world!"
```

Double quotes around  
the string

Exactly equivalent

22

Simple text can be represented as that text surrounded by either single quotes or double quotes. Again, because of the historical nature of keyboards, computing tends not to distinguish opening and closing quotes. The same single quote character, '`',`', is used for the start of the string as for the end and the same double quote character, '`",`', is used for the start and end of a string. However, a string that starts with a single quote must end with a single quote and a string that starts with a double quote must end with a double quote.

'He said "Hello, world!" to her.'

```
>>> print 'He said "Hello, world!" to her.'
```

He said "Hello, world!" to her.

"He said 'Hello, world!' to her."

```
>>> print "He said 'Hello, world!' to her."
```

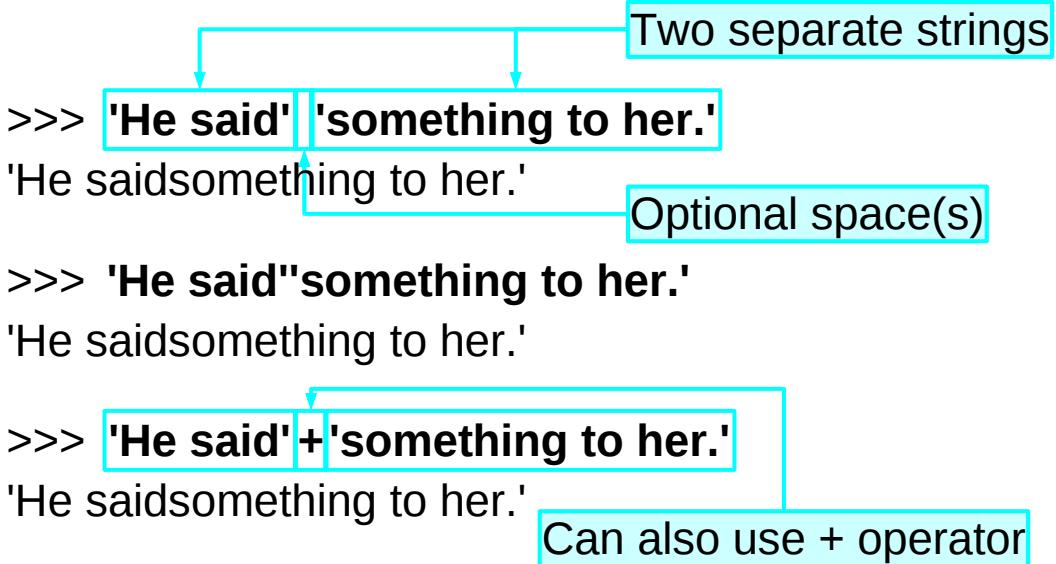
He said 'Hello, world!' to her.

23

The advantage offered by this flexibility is that if the text you need to represent contains double quotes then you can use single quotes to delimit it. If you have to represent a string with single quotes in it you can delimit it with double quotes.

(If you have to have both then stay tuned.)

# String concatenation



24

As you probably know, joining strings together is called “concatenation”. (We say we have “concatenated” the strings.)

In Python we can either do this using the addition operator “+”, or by just typing one string immediately followed by another (optionally separated by spaces). Note that no spaces are inserted between the strings to be joined, Python just joins them together exactly as we typed them, in the order we gave them to it. Note also that the original strings are *not* modified, rather Python gives us a *new* string that consists of the original strings concatenated.

# Special characters

\n → ↴

\' → '

\t → →|

\\" → "

\a → ⚡

>>> print 'Hello,\nworld!'

\\" → \

Hello,  
world!

"\n" converted  
to "new line"

25

Within the string, we can write certain special characters with special codes. For example, the “end of line” or “new line” character can be represented as “\n”. Similarly the instruction to jump to the next tab stop is represented with an embedded “tab” character which is given by “\t”. Two other useful characters are the “beep” or “alarm” which is given by “\a” and the backslash character itself, “\” which is given by “\\”.

This can also be used to embed single quotes and double quotes in a string without interfering with the quotes closing the string. Strictly speaking these aren’t “special” characters but occasionally they have to be treated specially.

# Long strings

Triple double quotes

"""Long pieces of  
text are easier to  
handle if literal new  
lines can be  
embedded in them. """

26

There's one last trick Python has to offer with strings.

Very long strings which cover several lines need to have multiple “\n” escapes within them. This can prove to be extremely awkward as the editor has to track these as the string is edited and recast. To assist with this, Python allows the use of triple sets of double or single quotes to enclose a string, within which new lines are accepted literally.

Single quotes and individual (i.e. not triple) double quotes can be used literally inside triple double quoted strings too.

## Long strings

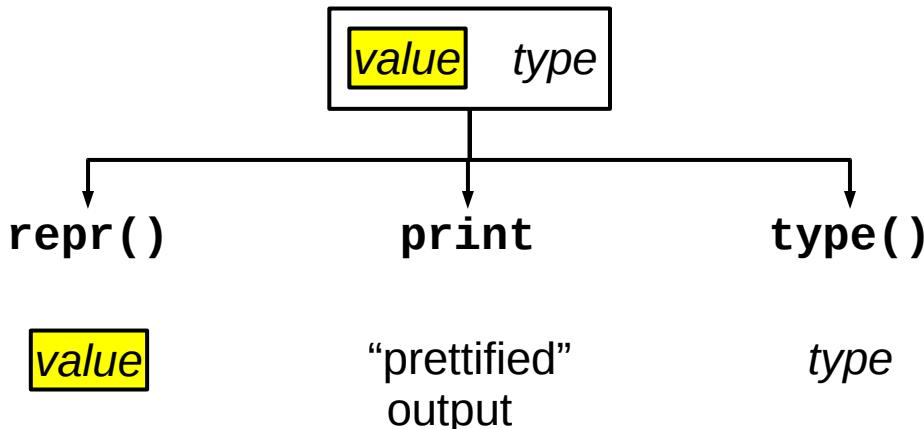
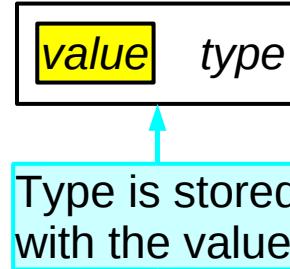
Triple single quotes

''' Long pieces of  
text are easier to  
handle if literal new  
lines can be  
embedded in them. '''

27

Triple double quotes are more common but triple single quotes also work.

# How Python stores values



28

We will take a brief look at how Python stores its values because this is one of the features that distinguishes languages from each other.

Python is a *dynamically typed* language, which means that it decides on the type of a variable (we'll start actually using variables soon) when you assign a value to the variable (technically, it decides on the variable's type at "run-time").

Python is also a *strongly typed* language, which means that once something has been given a type, you can't change that type. (You can, of course, re-define a variable so that it has a different type, but you can't change the type without re-defining it.)

In Python (and most other interpreted languages), a record is kept of what type of value it is (integer, floating point number, string of characters, etc.) alongside the system memory used to store the actual value itself. This means that the type of a variable is determined by the type of the value which that variable contains.

There are three ways of getting at Python's stored values.

The `print` command which we already met outputs a "prettified" version of the value. We will see this prettifying in a moment. We can also instruct Python to give us the type of the value with a function called "`type()`". Finally, we can tell Python to give us the value in a raw, unprettyfied version of the value with a function called "`repr()`" (because it gives the internal representation of the value). The `repr()` function displays the raw version of the value as a *string* (hence when you invoke the `repr()` function the result is displayed surrounded by single quotes). (Note that in certain circumstances the `repr()` function may still do some "prettifying" of values in order to display them as reasonable strings, but usually much less so than `print` does.)

We include parentheses (round brackets) after the names of the "`repr()`" and "`type()`" functions to indicate that, in common with most functions, they require parentheses (round brackets) around their arguments. It's `print` that is the special case here.

```
>>> print 1.2345678901234567  
1.23456789012  
  
>>> type( 1.2345678901234567 )  
<type 'float'>  
  
>>> repr( 1.2345678901234567 )  
'1.2345678901234567'
```

29

For many types there's no difference between the `print`ed output and the `repr()` output. For floating point numbers, though, we can distinguish them because `print` outputs fewer significant figures.

The easiest way to see this is to look at the float 1·2345678901234567. If we `print` it we get output looking like 1·23456789012, but if we apply the `repr()` function we see the full value.

The `type()` function will give the type of a value in a rather weird form. The reasons are a bit technical but what we need to know is that `<type 'float'>` means that this is a floating point number (called a “float” in the jargon).

Again, note that `repr()` and `type()`, in common with most functions, require parentheses (round brackets) around their arguments. It's `print` that is the special case here.

## Two other useful types

Complex

```
>>> (1.0 + 2.0j) * (1.5 + 2.5j)  
(-3.5+5.5j)
```

Boolean

```
>>> True and False  
False
```

```
>>> 123 == 234  
False
```

30

There are two other useful types we need to know about:

- **Complex numbers:** A pair of floats can be bound together to form a complex number (subject to all the caveats applied to floats representing reals). The complex numbers are built into Python, but with the letter “*j*” representing the square root of -1, rather than *i*.

The complex numbers are our first example of a “composite type”, built up out of other, simpler types.

For more details on some of the operations available to you if you are using complex numbers in Python see the “Numeric Types” sub-section of *The Python Standard Library* reference manual:

<http://docs.python.org/library/stdtypes.html#numeric-types-int-float-long-complex>

Python also provides complex number versions of many of the functions in the `math` module in another module, the `cmath` module – for details, type “`help('cmath')`” at the Python prompt or see:

<http://docs.python.org/library/cmath.html>

- **Booleans:** In Python, the truth or falsehood of a statement is a value. These values are of a type that can only take two values: `True` and `False` (with the obvious meanings). This type is called a “Boolean”, named after George Boole, a mathematician who developed the algebra of these sorts of values, and is called a “`bool`” in Python. Zero (whether the integer zero (`0`), floating point zero (`0.0`) or complex zero (`0.0 + 0.0j`)) and the empty string are equivalent to `False`, all other numbers and strings are equivalent to `True`.

Booleans arise as the result of tests, indicating whether the test is passed or failed. For example a Boolean would be returned if we tested to see if two numbers were the same (which we do with “`==`”, as on the slide above). `True` would be returned if they were, and `False` would be returned if they weren’t. And, as we might expect, Booleans can also be combined with `and` or `or`.

30

# Comparisons

```
>>> 1 == 2          >>> 'abc' == 'ABC'  
False                False  
  
>>> 1 < 2          >>> 'abc' < 'ABC'  
True                 False  
  
>>> 1 >= 2         >>> 'abc' >= 'ABC'  
False                True  
  
>>> 1 == 1.0
```

```
True
```

31

This brings us on to the subject of tests. The Python test for two things being equal is a double equals sign, “`==`” (note that there are *no* spaces between the two equal signs). A single equals sign means something else, which we will meet later.

There are also various other comparison tests, such as “greater than” (`>`), “less than” (`<`), “greater than or equal to” (`>=`), and “less than or equal to” (`<=`). You will note that “not equal to” is omitted from this list. We will return to that particular test in a moment.

You will note that string comparisons are done case sensitively, so that the string `'a'` is not the same as the string `'A'`. Strings are compared lexicographically with the lower case letters being greater than the upper case letters. (Python compares the strings lexicographically using their numeric representation as given by the `ord()` function. For example, `ord('a')` = 97 and `ord('A')` = 65, hence `'a' > 'A'`.)

Note also that Python will automatically convert numbers from one type to another (this is known as “*coercion*”) for the purposes of a comparison test. Thus the integer `1` is considered to be equal to the floating point number `1.0`.

## ... not equal to ...

```
>>> not 1 == 2      >>> not 'abc' == 'ABC'  
True                  True
```

```
>>> 1 != 2          >>> 'abc' != 'ABC'  
True                  True
```

32

There are two ways to express “is not equal to”. Because this is a common test, it has its own operator, equivalent to “==”, “>=” etc. This operator is “!=”. However, Python has a more general “is not” operator called, logically enough, “not”. This can precede any expression that evaluates to a Boolean and inverts it; `True` becomes `False` and *vice versa*.

# Conjunctions

```
>>> 1 == 2 and 3 == 3
```

False

```
>>> 1 == 2 or 3 == 3
```

True

33

As mentioned earlier, you can join Booleans with the conjunctions and or or. As you might expect, you can also join tests with these conjunctions.

The “and” logical operator requires both sub-tests to be `True`, so its “truth table” is as shown below:

<i>Test<sub>1</sub></i>	<i>Test<sub>2</sub></i>	<i>Test<sub>1</sub></i> and <i>Test<sub>2</sub></i>
True	True	True
True	False	False
False	True	False
False	False	False

The “or” operator requires only one of them to be `True` (although if they are both `True`, that’s fine as well), so its “truth table” is:

<i>Test<sub>1</sub></i>	<i>Test<sub>2</sub></i>	<i>Test<sub>1</sub></i> or <i>Test<sub>2</sub></i>
True	True	True
True	False	True
False	True	True
False	False	False

Python does so-called “short-circuit evaluation” or “minimal evaluation” – (sometimes (incorrectly) called “lazy evaluation”) – when you combine tests: it will evaluate as few tests as it can possibly manage to still get the final result. So, when you combine two tests with “and”, if the result of the first test is `False`, then Python won’t even bother evaluating the second test, since both “`False and False`” and “`False and True`” evaluate to `False`. Consequently, if the first test evaluates to `False`, the result of combining the two tests with “and” will also be `False`, regardless of whether the second test evaluates to `True` or `False`. Similarly, when you combine two tests with “or”, if the result of the first test is `True`, then Python won’t even bother evaluating the second test, since both “`True or False`” and “`True or True`” evaluate to `True`.

Evaluate the following Python expressions in your head:

```
>>> 2 - 2 == 1 / 2  
>>> True and False or True  
>>> 1 + 1.0e-16 > 1  
>>> 5 == 6 or 2 * 8 == 16  
>>> 7 == 7 / 2 * 2  
>>> 'AbC' > 'ABC'
```

Now try them interactively in Python and see if you were correct.

34

Here's another chance to play with the Python interpreter for yourself.

I want you to evaluate the following expressions in your head, i.e., for each of them decide whether Python would evaluate them as `True` or as `False`. Once you have decided how Python would evaluate them, type them into the Python interpreter and see if you were correct.

If you have any problems with this exercise, or if any questions arise as a result of the exercise, please ask the course giver or a demonstrator.

Give yourself 5 minutes or so to do this exercise (or as much as you can manage in that time) and then take a break. Taking regular breaks is very important when using the computer for any length of time. (Note that “take a break” means “take a break from this computer” **not** “take a break from this course to check your e-mail”).

In case you are wondering in what order Python evaluates things, it uses similar rules for determining the order in which things are evaluated as most other programming languages. We'll look briefly at this order after the break.

# Precedence

First	$x**y$ $-6, +6$ $x/y, x*y, x\%y$ $x+y, x-y$ $x < y, x \leq y, \dots$ $x \text{ in } y, x \text{ not in } y$ $\text{not } x$ $x \text{ and } y$	Arithmetic operations
Last	$x \text{ or } y$	

35

Having now got Python to evaluate various expressions, you may be wondering about the order in which Python evaluates operations within an expression. Python uses similar rules for *precedence* (i.e. the order in which things are evaluated) as most other programming languages. A summary of this order is shown on the slide above.

Python expressions are evaluated from left to right, but with operators being evaluated in order of precedence (highest to lowest). First any arithmetic operations are evaluated in order of precedence, then any comparisons ( $<$ ,  $\text{==}$ , etc.) and then any logical operations (such as `not`, `and`, etc.).

(Note that we have not yet met all of the operations shown on the slide above.)

You can also find a summary of the precedence order of all the operators in Python in the Python documentation, although note that the documentation lists the operators in ascending order of precedence (i.e. from lowest precedence to highest precedence, rather than from highest to lowest as we have done here):

<http://docs.python.org/reference/expressions.html#evaluation-order>

In common with most other programming languages you can change the order in which parts of an expression are evaluated by surrounding them with parentheses (round brackets), e.g.

```
>>> 2 + 3 * 5
```

17

```
>>> (2 + 3) * 5
```

25

# Flow control in Python: `if`

```
if x > 0.0 :  
    print 'Positive'  
  
elif x < 0.0 :  
    print 'Negative'  
    x = -1.0 * x  
  
else :  
    print 'Zero'
```

The diagram illustrates the flow control of a Python `if` statement. It shows three clauses: `if`, `elif`, and `else`. The `if` clause has a mandatory action (print 'Positive'). The `elif` clause has an optional, repeatable action (print 'Negative') and an assignment (x = -1.0 \* x). The `else` clause has an optional action (print 'Zero'). Indentation is indicated by arrows pointing from the left to the code lines. Brackets group the clauses into sets, with labels indicating their properties: 'compulsory' for the `if` clause, 'optional, repeatable' for the `elif` clause, and 'optional' for the `else` clause. A bracket also groups all three clauses as 'multiple lines indented'.

36

Python has several constructs for controlling the flow of your program. The first one we will look at is the `if` statement.

As one might expect, the `if` statement specifies a test and an action (or series of actions) to be taken if the test evaluates to `True`. (We will refer to these actions as the “`if`” block.)

You can then have as many (or as few) “`else if`” clauses as you wish. Each “`else if`” clause specifies a further test and one or more actions to carry out if its test evaluates to `True`. These “`else if`” clauses are introduced by the keyword “`elif`”. (We will refer to the actions specified in an “`else if`” clause as an “`elif`” block.)

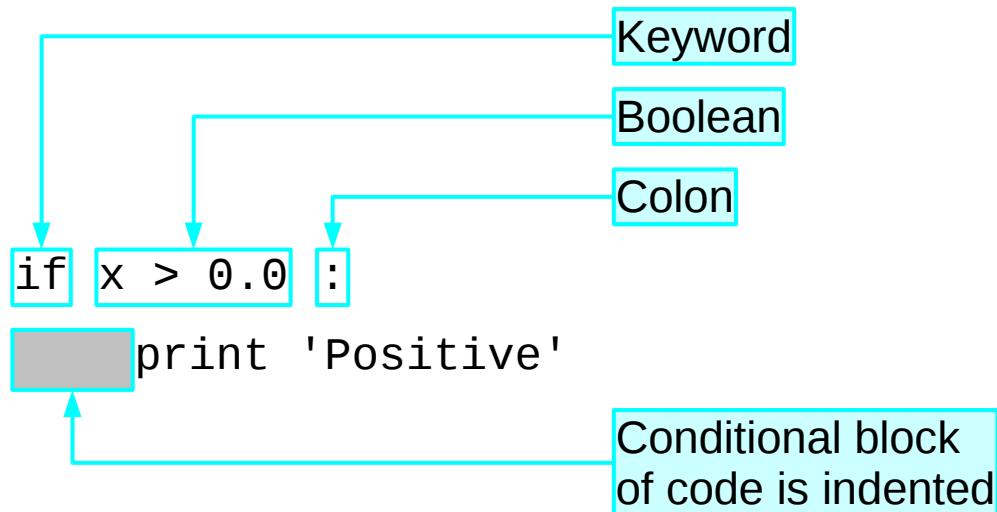
In order for an “`else if`” clause to be evaluated, the test specified by `if` must evaluate to `False`, and all the tests of any preceding “`else if`” clauses that are part of the `if` statement must also evaluate to `False`.

Finally, you can optionally have an “`else`” clause that specifies an action to be taken if *all* preceding tests in the `if` statement (including any from any “`else if`” clauses) evaluated to `False`. This “`else`” clause is introduced by the keyword “`else`” immediately followed by a colon (:). We will refer to the actions specified in this clause as the “`else`” block.

You will note that each of the lines starting “`if`”, “`elif`” or “`else`” end in a colon (:). This is standard Python syntax that means “I’ve finished with this line, and all the following indented lines belong to this section of code”. Python uses **indentation** to group related lines of code. Whenever you see a colon (:) in a Python script, the next line **must** be indented.

The amount of indentation doesn’t matter, but all the lines in a block of code must be indented by the same amount (except for lines in any sub-blocks, which will be further indented). If this seems a bit strange compare it with official documents with paragraphs, sub-paragraphs and sub-sub-paragraphs, each of which is indented more and more.

# Flow control in Python: `if`



37

Since this is the first flow control construct we've met in Python let's examine it in a bit more closely. This will also give us a chance to get to grips with Python's use of indentation.

The first line of an `if` statement is of the form "`if test:`".

The test is preceded with the Python keyword "`if`". This introduces the test and tells the Python interpreter that a block of code is going to be executed or not executed according to the evaluation of this test. We shall refer to this block of code as the "`if`" block.

The test itself is followed by a colon (`:`). This is standard Python to indicate that the test is over and the conditionally executed block is about to start. It ends the line.

The "`if test:`" line is followed by the conditional block of Python. This is indented (typically by a small number of space characters or a tab stop). Every line in that block is indented by the same amount (unless it contains sub blocks of its own which are indented further relative to it). Python uses indentation to mark the block. As soon as the indentation has stopped the "`if`" block ends and a new code block starts.

## Nested indentation

```
if x > 0.0 :  
    print 'Positive'  
  
else :  
    if x < 0.0 :  
        print 'Negative'  
        x = -1.0 * x  
    else :  
        print 'Zero'
```

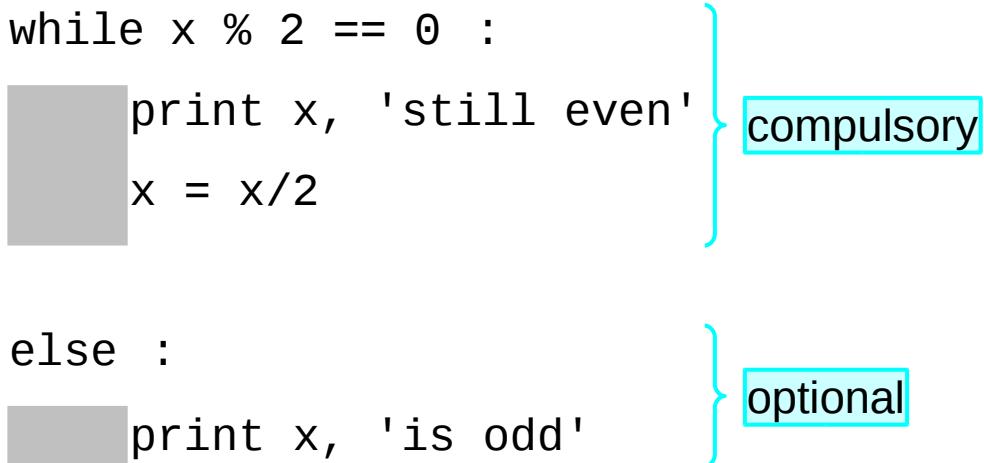
38

As mentioned earlier, if we have any sub-blocks, they will be further indented, as seen in the example above.

In the above example, in our “else” block we have a new `if` statement. The conditional blocks of this `if` statement will be further indented as they are sub-blocks of the “else” block.

# Flow control in Python: `while`

```
while x % 2 == 0 :  
    print x, 'still even'  
    x = x/2  
  
else :  
    print x, 'is odd'
```



39

The next flow control construct we will look at is a loop, the `while` loop, that does something as long as some test evaluates to `True`.

The idea behind this looping structure is for the script to repeat an action (or series of actions) as long as the specified test evaluates to `True`. Each time the script completes the action (or series of actions) it evaluates the test again. If the result is `True`, it repeats the action(s); if the result is `False` it stops. (We will refer to the action (or series of actions) as the “`while`” block.)

The syntax for the `while` loop is “`while test:`”.

This is immediately followed by the block of Python (which can be one or more lines of Python) to be run repeatedly until the test evaluates to `False` (the “`while`” block).

There is also an optional “`else`” clause which, if present, is executed when the test specified in the `while` loop evaluates to `False`, i.e. at the end of the `while` loop. In practice, this “`else`” clause is seldom used. As with the `if` statement, the “`else`” clause is introduced by “`else:`” followed by an indented “`else`” block.

(The reason that you might want to use the “`else`” block is that if your program “breaks out” of a `while` loop with the `break` statement then the “`else`” block will be skipped. For this introductory course, we’re not going to bother with the `break` statement.)

```

#!/usr/bin/python

epsilon = 1.0

while 1.0 + epsilon > 1.0:
    epsilon = epsilon / 2.0

epsilon = 2.0 * epsilon

print epsilon

```

epsilon.py

Approximate  
machine  
epsilon

$$1.0 + \epsilon > 1.0$$

$$1.0 + \epsilon/2 == 1.0$$

40

To illustrate the `while` loop we will consider a simple script to give us an approximation to machine epsilon. Rather than trying to find the smallest floating point number that can be added to 1·0 without changing it, we will find a number which is large enough but which, if halved, isn't. Essentially we will get an estimate between  $\epsilon$  and  $2\epsilon$ .

We start (before any looping) by setting an approximation to the machine epsilon that's way too large. We'll take 1.0 as our too large initial estimate:

`epsilon = 1.0`

We want to keep dividing by 2.0 so long as our estimate is too large. What we mean by too large is that adding it to 1.0 gives us a number strictly larger than 1.0. So that's the test for our `while` loop:

`1.0 + epsilon > 1.0`

What do we want to do each time the test is passed? We want to decrease our estimate by halving it. So that's the body of the `while` loop:

`epsilon = epsilon / 2.0`

Once the test fails, what do we do? Well, we now have the estimate one halving too small so we double it up again. Once we have done this we had better print it out:

`epsilon = epsilon * 2.0`  
`print epsilon`

And that's it!

You can find this script in your course home directories as `epsilon.py`.

PS: It works.

If you run this script on different machines you may get different answers. Machine epsilon is machine specific, which is why it is called *machine* epsilon and not “floating point epsilon”.

```
#!/usr/bin/python

# Start with too big a value
epsilon = 1.0

# Halve it until it gets too small
while 1.0 + epsilon > 1.0:
    epsilon = epsilon / 2.0

# It's one step too small now,
# so double it again.
epsilon = 2.0 * epsilon

# And output the result
print epsilon
```

epsilon.py

41

As I'm sure you all know, it is very important that you comment your code while you are writing it. If you don't, how will you remember what it does (and why) next week? Next month? Next year?

So, as you might expect, Python has the concept of a “comment” in the code. This is a line which has no effect on what Python actually does but just sits there to remind the reader of what the code is for.

Comments in Python are lines whose first non-whitespace character is the hash symbol “#”. Everything from the “#” to the end of the line is ignored.

(You can also put a comment at the end of a line: in this case, anything that follows the hash symbol (#) until the end of the line will be ignored by Python.)



## Time for a break...

Have a look at the script `epsilon2.py` in your home directory.

This script gives a better estimate of machine than the script we just wrote.

See if you can figure out what it does – if there is anything you don't understand, tell the course giver or a demonstrator.

42

Examine the script `epsilon2.py` in your course home directories to see a superior script for approximating machine epsilon. This uses nothing that we haven't seen already but is slightly more involved. If you can understand this script then you're doing fine.

You should also run the script and test its output.

And then it's time for a break. As you should all know, you need to look after yourselves properly when you're using a computer. It's important to get away from the computer regularly.

You need to let your arms and wrists relax. Repetitive strain injury (RSI) can cripple you.

Your eyes need to have a chance to relax instead of focussing on a screen. Also while you are staring intently at a screen your blink rate drops and your eyes dry out. You need to let them get back to normal.

You also need to let your back straighten to avoid long term postural difficulties. Computing screws you up, basically.

```

#!/usr/bin/python

too_large = 1.0
too_small = 0.0
tolerance = 1.0e-27

while too_large - too_small > tolerance:

    mid_point = (too_large + too_small)/2.0

    if 1.0 + mid_point > 1.0:
        too_large = mid_point
    else:
        too_small = mid_point

print too_small, '< epsilon <', too_large

```

A better estimate for machine epsilon

`epsilon2.py`

43

This is the `epsilon2.py` script you were asked to look at before the break.

There are two quite distinct aspects to understanding it: *what* the Python does, and *why* it does it.

In terms of what it does, this is a relatively simple script. It sets up three values at the start and then runs a `while` loop which modifies one of those values at a time until some condition is broken. Inside the `while` loop a new value is calculated and then an `if` statement is run depending on some other test. One value or another is changed according to the results of that test. Once the `while` loop is done it prints out three values on a line. The Python contains an `if` statement nested inside a `while` loop but other than that it is quite straightforward.

It's what is does that's the clever bit. As many of you will know, the posh, computer-y name for "what it does" is *algorithm*.

The algorithm here uses a technique called *bisection*. We start with a range of real numbers which we know must contain machine epsilon. Then we calculate the mid-point of that range and ask if that number is bigger or smaller than the machine epsilon. If the estimate is too small we can change the lower bound of our interval to the value of this mid-point. If it is too large we can change the upper bound of our interval to this mid-point estimate. Either way we get an interval that contains the real value that is half the length of the interval we started with (i.e., the interval has been *bisected*). This is the `if` statement of our Python. We can repeat this to halve the length of our interval time and time again until the length of the interval is short enough for us. This is the `while` statement in our Python script.

Some of you may be wondering why we keep going until the length of our interval is less than  $10^{-27}$ . Where does the value of  $10^{-27}$  come from? This value was chosen because, on the PCs we are using for this course, if the interval is any smaller than this then when we use `print` to display its lower and upper bounds there will be no difference on the screen between the two values displayed. This is because `print` will not display these bounds to enough significant figures for us to see that they are different from each other if the difference between them is less than  $10^{-27}$ . (Recall that `print` may "prettyify" values; in particular, for floating point numbers it only displays a certain number of significant figures. Note that, as machine epsilon is *machine specific*, on other computers you may need to use a higher or lower value than  $10^{-27}$ .)

January February March April May June July  
August September October November December

# Lists

H He Li Be B C N O F Ne Na Mg Al Si P S Cl Ar

Red Orange Yellow Blue Indigo Violet

44

We've already met various simple data types in Python. We've also seen that Python has "composite" types, built up of the simpler data types, for example, complex numbers are such a composite type (built up of two floating point numbers).

Python also has a more general type designed to cope with arbitrary lists of values and it is this type and a corresponding control structure that we will now discuss.

The Python data type for storing a collection of sequential, related data is the "list". Unlike a complex number – which always contains two floating point numbers – a list can contain any number of items, and the items can be of any type. The items in a list don't all have to be of the same type, although it is generally a bad idea to have lists whose items are of different types. (If you need to do this, you should use a different structure, known as a "tuple", instead. We'll be meeting tuples later.)

Python's lists are roughly equivalent to most other languages' arrays, except that most languages require their arrays to be of fixed length and all the items in the array to be of the same type.

(The individual items in the list (or indeed in any type of sequence) are often referred to as the *elements* of the list (or sequence), although we will try to avoid using this terminology, as it can be confusing in a scientific context, where "element" usually means something different, e.g. the atomic elements given in the periodic table.)

```
>>> [ 2, 3, 5, 7, 11, 13, 17, 19]  
[ 2, 3, 5, 7, 11, 13, 17, 19]  
  
>>> type([ 2, 3, 5, 7, 11, 13, 17, 19])  
<type 'list'>  
  
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```

45

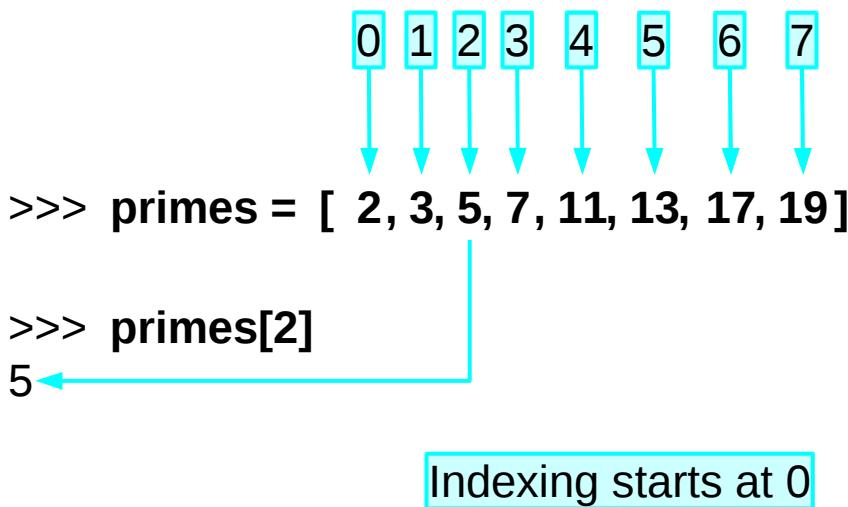
So how do we create lists in Python?

The syntax for representing a list is relatively straightforward. The items forming the list appear in order, are separated by commas and are surrounded by square brackets ([]). (The brackets must be square rather than any of the other sorts of brackets you may have at your disposal on your keyboard.)

Do note that this is not just syntactic sugar. A list is a genuine Python type, on a par with integers or floating point numbers.

As mentioned earlier, the type here is just “list” and not “list of integers”. Some languages are more prescriptive about this sort of type; Python isn’t.

When working with lists, it is very common to pick variable names for our lists that are the plural of whatever is in the list. So we might use `prime` as the name of the variable that corresponds to a single prime number from the list and we would use `primes` as the name of the variable corresponding to the list itself.



46

As with most languages that have this sort of data type, we start counting the items in the list from zero, not one. So what we would regard as “the third item in the list” is actually referred to by Python as “item number *two*”. The number of the item in the list is known as its *index*.

The way to get an individual item from a list is to take the list, or the variable name corresponding to the list, and to follow it with the index of the item wanted in square brackets ([ ]), as shown in the example on the slide above. (Note that these are square brackets because that’s what Python uses around indices, not square brackets because that’s what lists use.)

```
0 1 2 3 4 5 6 7  
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]  
19  
>>> primes[-1]  
-8 -7 -6 -5 -4 -3 -2 -1
```

47

Perhaps slightly surprisingly, we can ask for the “minus first” item of a list. Python has a dirty trick that is occasionally useful where, if you specify a negative index, it counts from the end of the list.

This negative index trick can be applied all the way back in the list.

If a list has eight items (indexed 0 to 7) then the valid indices that can be asked for run from -8 to 7.

A diagram showing a list of prime numbers: [2, 3, 5, 7, 11, 13, 17, 19]. Above the list, indices 0 through 7 are shown in boxes, each with a blue arrow pointing down to its corresponding list element. The list elements are also enclosed in boxes.

```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```

```
>>> primes[8]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Where the error was.

The error message.

48

So what happens if we ask for an invalid index? We get an error.

As this is our first error message we will take our time and inspect it carefully.

The first line declares that what you are seeing is a “traceback” with the “most recent call last”.

A traceback is the command’s history: how you got to be here making this mistake, if you like.

The error itself will come at the end. It will be preceded with how you got to be there. (If your script has jumped through several hoops to get there you will see a list of the hoops.) In our example we jumped straight to the error so the traceback is pretty trivial.

The next two lines are the traceback. In more complex examples there would be more than two but they would still have the general structure of a “file” line followed by “what happened” line.

The file line says that the error occurred in a file called “<stdin>”. What this actually means is that the error occurred on Python being fed to the interpreter from “standard input”. Standard input means your terminal. You typed the erroneous line and so the error came from you.

Each line at the “>>>” prompt is processed before the prompt comes back. Each line counts as “line 1”.

If the error had come from a script you would have got the file name instead of “<stdin>” and the line number in the script.

The “<module>” refers to the module or function you were in. We haven’t done modules or functions yet so you weren’t in either a module or a function.

The third line gives information about the error itself. First comes a description of what type of error has happened. This is followed by a more detailed error message. If the error had come from a script the line of Python would be reproduced too.

## Counting from zero and the `len()` function

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

```
>>> primes[0]
```

```
2
```

```
>>> primes[7]
```

```
19
```

```
>>> len(primes)
```

```
8
```

`0 ≤ index ≤ 7`

`length 8`

49

We can confirm that Python does count from zero for list indices by asking explicitly for item number zero.

The `len()` function returns the number of items in (i.e. the **length** of) a list.

# Changing an item in a list

```
>>> data = [56.0, 49.5, 32.0]
      ↑
>>> data[1] ← "item number 1" ("2nd item")
49.5
      ↑
>>> data[1] = 42.25 ← Assign new value to
      ↑           "item number 1" in list
>>> data   [56.0, 42.25, 32.0] ← List is modified "in place"
```

50

As you would expect, we can easily change the value of a particular item in a list provided we know its index. The syntax is:

`listname[index] = new_value`

Note that this changes the list itself (i.e. the list is modified in place), it does not create a new list with the new value and then make the old list equal to this new list.

Note also that this is **not** a way to add a new item to the end of a list. (We will see how to do that in a little while.)

# Empty lists

```
>>> empty = [ ]
```

```
>>> len(empty)  
0
```

```
>>> len([ ])  
0
```

51

Note that it is quite possible to have an empty list. The list `[ ]` is perfectly valid in Python.

A common trick for creating a list is to start with an empty list and then to append items to it one at a time as the program proceeds.

(Note that in the slide above we could have used any variable name for our empty list; we didn't have to call it `empty`. There's nothing special about the word "empty" in Python.)



## Single item lists

*A list with one item is not the same as the item itself!*

```
>>> [1234] == 1234  
False
```

```
>>> type([1234])  
<type 'list'>
```

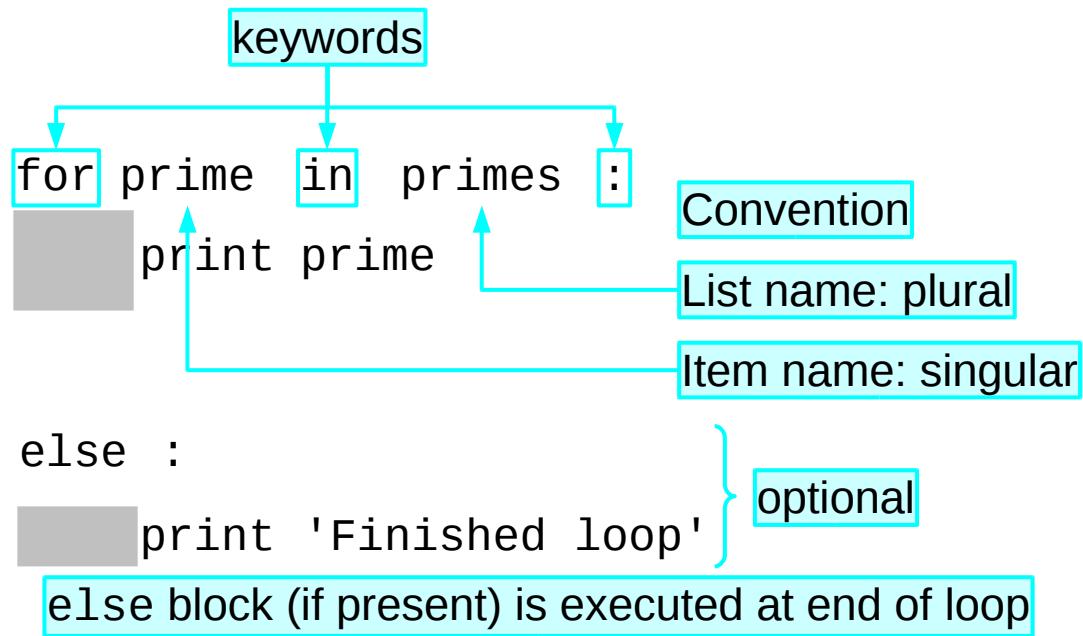
```
>>> type(1234)  
<type 'int'>
```

52

It is also possible to have a list which only has a single item.

Please note that a list with one item in it is quite different from the item itself. Some languages deliberately confuse the two. While it may make some simple tricks simpler it makes many, many more things vastly more complicated.

# Flow control in Python: `for`



There is a special loop construct, the `for` loop, that does something for every item in a list.

The idea behind this looping structure is for the script to get through the list, one item at a time, asking if there are any items left to go. If there are a variable (known as the “loop variable”) gets set to refer to the current item in the list and a block of code is run. Then it asks whether there are any more items left to go and, if so, runs the same block of code, and so on until there are no more items left in the list.

The syntax for the `for` loop is very similar to the `while` loop but instead of “`while test:`” we have “`for item in list:`”.

The words “`for`” and “`in`” are just syntax. The word immediately after “`for`” is the name of the “loop variable” – the variable that is going to be used to track the items in the list that appears immediately after “`in`”.

The block of Python (typically several lines of Python rather than just one) to be run for every item in the list is indented just as it was for the `while` loop.

There is also an optional “`else`” block which, if present, is executed at the end of the `for` loop, i.e. after it has finished processing the items in the list. In practice, this “`else`” block is almost never used. As with the `while` loop, the “`else`” clause is introduced by “`else:`” followed by an indented “`else`” block.

(The reason that you might want to use the “`else`” block is that if your program “breaks out” of a `for` loop with the `break` statement then the “`else`” block will be skipped. For this introductory course, we’re not going to bother with the `break` statement.)



## Warning: loop variable persists

Definition of loop variable

```
for prime in primes :  
    print prime ← Correct use of loop variable  
print 'Done!'  
print prime  
↑  
Improper use of loop variable  
But legal!
```

54

And now a warning.

The loop variable, `prime` in the example above, persists after the loop is finished, which means we can reuse it. This is not a good idea, as in long scripts we can easily confuse ourselves if we have variables whose names are the same as any of our loop variables. We created the loop variable for use in our `for` loop, and we really shouldn't use it anywhere else.

# Loop variable “hygiene”

```
for prime in primes :  
    print prime  
del prime  
print 'Done!'
```

Annotations:

- Create loop variable: points to the word "prime" in the first line of code.
- Use loop variable: points to the word "prime" in the second line of code.
- Delete loop variable: points to the word "prime" in the third line of code.

55

This brings us to a Python operator which we will use to encourage code cleanliness: `del`. The `del` operator deletes a variable from the set known about by the interpreter.

The well-written `for` loop has a self-contained property about itself.

Before the loop starts the interpreter knows about the list and not the loop variable. The “`for`” line defines an extra variable, the loop variable, which is then used in the loop. Using the `del` operator, we can delete this loop variable on completion of the `for` loop. So now after the loop the interpreter is exactly the same state as it was in before; it knows about the list but not the loop variable. The loop has left the system “unscathed”.

```
#!/usr/bin/python

# This is a list of numbers we want
# to add up.
weights = [ 0.1, 0.5, 2.6, 7.0, 5.3 ]

# Add all the numbers in the list
# together.

# Print the result.
print
```

What goes here?

addition.py

56

And now for an exercise.

In your course home directories you will find an incomplete script called `addition.py`.

Complete the script so that it adds up all the numbers in the list `weights` and prints out the total. Obviously, one way of doing this would be to manually type the numbers in the list out as a long sum for Python to do. That would not be a very sensible way of doing things, and you wouldn't learn very much by doing it that way. I suggest you try something else.

If you have any questions about anything we've covered so far, now would be a good time to ask them.

## Answer

```
#!/usr/bin/python

# This is a list of numbers we want
# to add up.
weights = [ 0.1, 0.5, 2.6, 7.0, 5.3 ]

# Add all the numbers in the list
# together.
total = 0.0
for weight in weights:
    total = total + weight
del weight

# Print the result.
print total
```

addition.py

57

And above is a solution to the exercise.

...If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

# Lists of anything

```
primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

List of integers

```
names = [ 'Alice', 'Bob', 'Cathy', 'Dave' ]
```

List of strings

```
roots = [ 0.0, 1.57079632679, 3.14159265359 ]
```

List of floats

```
lists = [ [ 1, 2, 3 ], [5], [9, 1] ]
```

List of *lists*

58

We've mentioned already that the Python type is "list" rather than "list of integers", "list of floating point numbers", etc. It is possible in Python to have lists of anything (including other lists).

## Mixed lists

```
stuff = [ 2, 'Bob', 3.14159265359, 'Dave' ]
```



Legal, but not a good idea.

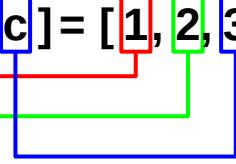
See “tuples” later.

59

It is also possible to have lists with mixed content where the 1<sup>st</sup> item in the list is an integer, the 2<sup>nd</sup> a string, the 3<sup>rd</sup> a floating point number, etc. However, it is rarely a good idea. If you need to bunch together collections of various types then there is a better approach, called the “tuple”, that we will meet later.

# Lists of variables

```
>>> [a, b, c] = [1, 2, 3]
```



```
>>> a
```

```
1
```

```
>>> b
```

```
2
```

```
>>> c
```

```
3
```

60

We can also have lists of variables, both to contain values and to have values assigned to them.

A list of variables can be assigned to from a list of values, so long as the number of items in the two lists is the same.

# All or nothing

Traceback: where  
the error happened

```
>>> [d, e, f] = [1, 2, 3, 4]
```

Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
ValueError: too many values to unpack

```
>>> d
```

Error message

Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
NameError: name 'd' is not defined

61

However, if there are too few variables (or too many values) no assignment is done.

```
>>> [d, e, f] = [1, 2, 3, 4]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: too many values to unpack  
>>> d  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'd' is not defined  
>>> e  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'e' is not defined  
>>> f  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'f' is not defined  
>>>
```

Note that we do not get d, e and f assigned with an error for the dangling 4. We get no assignment at all.

# All or nothing

```
>>> [g, h, i, j] = [1, 2, 3]
```

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

    ValueError: need more than 3 values to unpack

```
>>> g
```

Error message

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

    NameError: name 'g' is not defined

62

Similarly, if there are too many variables (or too few values) no assignment is done.

```
>>> [g, h, i, j] = [1, 2, 3]
```

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

    ValueError: need more than 3 values to unpack

```
>>> g
```

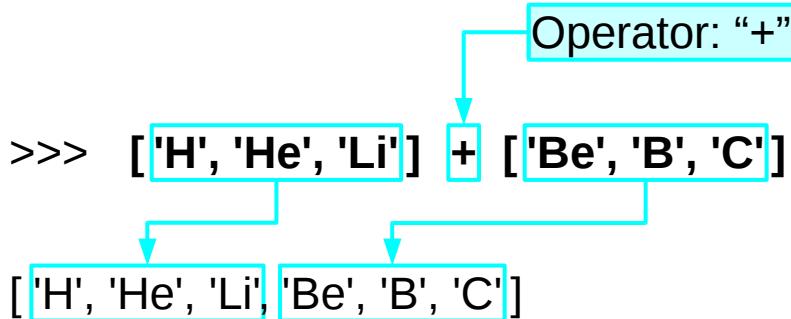
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

    NameError: name 'g' is not defined

```
>>>
```

# Concatenating lists



63

Python uses the “+” character for adding two lists together (adding two lists together is called “concatenating” the lists).

Please note that lists are not the same as sets. If an item appears in two lists being concatenated then the item appears twice in the new list.

```
>>> ['H', 'He', 'Li'] + ['Li', 'Be', 'B']  
['H', 'He', 'Li', 'Li', 'Be', 'B']
```

Similarly, order matters. Concatenating list A with list B is not the same as concatenating B with A.

```
>>> ['H', 'He', 'Li'] + ['Be', 'B', 'C']  
['H', 'He', 'Li', 'Be', 'B', 'C']  
>>> ['Be', 'B', 'C'] + ['H', 'He', 'Li']  
['Be', 'B', 'C', 'H', 'He', 'Li']
```

It's worth noting in passing that concatenating two lists takes the two lists and creates a third. It does not modify one list by adding the other list's items at the end.

# Appending an item: `append()`

```
>>> symbols = [ 'H', 'He', 'Li', 'Be' ]  
>>> symbols  
[ 'H', 'He', 'Li', 'Be' ]  
  
>>> symbols.append('B')  
None  
>>> symbols  
[ 'H', 'He', 'Li', 'Be', 'B' ]
```

Diagram annotations:

- “appending is a “method”” points to the word `append` in the command `symbols.append('B')`.
- “the item to append” points to the argument `'B'` in the command `symbols.append('B')`.
- “no value returned” points to the empty box after `None`.
- “the list itself is changed” points to the final state of the list `[ 'H', 'He', 'Li', 'Be', 'B' ]`.

64

This may trigger a sense of déjà vu. We've added an item to the end of a list. Isn't this what we have already done with concatenation?

No it isn't, and it's important to understand the difference. Concatenation joined two *lists* together. Appending (which is what we are doing here) adds an *item* to a list. Recall that a list of one item and the item itself are two completely different things in Python.

Appending lets us see a very common Python mechanism so we will dwell on it for a moment. Note that the Python command is *not* “`append(symbols, 'B')`” or “`append('B', symbols)`” but rather it is “`symbols.append('B')`”.

The `append()` function can only work on lists. So rather than tempt you to use it on non-lists by making it generally available it is built in to lists themselves (the technical term for this is *encapsulation*). Almost all Python objects have these sorts of built in functions (called “methods” in the jargon) but appending an item to a list is the first time we have encountered them.

A method is a function built in to a particular type so all items of that type will have that method. You, the programmer, don't have to do anything. The name of the method follows the thing itself separated by a dot. In all other ways it is exactly like a function that uses brackets, except that you don't need to put the object itself in as an argument.

This looks like a lot of fuss for no gain. In practice, this sort of organisation makes larger, more complex scripts vastly easier to manage. Of course, we're still at the stage of writing very simple scripts so we don't see that benefit immediately.

Note that what the `append()` method does is change the actual list itself, it does not create a new list made up of the original list with a new item added at the end. As `append()` changes the list itself, it does not return a value when we use it, but just silently updates the list “in place”.

# Membership of lists

keyword: “in”

```
>>> 'He' in [ 'H', 'He', 'Li' ]
```

True

```
>>> 'He' in [ 'Be', 'B', 'C' ]
```

False

65

We need a means to test to see if a value is present in a list.

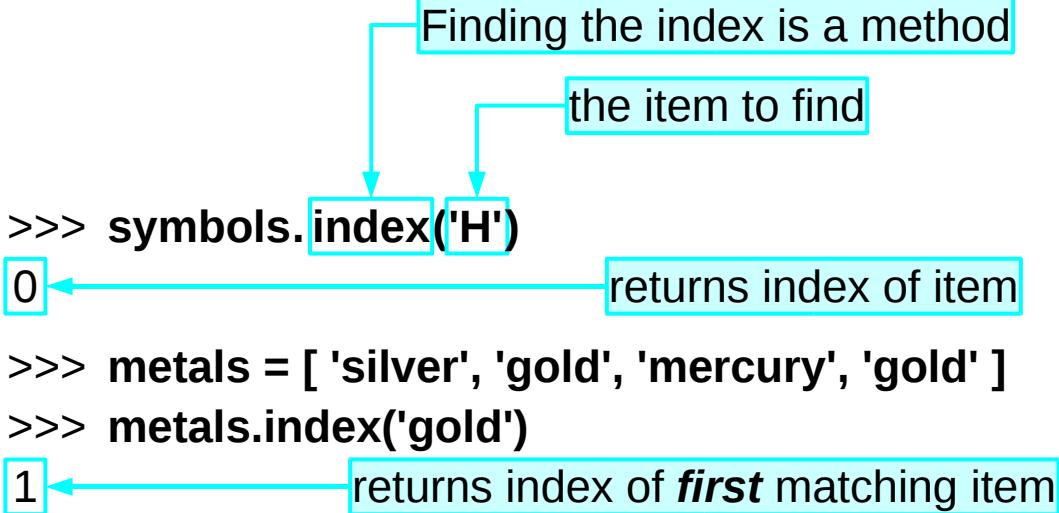
The keyword “in”, used away from a “for” statement, is used to test for presence in a list. The Python expression

*item in list*

evaluates to a Boolean depending on whether or not the item is in the list.

# Finding the index of an item

```
>>> symbols = [ 'H', 'He', 'Li', 'Be' ]
```



66

The `index()` method returns the **index** of the *first* matching item in a list (there must be at least one matching item or you will get an error). For example:

```
>>> data = [2, 3, 5, 7, 5, 12]
>>> data.index(5)
2
```

There are also some other methods of lists which may be of interest. The `remove()` method **removes** the *first* matching item from a list (there must be at least one matching item or you will get an error). As with `append()`, `remove()` works by modifying the list itself, not by creating a new list with one fewer item. For example:

```
>>> symbols = ['H', 'He', 'Li', 'Be', 'B', 'Li']
>>> symbols.remove('Li')
>>> symbols
['H', 'He', 'Be', 'B', 'Li']
```

The `insert()` method **inserts** an item into a list at the given position. It takes two arguments: the first is the index of the list at which the item is to be inserted, and the second argument is the item itself. As with `append()`, `insert()` works by modifying the list itself, not by creating a new list with an extra item. For example:

```
>>> symbols = ['H', 'He', 'Be', 'B']
>>> symbols.insert(2, 'Li')
>>> symbols
['H', 'He', 'Li', 'Be', 'B']
```

Lists have a number of other methods, and appended to these notes is a guide to many of them.

# Functions that give lists: `range()`

```
>>> range(0, 8)  
[0, 1, 2, 3, 4, 5, 6, 7]
```

First integer  
in list

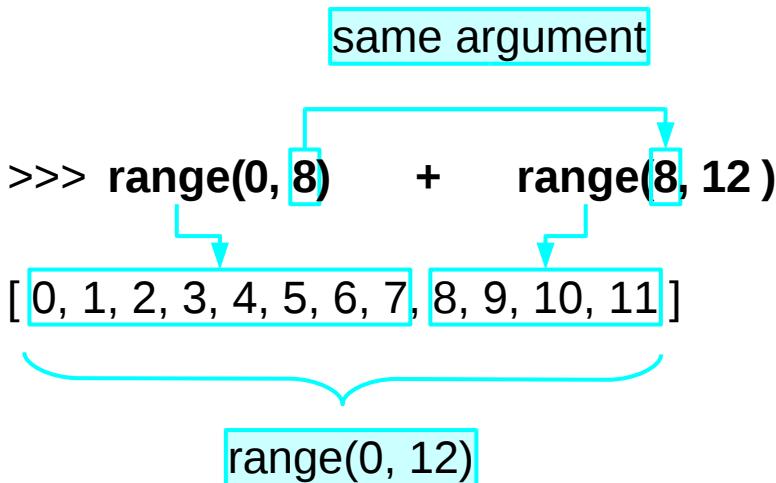
One beyond  
last integer  
in list

67

We have been happily manipulating lists. However, apart from entering them all manually, how do we get them? We cover this next as we look at some of the functions that return lists as their results.

One of the simplest built-in functions to give a list is the function `range()`. This takes two integers and gives a list of integers running from the first argument to one below the second.

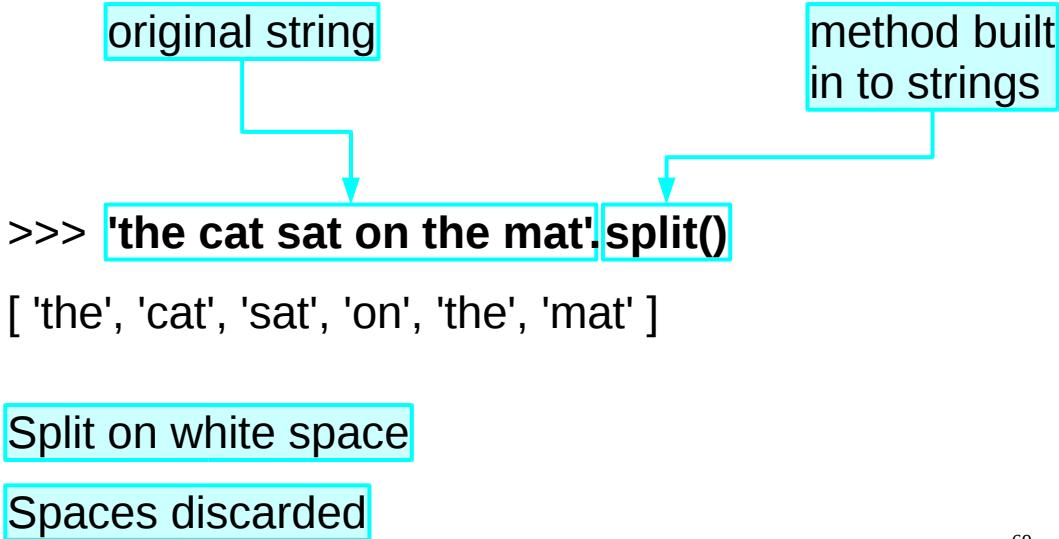
# `range( )`: Why miss the last number?



68

This business of “stopping one short” seems weird at first glance, but does have a purpose. If we concatenate two ranges where the first argument of the second range is the second argument of the first one then they join with no duplication of gaps to give a valid range again.

# Functions that give lists: `split()`



Another very useful function that gives a list is “splitting”. This involves taking a string and splitting it into a list of sub-strings. The typical example involves splitting a phrase into its constituent words and discarding the spaces between them.

Note that `split()` is a method of strings. It returns a list of strings which are the words in the original string, with the white space between them thrown away. It is possible to use `split()` to chop on other than white space. You can put a string in the arguments of `split()` to tell it what to cut on (rather than white space) but we advise against it. It is very tempting to try to use this to split on commas, for example. There are much better ways to do this and we address them in the course “Python: Regular Expressions”. Stick to splitting on white space.

For details of the “Python: Regular Expressions” course, see:

<http://training.csx.cam.ac.uk/course/pythonregexp>

# **split( )**: Only good for trivial splitting

```
>>> 'the cat sat on the mat'.split()
```

```
[ 'the', 'cat', 'sat', 'on', 'the', 'mat' ]
```

Split on white space

Spaces discarded

Trivial operation

Regular expressions

Comma separated values

Use the specialist  
Python support  
for these.

70

To re-iterate: the `split( )` method is very, very primitive.

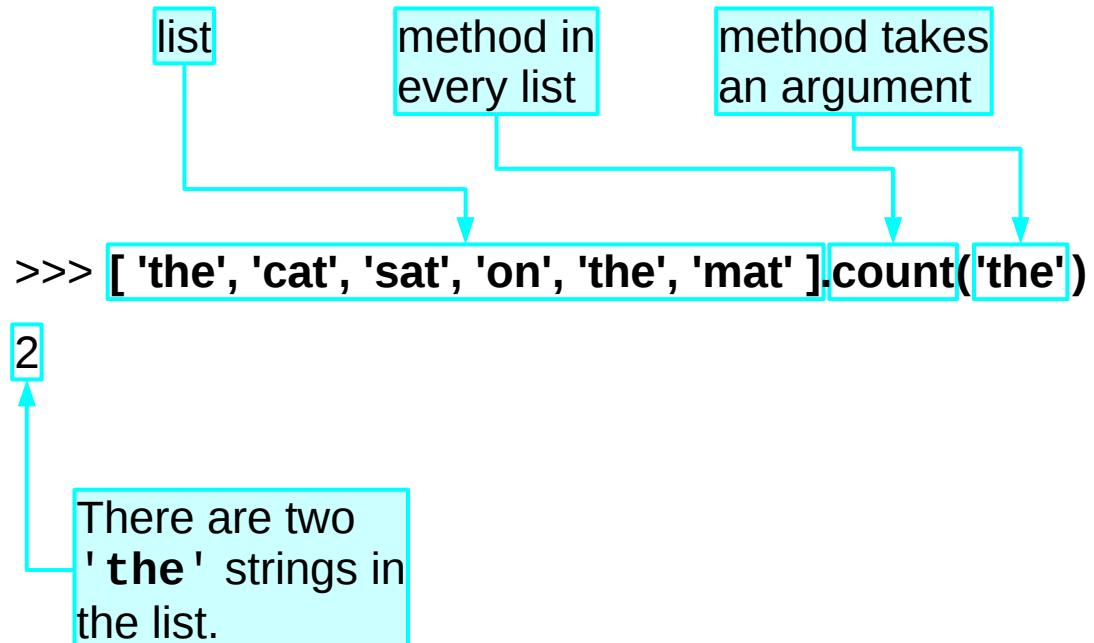
There are many better approaches, such as using Python's support for regular expressions or for CSV (comma separated values) files.

Python's support for CSV files is covered in the "Python: Further Topics" course.  
For details of this course, see:

<http://training.csx.cam.ac.uk/course/pythonfurther>

For details of the "Python: Regular Expressions" course, which covers the use of regular expressions in Python, see:

<http://training.csx.cam.ac.uk/course/pythonregexp>



71

We'll divert very briefly from looking at functions that give lists to explore the “methods” idea a bit further. As we've seen, Python lists are *bona fide* objects with their own set of methods. One of these, for example, is “`count()`” which takes an argument and reports back how many times that item occurs in the list.

# Combining methods

```
>>> 'the cat sat on the mat'.split().count('the')
```

```
2
```

First run  
**split()** to  
get a list

Second run  
**count( 'the' )**  
on that list

72

We can run the `split()` and the `count()` methods together. If we are interested in seeing how often the word “the” occurs in a string we can split it into its constituent words with `split()` and then count the number of times the word “the” occurs with `count()`. But we don’t need to create a variable to refer to the list; we can just run together the two methods as shown above.

(Note that strings also have a “`count()`” method which does something completely different. The `count()` method for strings takes an argument and reports how many times that argument occurs as a sequence of characters (a “sub-string”) in the string. For example:

```
>>> 'the cat sat on the mat'.count('at')
```

```
3
```

Because we want to find the number of times the *word* “the” occurs, we need to first split the string into a list of words and then use the `count()` method of that list. If we just used the `count()` method of the string then we would get the number of times the sub-string “the” occurred in the string, which might be more than the number of times the word “the” occurs, as below:

```
>>> 'three of the cats are over there'.count('the')
```

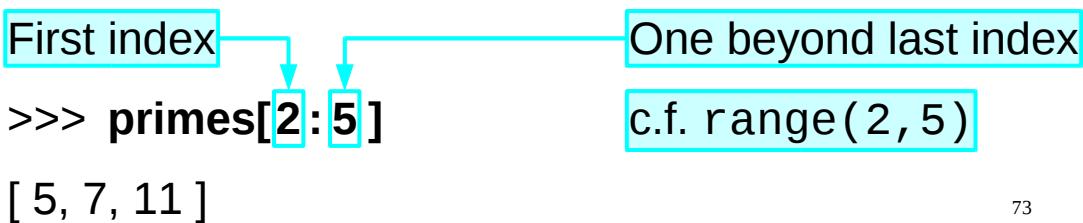
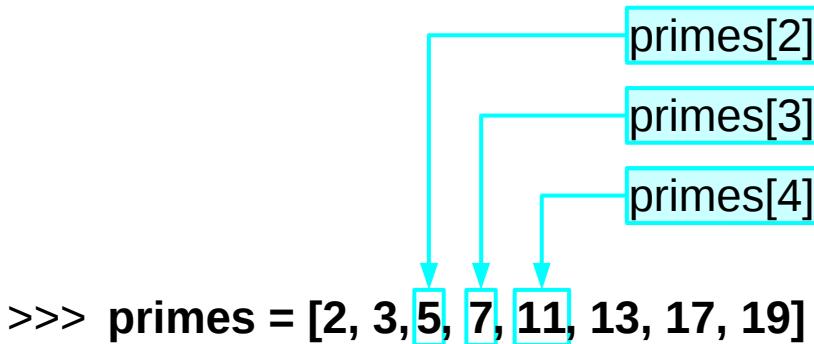
```
2
```

```
>>> 'three of the cats are over there'.split().count('the')
```

```
1
```

```
)
```

## Extracts from lists: “slices”



73

A very common requirement is to extract a sub-list from a list. Python calls these “slices” and they are the last of the things returning lists that we will consider.

The syntax for extracting a sub-list is very similar to that for extracting a single item. Instead of using a single index, though, we use a pair separated by a colon. The first index is the index of the first item in the slice. But the second index is the one *beyond* the last item in the slice. In this way it is very similar to the `range()` function we saw earlier, and, just as with `range()`, this quirk gives Python the property that slices can be concatenated in an elegant manner.

```
>>> primes[2:5] [ 5, 7, 11 ] Both limits given  
>>> primes[:5] [ 2, 3, 5, 7, 11 ] Upper limit only  
>>> primes[2:] [ 5, 7, 11, 13, 17, 19 ] Lower limit only  
>>> primes[:] [ 2, 3, 5, 7, 11, 13, 17, 19 ] Neither limit given
```

74

Slices don't actually need both indices defined. If either is missing, Python interprets the slices as starting or ending at the start or end of the original list. If both are missing then we get a copy of the whole list. (In fact, the way to copy a list in Python is to take a slice of the list with both indices missing.)

```

#!/usr/bin/python

# This is a list of some metallic
# elements.
metals = [ 'silver', 'gold', ... ]

# Make a new list that is almost
# identical to the metals list: the new
# contains the same items, in the same
# order, except that it does *NOT*
# contain the item 'copper'.

# Print the new list.

```

75

**What goes here?**

Time for some more exercises.

In your course home directories you will find an incomplete script called `metals.py`.

Complete the script so that it takes the list `metals` and produces a new list that is identical to `metals` except that the new list should not contain any '`copper`' items. Then print out the new list.

Obviously, one way of doing this would be to manually type the items from the `metals` list, except for '`copper`', into a new list. That would not be a very sensible way of doing things, and you wouldn't learn very much by doing it that way. I suggest you try something else.

If you have any questions about anything we've covered so far, now would be a good time to ask them.

```
#!/usr/bin/python

# This is a list of some data values.
data = [ 5.75, 8.25, ... ]

# Make two new lists from this list.
# The first new list should contain
# the first half of data, in the same
# order, whilst the second list should
# contain the second half, so:
#     data = first_half + second_half
# If there are an odd number of items,
# make the first new list the larger
# list.

# Print the new lists.
```

**What goes here?**

data.py

76

When you've finished the previous exercise, here's another one for you to try.

In your course home directories you will find an incomplete script called `data.py`.

Complete the script so that it takes the list `data` and produces two new lists. The first list should contain the first half of the `data` list, while the second list should contain the second half of the `data` list. (If `data` contains an odd number of items, then the first new list should be the larger list.) Then print out the new lists.

Obviously, one way of doing this would be to manually type the items from `data` into two new lists. That would not be a very sensible way of doing things, and you wouldn't learn very much by doing it that way.

If you have any questions about either this exercise or the previous one, or about anything we've covered so far, now would be a good time to ask them.

## An answer

```
#!/usr/bin/python

# This is a list of some metallic
# elements.
metals = [ 'silver', 'gold', ... ]

# Make a new list that is almost
# identical to the metals list: the new
# contains the same items, in the same
# order, except that it does *NOT*
# contain the item 'copper'.
new_metals = []
for metal in metals:
    if metal != 'copper':
        new_metals.append(metal)

# Print the new list.
print new_metals
```

metals.py

77

Here is one solution to the first exercise.

It's not the only possible solution, but it is quite a nice one because it works regardless of the number of times 'copper' appears in the `metals` list.

If, as in the example `metals` list given, the item you want to remove only appears once, you could use either of the solutions below to create the `new_metals` list.

One possible solution is to make two list slices, one on either side of the item we wish to remove ('copper'):

```
bad_index = metals.index('copper')
new_metals = metals[:bad_index] + metals[bad_index+1:]
```

Can you see why the above solution works even if 'copper' is the first or last item in the `metals` list?

Another possible solution is to make a copy of the list and then remove the unwanted item using the `remove()` method of lists:

```
new_metals = metals[:]
new_metals.remove('copper')
```

(Note that we need to make an *independent* copy (a so-called “deep copy”) of the `metals` list, so we need to make a slice of the entire `metals` list (“`metals[:]`”) rather than using “`new_metals = metals`”.)

If there is anything in any of the above solutions that you don't understand, or if your answer was wildly different to all of the above solutions, please let the course giver know now.

## Answer

```
#!/usr/bin/python

# This is a list of some data values.
data = [ 5.75, 8.25, ... ]

# Make two new lists from this list.
# The first new list should contain
# the first half of data, in the same
# order, whilst the second list should
# contain the second half, so:
#     data = first_half + second_half
# If there are an odd number of items,
# make the first new list the larger
# list.
if len(data) % 2 == 0:
    index = len(data) / 2
else:
    index = (len(data) + 1) / 2

first_half = data[:index]
second_half = data[index:]

# Print the new lists.
print first_half
print second_half
```

data.py

78

And above is a solution to the second exercise I asked you to try. It's not the only possible solution, but I've chosen this one because it is one of the most straightforward solutions.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

# Dictionaries

Key → Value  
dictionary

79

For every list – a sequence of items – there is a way we can say “item zero”, “item one” and so on. The fact that they are in a sequence implies an index number which in turn can be used to identify individual items in the list. We could regard lists as a mapping from the numbers 0, 1, ... to the items in the list:

$$\{0, 1, \dots\} \rightarrow \{\text{items in list}\}$$

We can give the list an integer and it will return the item whose index corresponds to that integer.

We can generalise this idea. Consider a list not as a sequence that implies an index, but rather as a disorganised set of items whose individual items can be identified by a number. So the “index” is no longer implied by any internal ordering but is an explicit part of the construct. We have some object that takes a number and hands back an item. In this case what was called an “index” is now called a “key”. Keys are explicit parts of the object, not numbers implied by the internal structure of the object.

Once we have made this jump then there is no reason for the keys to be numbers at all. Instead of “give me the item corresponding to the integer 2” we can ask “give me the item corresponding to the string ‘He’” (for example).

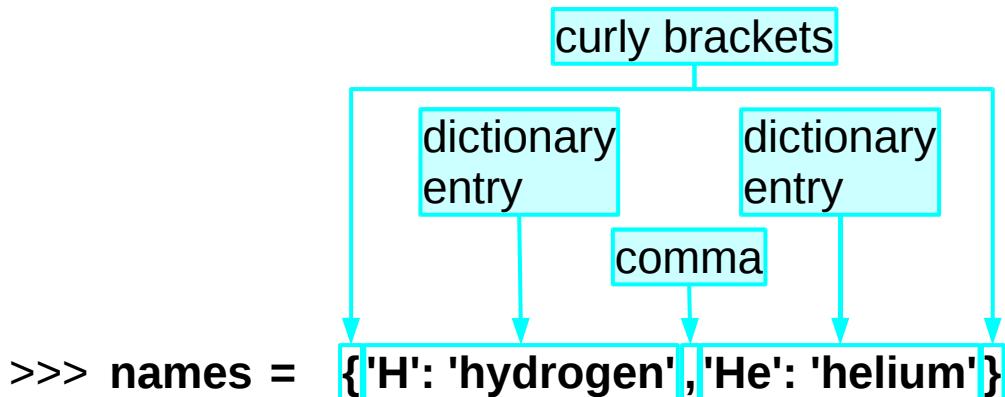
This general mapping from some type to another type is called a “dictionary” in Python. A dictionary maps a “key” (which can be almost any simple data type) to a “value” (which can be any data type):

$$\{\text{keys}\} \rightarrow \{\text{values}\}$$

These are the objects we will be considering next.

The jargon term for this sort of structure is an “associative array”, and many modern programming languages have an “associative array” type. As just mentioned, in Python, these are known as “dictionaries”. In Perl they are called “hashes”. In Java and C++ these are known as “maps”. In Visual Basic there is no standard implementation of this type that is common to all versions of Visual Basic (in Visual Basic 6.0 you can use the Scripting.Dictionary object, whilst in Visual Basic.NET you use the collection classes from the .NET Framework).

# Creating a dictionary — 1

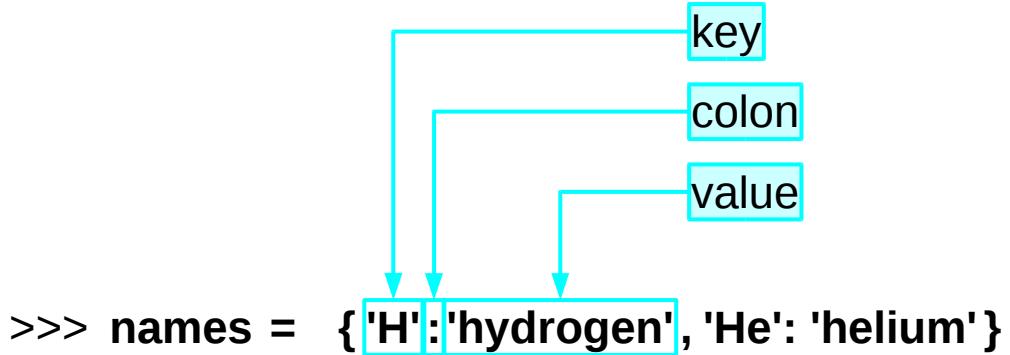


80

We can create a dictionary all in one go by listing all the key→value pairs in a manner similar to a list but with curly brackets (“braces”) around them.

The convention we’re using here is to name the dictionary after the *values* it contains (not the keys). This is just a convention and nothing like as common as the convention for lists. The authors’ preferred convention is to call the dictionary `symbol_to_name` or something like that, but long names like that don’t fit on slides.

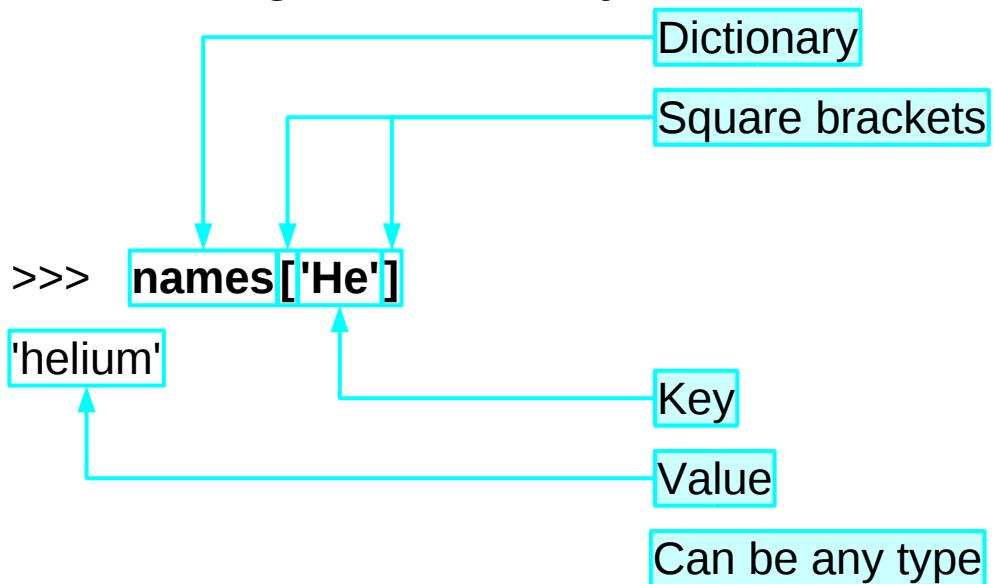
## Creating a dictionary — 2



81

The individual key → value pairs are separated by a colon.

# Accessing a dictionary



82

To get at the value corresponding to a particular key we do have the same syntax as for lists. The key follows the dictionary surrounded by square brackets.

Given this repeated similarity to lists we might wonder why we didn't use square brackets for dictionary definition in the first place. The answer is that the interpreter would then not be able to distinguish the empty list ("[ ]") from the empty dictionary ("{}").

## Creating a dictionary — 3

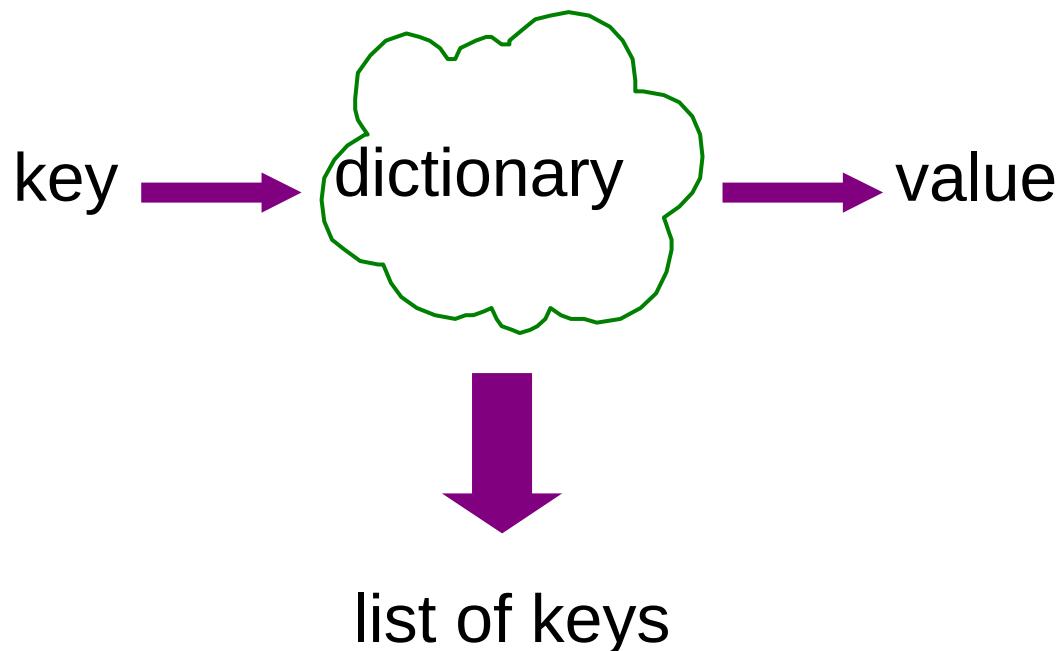
```
>>> names = {}  
>>> names[ 'H' ] = 'hydrogen'  
>>> names[ 'He' ] = 'helium'  
>>> names[ 'Li' ] = 'lithium'  
>>> names[ 'Be' ] = 'beryllium'
```

Start with an empty dictionary

Add entries

83

The other way to create a dictionary is to create an empty one and then add key → value items to it one at a time.



84

So we have a dictionary which turns keys into values. Therefore it must know all its keys. So how do we get at them?

Dictionaries can be easily persuaded to hand over the list of their keys.

# Treat a dictionary like a list...

```
Python expects a list here  
for symbol in names:  
    print symbol, names[symbol]  
    del symbol  
Dictionary key
```

...and it behaves like a list of keys

85

The basic premise is that if you plug a dictionary into any Python construction that requires a list then it behaves like a list of its keys. So we can use dictionaries in `for` loops like this.

Note that we use the variable name “`symbol`” simply because it makes sense as a variable name; it is not a syntactic keyword in the same way as “`for`” and “`in`” (or even “`:`”). The script fragment

```
for symbol in names:  
    print symbol, names[symbol]
```

is exactly the same as

```
for long_variable_name in names:  
    print long_variable_name, names[long_variable_name]
```

except that it’s shorter and easier to read!

Note that we name the variable `symbol`, which will be carrying the values of keys in the dictionary, after the keys, not the values. As the `names` dictionary has the symbols of atomic elements as *keys* and the names of atomic elements as values, we name the variable `symbol` after the *symbols* of atomic elements.

If we just want to get a list of all the keys of a dictionary, we can use the `keys()` method of the dictionary, which returns a list of the keys in the dictionary (in no particular order).

```
>>> names = {'H': 'hydrogen', 'He': 'helium'}  
>>> names.keys()  
['H', 'He']
```

## Example

```
#!/usr/bin/python

names = {
    'H': 'hydrogen',
    'He': 'helium',
    ...
    'U': 'uranium',
}

for symbol in names:
    print names[symbol]
del symbol

chemicals.py
```

```
$ python chemicals.py
ruthenium
rhenium
...
astatine
indium
```

No relation between  
order in file and output!

86

Note that there is no implied ordering in the keys. If we treat a dictionary as a list and get the list of keys then we get the keys in what looks like a random order. It is certainly not the order they were added in to the dictionary.

# Missing keys

```
>>> names['Np']
```

missing key

Traceback (most recent call last):  
File "<stdin>", line 1, in <module>

KeyError: 'Np'

Type of  
error

Missing key

87

What happens if you ask for a key that the dictionary does not have? Just as there is an error for when a list is given an out of range index, the dictionary triggers an error if it is given an unknown key.

## Treat a dictionary like a list...

Python expects a list here

```
if symbol in names:  
    print symbol, names[ symbol ]
```

...and it behaves like a list of keys

88

Ideally, we would be able to test for whether a key is in a dictionary. We can.

The Python “in” keyword can be used to test whether an item is in a list or not. This involves treating something like a list so we can get the same effect with a dictionary. The “in” keyword will test to see if an item is a *key* of the dictionary.

There is no corresponding test to see whether an item is a valid *value* in a dictionary.

However, there is a sneaky way we can do this. Dictionaries have a `values()` method, which returns a list of the values of the dictionary (in no particular order). We can then use the “in” keyword to see whether an item is in this list. If so, then it is a valid *value* in the dictionary.

```
>>> names = {'H': 'hydrogen', 'He': 'helium'}  
>>> names.values()  
['hydrogen', 'helium']  
>>> 'helium' in names.values()  
True
```

(At this point, those of you who are overly impressed with your own cleverness may think you that you can use the `keys()` and `values()` methods together to get two lists where the order of the items in the list from `keys()` is related to the order of the items in the list from `values()`, i.e. the first item in the list from `keys()` is the key in the dictionary whose value is the first item in the list from `values()`, and so on. Be aware that this is **not** always true. **So don't rely on it.**)

# Missing keys

```
>>> names['Np']
```

Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
KeyError: 'Np'

```
>>> 'Np' in names
```

Test for membership of a *list*

```
False
```

'Np' is not a key in the dictionary

89

So, as for lists, we get a simple Boolean (True or False) when we test for key membership.

```
>>> names = {'H': 'hydrogen', 'He': 'helium', 'Li':  
'lithium'}  
>>> 'Np' in names  
False  
>>> 'Li' in names  
True
```

Note that the “names =” line above is a single line and should be typed in as such – it is split across two lines here to make it easier to read.

# And now for something completely...



Obviously when you create a dictionary you need to be clear about which items are the keys and which are the values. But what if you are given a dictionary that is the “wrong way round”?

Have a look at the script `chemicals_reversed.py` in your home directory.

See if you can figure out what it does – if there is anything you don’t understand, tell the course giver or demonstrator.

90

To see dictionaries in practice, stop for a while and inspect a Python script we have written for you. This script takes a dictionary that resolves symbols of atomic elements to names of atomic elements and creates a dictionary that takes names of atomic elements and gives their symbols. Examine the script and see if you can figure out what it does (and why it does it). The script is called `chemicals_reversed.py` and is in your course home directories.

If you have any questions, please ask the course giver or a demonstrator. After you’ve finished looking at this script, take a break.

In case there was anything in the script you didn’t understand, this is what it does:

**Start:** We start with the “right way round” dictionary and we name this dictionary, which maps symbols of atomic elements to their names, `names`.

```
names = {...}
```

**Create empty dictionary:** Next we create the empty “reversed” dictionary which maps names of atomic elements to the corresponding symbols, `symbols`:

```
symbols = {}
```

**Loop:** Then we need to fill `symbols`. We know how to do a `for` loop, so we’ll do one here to fill in the reversed dictionary, `symbols`, one item at a time. We name the loop variable after the keys it steps through (hence “symbol”).

```
for symbol in names:
```

**Look up name of atomic element:** In the “indented block” that is run once for each key we will look up the key (symbol of the atomic element) in the `names` dictionary to get the name of the atomic element.

```
name = names[symbol]
```

**Add reversed entry to dictionary:** We will then assign it in reversed fashion into `symbols`.

```
symbols[name] = symbol
```

**Clean up:** Once out of the loop (and therefore unindented) we will `del` the variables we used for the loop because we’re good programmers who write clean, hygienic code.

```
del name  
del symbol
```

**Print the dictionary:** Finally, now that we have the “reversed” dictionary, we print it out.

# Defining functions

define a function

```
def reverse( a_to_b ):
```

Values in:

a\_to\_b

b\_to\_a = {}

Values out:

b\_to\_a

for a in a\_to\_b :

Internal values: a b

b = a\_to\_b[a]

Internal values  
are automatically  
cleaned up on exit.

b\_to\_a[b] = a

return b\_to\_a

91

As with most programming languages, Python allows us to define our own functions.

So let's examine what we want from a function and how it has to behave.

Our function will have a well-defined, and typically small, set of inputs which will be passed to it as its arguments.

Similarly, we want a well-defined set of outputs. Python functions are permitted a single output only, though as we will see shortly, it is standard practice to pass several values bundled together in that single output.

So we have well-defined inputs and outputs. What about variables?

Python works like this: If any variables are created or updated inside the function then they are automatically made local to that function. Any external variables of the same name are ignored. If a variable is read, but not changed then Python will first look for a local variable and, if it doesn't exist, then go looking for an external one.

How do we define a function?

To define a function we start with the keyword “**def**” to announce the **definition** of a function. This is followed by the name of the function being defined. In the example above the function is called “**reverse**”.

Then comes the specification of the input[s] required by the function. These appear just as they do when the function is used: in round brackets. If there is more than one input to the function then they are separated by commas. The whole line ends with a colon...

...and after a colon comes indentation. The body of our function will be indented in the Python script.

(Note how every single variable is local to the function either because it was passed in as a function argument (**a\_to\_b**) or because it was created inside the function (**b\_to\_a**, **a**, **b**).)

Finally, we need to get the desired value out of the function. We do this with the “**return**” statement. This specifies what value the function should return and marks the end of the function definition.

## Example

```
#!/usr/bin/python

def reverse(a_to_b):
    b_to_a = {}
    for a in a_to_b:
        b = a_to_b[a]
        b_to_a[b] = a
    return b_to_a

names = {...}

symbols = reverse(names)
...
    chemicals2.py
```

92

We've put this function in the script `chemicals2.py` in your course home directories. You can try it out and compare it to the `chemicals_reversed.py` script.

In this script we have taken the main body of the `chemicals_reversed.py` script and turned it into functions.

```
def reverse(a_to_b):
    b_to_a = {}
    for a in a_to_b:
        b = a_to_b[a]
        b_to_a[b] = a
    return b_to_a
```

function to  
reverse a  
dictionary

```
def print_dict(dict):
    for item in dict:
        print item, dict[item]
```

function to  
print a  
dictionary

```
names = {...}
symbols = reverse(names)
print_dict(symbols)
```

main body  
of script

93

Note that the function definitions are at the start of the script. It is traditional that function definitions appear at the start of a script; the only requirement is that they are defined before they are used.

Where we used to have the functionality that created `symbols` by reversing `names` we now have a single line that calls the function

```
symbols = reverse(names)
```

The printing functionality has been replaced by another single line that calls the function to print the new dictionary:

```
print_dict(symbols)
```

Please note that the line

```
symbols = reverse(names)
```

means “pass the value currently held in the external variable `names` into the function `reverse()` and take the value returned by that function and stick it in the variable `symbols`”.

## Let's try it out...

```
#!/usr/bin/python

def reverse(a_to_b):
    b_to_a = {}
    for a in a_to_b:
        b = a_to_b[a]
        b_to_a[b] = a
    return b_to_a

names = {...}

symbols = reverse(names)
...
```

chemicals2.py

```
$ python chemicals2.py
gold Au
neon Ne
cobalt Co
germanium Ge
...
tellurium Te
xenon Xe
```

94

Run the `chemicals2.py` script and see what it does. You should find it behaves exactly the same as the `chemicals_reversed.py` script.

# Re-using functions: “modules”

Two functions:

```
reverse(a_to_b)  
print_dict(dict)
```

currently in chemicals2.py

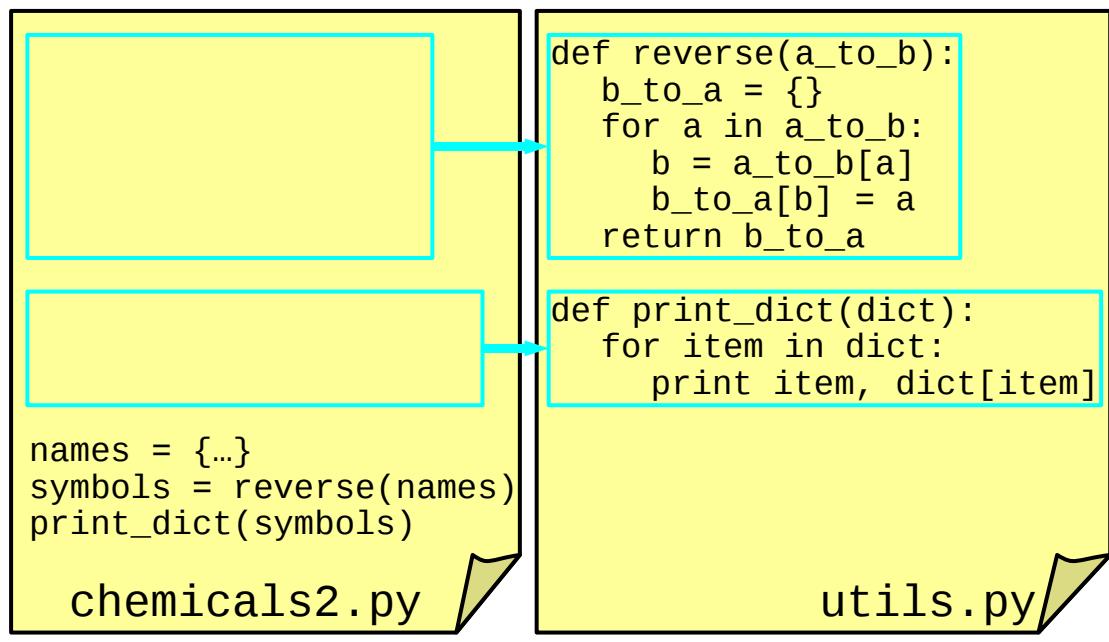
95

We have a function that can reverse any one-to-one dictionary (i.e. a dictionary where no two keys give the same corresponding value) and another function that prints out the key → value pairs of any dictionary. These are general purpose functions, so wouldn't it be nice to be able to use them generally? We can reuse a function within a script, but what about between scripts?

In Python, we re-use functions by putting them in a file called a “*module*”. We can then load the module and use the functions from any script we like. Modules are Python's equivalent of libraries in other programming languages.

# Modules — 1

Put function definitions to new file `utils.py`



So let's do precisely that: use our function in different scripts.

We start by

- (a) opening a new file called `utils.py`,
- (b) cutting the definitions of the two functions from `chemicals2.py`,
- (c) pasting them into `utils.py`, and
- (d) saving both files back to disc.

The script `chemicals2.py` no longer works as it uses functions it doesn't know the definitions for:

```
$ python chemicals2.py
Traceback (most recent call last):
  File "chemicals2.py", line 98, in <module>
    symbols = reverse(names)
NameError: name 'reverse' is not defined
```

So we need to connect `chemicals2.py` with `utils.py`.

# Modules — 2

## “import” the module

```
import utils
names = {...}
symbols = reverse(names)
print_dict(symbols)

chemicals2.py
```

```
def reverse(a_to_b):
    b_to_a = {}
    for a in a_to_b:
        b = a_to_b[a]
        b_to_a[b] = a
    return b_to_a

def print_dict(dict):
    for item in dict:
        print item, dict[item]
```

utils.py

We replace the definitions with one line, “`import utils`”, which is the instruction to read in the definitions of functions in a file called `utils.py`.

Our script still doesn’t work.

If you try you should get the same error message as before (with a different line number) but if you have mistyped the `import` line you will get this error for mistyping “`import`”:

```
$ python chemicals2.py
File "chemicals2.py", line 3
    imoprt utils
          ^
SyntaxError: invalid syntax
```

and this error for mistyping “`utils`”:

```
$ python chemicals2.py
Traceback (most recent call last):
  File "chemicals2.py", line 3, in <module>
    import uitls
ImportError: No module named uitls
```

Note the use of the word “`module`”. We will see that again soon.

You may wonder how Python knows where to find the `utils.py` file. We’ll return to this later. For now you just need to know that, unless someone has configured Python to behave differently, Python will search for any file you ask it to `import` in the current directory (or the directory containing the script you are running when you are running a Python script), and, if it can’t find the file there, it will then try some system directories.

# Modules — 3

Use functions from the module

```
import utils

names = {...}
symbols = utils.reverse(names)
utils.print_dict(symbols)
```

chemicals2.py

```
def reverse(a_to_b):
...
def print_dict(dict):
...
utils.py
```

The problem is that Python is still looking for the function definitions in the same file (`chemicals2.py`) as it is calling them from. We need to tell it to use the functions from the `utils.py` file. We do that by prefixing “`utils.`” in front of every function call that needs to be diverted.

Now the script works:

```
$ python chemicals2.py
gold Au
neon Ne
cobalt Co
germanium Ge
...
manganese Mn
tellurium Te
xenon Xe
```

and we have a `utils.py` file that we can call from other scripts as well.

## Let's check it still works...

```
#!/usr/bin/python

import utils

names = {...}

symbols = utils.reverse(names)
utils.print_dict(symbols)
```

chemicals2.py

\$ **python chemicals2.py**

```
gold Au
neon Ne
cobalt Co
germanium Ge
...
tellurium Te
xenon Xe
```

99

After you've moved the functions into the `utils.py` file and adapted the `chemicals2.py` script appropriately, you should run your modified `chemicals2.py` script and check that it still works.

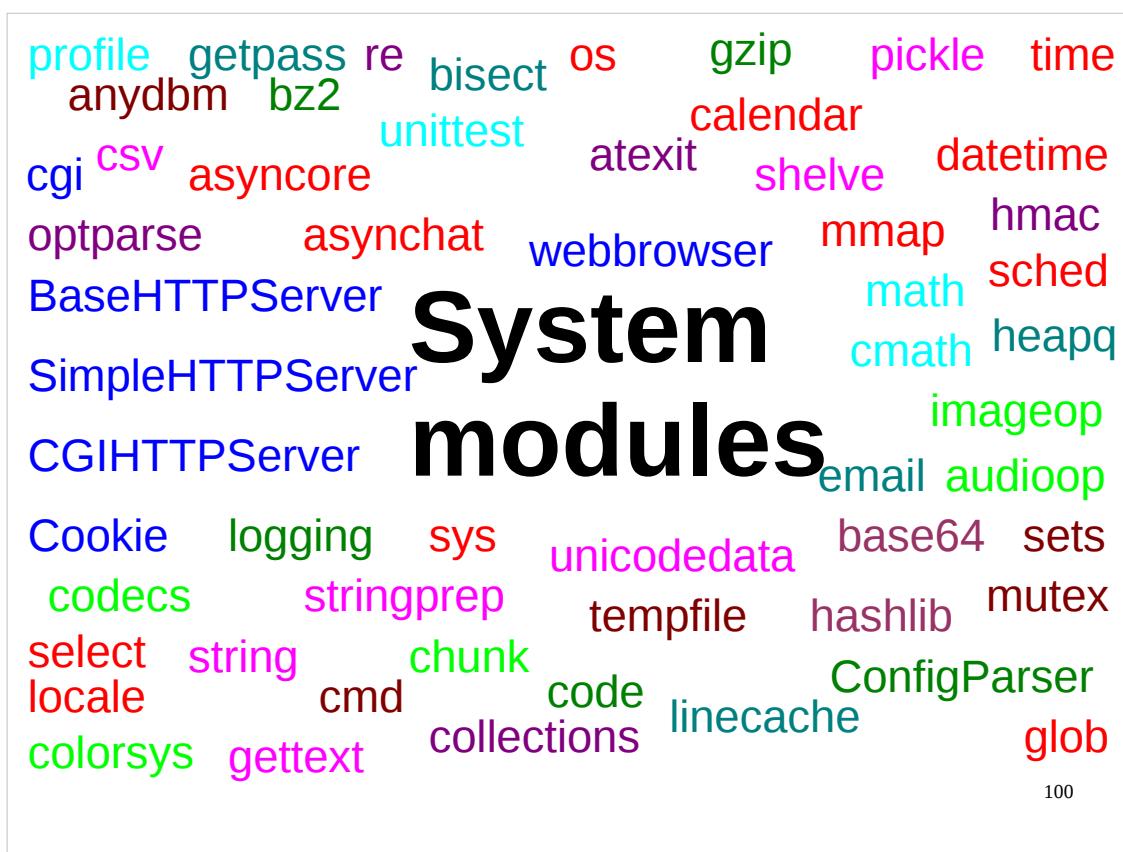
The collection of functions provided by the file `utils.py` is known as the “`utils module`” and the operation performed by the line “`import utils`” is referred to as “importing a module”.

When you ask Python to import a module, it first checks to see whether the module is one of the built-in modules that are part of Python. If not, it then searches in a list of directories and loads the **first** matching file it finds. Unless someone has configured Python differently, this list of directories consists of the current directory (or the directory in which your script lives when you are running a script) and some system directories. This list of directories is kept in the `sys` module in a variable called `path`. You can see this list by importing the `sys` module and then examining `sys.path`:

```
>>> import sys
>>> sys.path
['', '/usr/lib/python26.zip', '/usr/lib/python2.6', '/usr/lib/python2.6/plat-
linux2', '/usr/lib/python2.6/lib-tk', '/usr/lib/python2.6/lib-old',
'/usr/lib/python2.6/lib-dynload', '/usr/lib/python2.6/site-packages',
'/usr/lib/python2.6/site-packages/Numeric', '/usr/lib/python2.6/site-
packages/PIL', '/usr/local/lib/python2.6/site-packages',
'/usr/lib/python2.6/site-packages/gtk-2.0', '/usr/lib/python2.6/site-
packages/wx-2.8-gtk2-unicode']
```

(Note that in this context, the empty string, ‘`''`’, means “look in the current directory”.) As `sys.path` is a Python list, you can manipulate it as you would any other Python list. This allows you to change the directories in which Python will look for modules (and/or the order in which those directories are searched).

You can also affect this list of directories by setting the `PYTHONPATH` environment variable before you start the Python interpreter or run your Python script. If the `PYTHONPATH` environment variable is set, Python will add the directories specified in this environment variable to the *start* of `sys.path`, *immediately after* the ‘`''`’ item, i.e. it will search the current directory (or the directory containing the script when you are running a script), then the directories specified in `PYTHONPATH` and then the system directories it normally searches.



There are, of course, very many modules provided by the system to provide collections of functions for particular purposes.

Almost without exception, every course that explains how to do some particular thing in Python starts by importing the module that does it. Modules are where Python stores its big guns for tackling problems. Appended to the notes is a list of the most commonly useful Python modules and what they do.

On any given installation of Python you can find the names of all the modules that are available using the `help()` function:

```
>>> help('modules')
```

Please wait a moment while I gather a list of all available modules...

Alacarte	_LWPCookieJar	gconf	pycompile
ArgImagePlugin	_MozillaCookieJar	gdbm	pyclbr
...			
XbmImagePlugin	functools	pty	zope
XpmImagePlugin	gc	pwd	

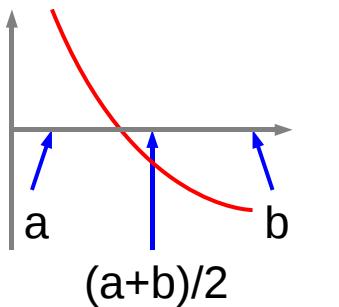
Enter any module name to get more help. Or, type "modules spam" to search for modules whose descriptions contain the word "spam".

>>>

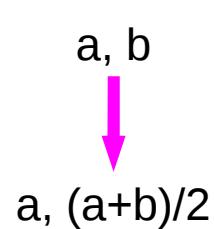
The complete list of the modules that ship with Python can be found in the Python documentation at the following URL:

<http://docs.python.org/modindex.html>

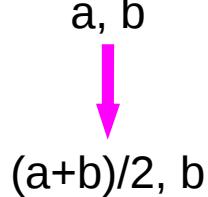
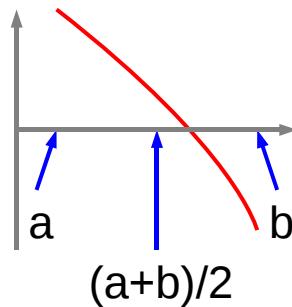
# Root-finding by bisection



$y=f(x)$



until  $b-a<\delta$



101

Next we will look at getting more complex information into and out of our functions. Our functions to date have taken a single input argument and have returned either no output or a single output value. What happens if we want to read in or to return several values?

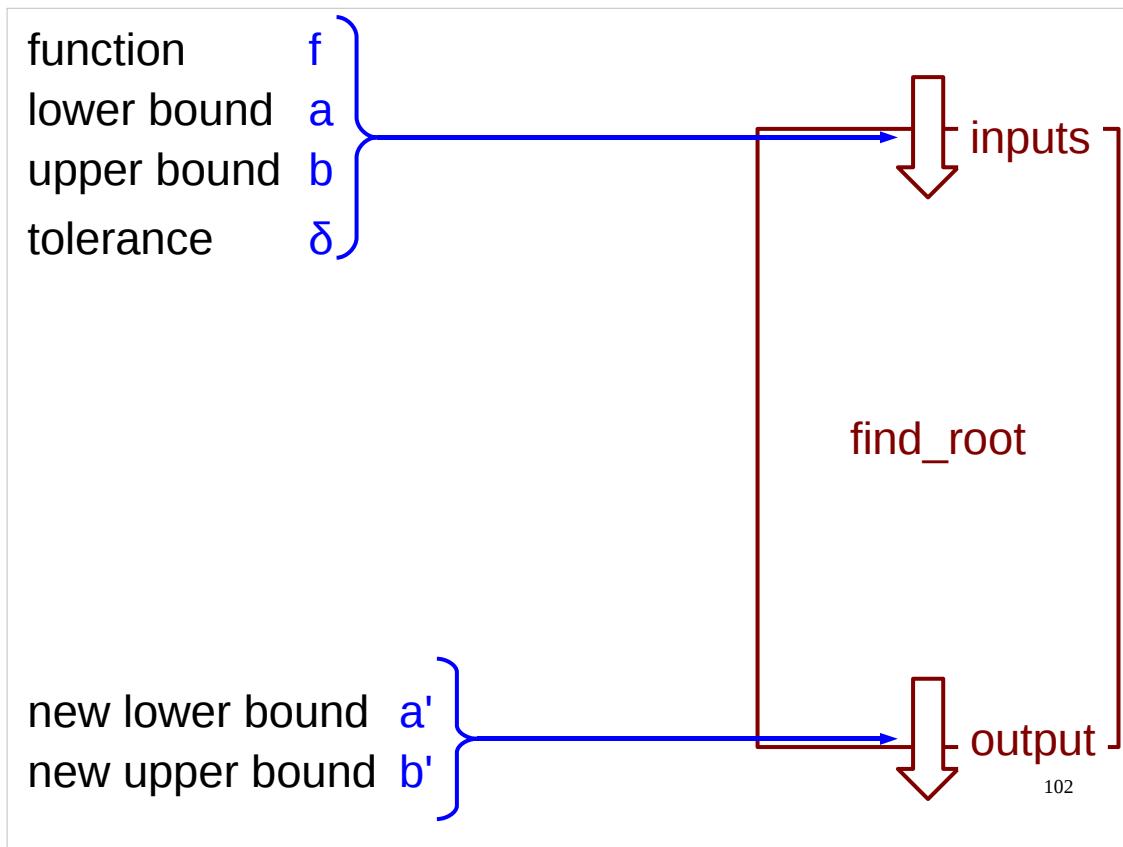
I'm picking a simple numerical technique as my example.

We want to find numerical approximations to the roots of (mathematical) functions. We know the root is in a particular interval if the function is positive at one end of the interval and negative at the other end. This means that the function is zero somewhere in between the ends of the interval. (This is the “intermediate value theorem” for the maths nargs reading.)

The trick is to cut the interval in two and calculate the function at the mid point. If it's positive then the root is between the midpoint and the end that has a negative value. If it's negative the root is between the midpoint and the end with the positive value.

There's a neat trick we can apply doing this. If the root lies between  $(a+b)/2$  and  $b$  then the function values at these two points have different signs; one is positive and one is negative. This means that their product is negative. If the root lies between  $a$  and  $(a+b)/2$  then the function is either positive at both  $(a+b)/2$  and  $b$  or negative at both these points. Either way, the product of the values at the two points is positive. So we can use the sign of  $f(b) \times f((a+b)/2)$  as a test for which side of  $(a+b)/2$  the root is on.

We repeat this bisection until we have an interval that's close enough for us.



Our Python function will need four inputs, then:

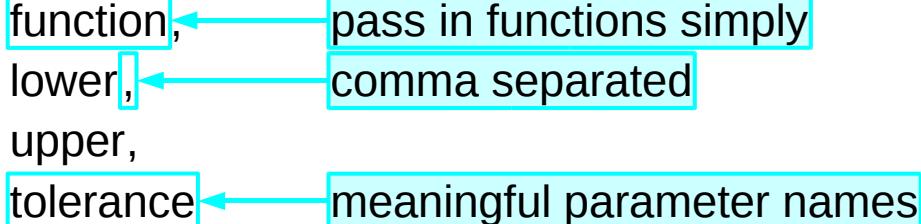
- the mathematical function whose root we are looking for,
- the two ends of the interval, and
- some measure of how small the interval has to be to satisfy us.

And it will return two outputs:

- the two ends of the new interval.

# Multiple values for input

```
def find_root(  
    function,  
    lower,  
    upper,  
    tolerance  
):  
    #function body
```



103

So how do we code this in Python? Changing to multiple inputs is easy; we simply list the inputs separated by commas. When we call the function we will pass in multiple values separated by commas in the same order as they appear in the definition.

Note that you can spread the arguments over multiple lines, using the standard Python indenting style. This lets you pick sensible variable names.

Note also that you can pass a function as input to another function just by giving its name. You don't have to use a special syntax to pass a function as input, nor do you need to mess around with pointers and other such strange beasts as you do in some other programming languages.

# Multiple values for output

```
def find_root(  
    ...  
):  
    function body  
  
    return (  
        lower ,  
        upper  
    )
```

A curly brace on the right side of the code block groups the 'lower' and 'upper' variables, with the text 'typically on a single line' enclosed in a red box below it.

104

So that was how to handle multiple inputs. What about multiple outputs?

The answer is simple. Exactly as we did for inputs we return a set of values in round brackets and separated by commas. As with the inputs, you can spread the arguments over multiple lines, using the standard Python indenting style.

So, instead of returning a single value we can return a pair.

```

def find_root(
    function,
    lower,
    upper,
    tolerance
):

    while upper - lower > tolerance:
        middle = (lower + upper) / 2.0
        if function(middle)*function(upper) > 0.0:
            upper = middle
        else:
            lower = middle

    return (lower, upper)

```

utils.py

105

So here is our completed function in our `utils.py` file. You should modify your copy of `utils.py` so that it has this function definition in it as it will be needed for the next example.

Don't get hung up on the details of the function, but notice that if it is using floating point numbers it should use them consistently and uniformly throughout.

Recall why we multiply the two function values together and check whether its positive or not:

If the root lies between  $(a+b)/2$  and  $b$  then the function values at these two points have different signs; one is positive and one is negative. This means that their product is negative. If the root lies between  $a$  and  $(a+b)/2$  then the function is either positive at both  $(a+b)/2$  and  $b$  or negative at both these points. Either way, the product of the values at the two points is positive. So we can use the sign of  $f(b) \times f((a+b)/2)$  as a test for which side of  $(a+b)/2$  the root is on.

```
#!/usr/bin/python

import utils

def poly(x):
    return x**2 - 2.0

print utils.find_root(poly, 0.0, 2.0, 1.0e-5)
```

Find the root  
of this function

sqrt2.py

\$ **python sqrt2.py**

(1.4142074584960938, 1.414215087890625)

106

We wrote our root finder in `utils.py`, so we can quickly write a script to exploit it, and there is such a script in the file `sqrt2.py` in your course home directories. Note how we write a simple function in Python and pass it in as the first argument.

Recall that `x**2` means  $x$  squared ( $x^2$ ).

Recall that our `find_root()` function returns a pair of values. So, what does a pair of values look like? Well, we can print it out and we get, printed, the two values inside parentheses and separated by a comma. Simple, really!

```

#!/usr/bin/python

import utils

def poly(x):
    return x**2 - 2.0

(lo, up) = utils.find_root(poly, 0.0, 2.0, 1.0e-5)

print lo
print up

```

sqrt2.py

**\$ python sqrt2.py**

1.4142074585  
1.41421508789

107

But how can we fiddle with pairs of values? How can we get at just one of the values, for example?

The easiest way is to assign a pair of variables to the pair of values in a single operation, as shown. This is analogous to how we assigned a list of variables to a list of values; we can assign a pair of variables to a pair of values.

(Note that when we `print` out the values separately, as here, `print` “prettifies” them, whilst when we `print` the pair of values as we did before, `print` leaves the individual values in the pair alone and displays them “as is”.)



## Let's break for an exercise...

Write a function that takes a list of numbers as input, and returns the following:

- smallest number in list
- arithmetic mean of list
- largest number in list

If you run into problems with this exercise, ask the course giver or a demonstrator for help.

108

So let's break for another exercise.

Write a function that takes a list of numbers as its input and returns the smallest number in the list, the arithmetic mean (also known as the *average*) of the numbers in the list, and the largest number in the list. You may assume that the list has at least one number in it.

You should test your function after you've written it to make sure it works properly.

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

Also, if you have any questions on anything we've done so far, now would be a good time to ask them.

This page intentionally left blank

~~SILLY~~



109

The exercise just set is fairly straightforward, but if, and only if, you are not getting anywhere with it, then turn the page and have a look at the answer. Then tell the course giver where you were having difficulty with the exercise.

...and regardless of how easy or difficult you are finding the exercise, pause for a moment and reflect on the genius that is Monty Python's Flying Circus.

## Answer

```
def stats(numbers):  
  
    min = numbers[0]  
    max = numbers[0]  
    total = 0  
  
    for number in numbers:  
        if number < min:  
            min = number  
        if number > max:  
            max = number  
        total = total + number  
  
    return (min,  
           total/(len(numbers)+0.0),  
           max)
```

n.b. Function *fails* if the list is empty.

utils.py

110

Here's my answer to the exercise set over the break. Note that the function fails if the list is empty. (I've called my function `stats()`, but you can of course call your function whatever you want.)

Note also that I initially set `total` to `0` rather than `0.0`. This is because I didn't specify whether the numbers in the list that the function takes as input were integers or floating point numbers (or complex numbers for that matter). If I set `total` to the *integer* `0`, then as soon as I add one of the numbers from the list to it, it will be converted (coerced) to the correct type. If, however, I were to set `total` to `0.0` and my function was given a list of integers, then all those integers would be turned into floating point numbers before being added to `total`, potentially unnecessarily losing precision.

Finally, note that when I do the division required to calculate the arithmetic mean (or average) from `total`, I force one of the numbers to be a floating point number (by adding `0.0` to it) to ensure that we don't inadvertently do integer division.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

Singles  
Doubles  
Triples  
Quadruples  
Quintets

# Tuples

( 42 , 1.95 , 'Bob' )

( -1 , +1 )

( 'Intro. to Python', 25, 'TTR1' )

“not the same as lists”

111

So what are these strange bracketed collections of values we've been using for returning multiple outputs from our functions?

These collections of values (pairs, triplets, quads etc.) are collectively called “tuples” and are the next data type we will look at.

**Note** that technically tuples don't have to be surrounded by parentheses (round brackets), although they almost always are. A list of comma separated items not surrounded by *any* brackets is also a tuple. For instance, both of the collections of values below are tuples, although only one is explicitly surrounded by round brackets:

```
>>> ('Bob', 1.95, 42)
('Bob', 1.95, 42)
>>> 'Bob', 1.95, 42
('Bob', 1.95, 42)
```

You'll note that when Python evaluates the second tuple it displays the round brackets for us.

# Tuples are not the same as lists

(minimum, maximum)

(age, name, height)

(age, height, name)

(age, height, name, weight)

Independent,  
grouped items

Related,  
sequential  
items

[ 2, 3, 5, 7 ]

[ 2, 3, 5, 7, 11 ]

[ 2, 3, 5, 7, 11, 13 ]

112

The first question has to be “what is the difference between a tuple and a list?”

The key difference is that a tuple has a fixed number of items in it, defined by the script and that number can never change. A list, on the other hand, is a flexible object which can grow or shrink as the script develops. Our root finding function, for example, takes four values as its arguments, not an open-ended list of them. It returns precisely two values as a pair.

If you just want to bundle up a fixed collection of values to pass around the program as a single object then you should use a tuple.

For example, you might want to talk about the minimum and maximum possible values of a range. That’s a pair (minimum, maximum), not a list. After all, what would the third item be in such a list?

If you are dealing with statistics of people you might want to bundle name, height in metres and age in years. That’s a triplet (3-tuple) of a string, floating point number and integer. You might add a fourth component later (weight in kilograms) but it doesn’t follow as the result of a sequence. There is no ordering in these components. Your program would make as much sense if it worked with (age, name, height) as if it worked with (height, age, name).

A set of primes would not be a good tuple. There is an obvious ordering. The list [2, 3, 5, 7, 11] makes more sense than the list [3, 7, 2, 11, 5]. There is also an obvious “next item”: 13.

There is no constraint on the types of the components of a tuple. Because it is fixed and there is no concept of “next component” the types can be safely mixed. A triplet could have one integer, one floating point number and one string (age in years, height in metres, and name say) with no difficulties.

# Access to components

Same access syntax as for lists:

```
>>> ('Bob', 42, 1.95)[0]  
'Bob'
```

But tuples are *immutable*:

```
>>> ('Bob', 42, 1.95)[1] = 43
```

Traceback (most recent call last):

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support  
item assignment
```

113

Another way to consider a tuple is that you usually access all the values at once. With a list you typically step through them sequentially.

It is possible to access individual elements of a tuple and the syntax is unfortunately very similar to that for lists. It is almost always a bad idea to access individual elements like this; it defeats the purpose behind a tuple.

```
>>> person = ('Bob', 1.95, 42)  
>>> person  
( 'Bob', 1.95, 42)  
>>> name = person[0]  
>>> name  
'Bob'
```

This is a bad idea. You are much better off doing this:

```
>>> (name, height, age) = person  
>>> name  
'Bob'
```

If you really don't want `age` and `height`, you can always `del` them.

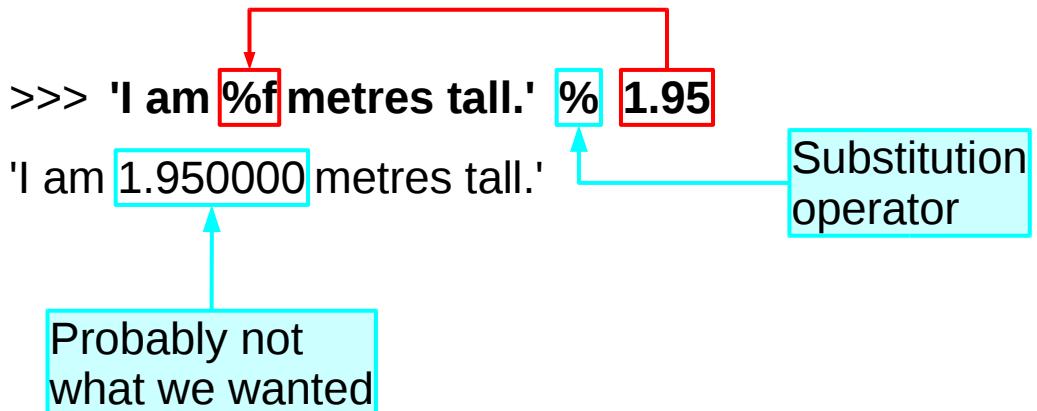
The third difference is that items in a list can be changed; components of a tuple can't be. Just as the components of a tuple are accessed all at once, they can only be changed all at once, basically by replacing the whole tuple with a second one with updated values.

If we try to change the value of a single component:

```
>>> person = ('Bob', 1.95, 42)  
>>> person[2]  
42  
>>> person[2] = 43  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

This is another reason not to dabble with direct access to individual components of tuples.

## String substitution



114

We started with tuples as a way to get multiple values in and out of functions.

Then we looked at them as items in their own right, as means to collect values together. (You can see another example of this use of tuples in the file `chemicals3.py` in your course home directories which we will use in an exercise in a little while.)

The third use we can make of tuples involves “string substitution” (also known as “string formatting”). This is a mechanism for a (typically long) string to have values (from a tuple) substituted into it.

Python uses the “%” operator to combine a string with a value or a tuple of values.

On the slide above, inside the string a “%f” marks the point where a floating point number needs to be inserted.

Note that the formatting might not be all you wanted.

# Substituting multiple values

```
>>> 'I am %f metres tall and my name is %s.'  
% (1.95, 'Bob')  
'I am 1.950000 metres tall and my name is Bob.'
```

The diagram illustrates the substitution process. It shows the original string 'I am %f metres tall and my name is %s.' with two red boxes: one around the placeholder '%f' and another around the value '1.95'. Below it, a tuple '% (1.95, 'Bob')' is shown with green boxes around the placeholder '%s' and the value 'Bob'. Two arrows point from the red boxes to the corresponding red boxes in the tuple. Another arrow points from the green box in the tuple to the green box in the string.

115

Note that if you only want to substitute a single value into a string you don't need to use a tuple (although you can if you wish). However, to substitute multiple values you need to use a tuple. The number of markers in the string must match the number of items in the tuple exactly.

There are a number of these substitution markers:

%d integer  
%f floating point number  
%s string

The complete list of these markers can be found in the Python documentation at the following URL:

<http://docs.python.org/library/stdtypes.html#string-formatting-operations>

(For those of you familiar with C/C++, the substitution markers used by Python for formatting are very similar to the format specifiers used by C's `sprintf()` function. And if you're not familiar with C or C++, then just ignore that remark.)

# Formatted substitution

```
>>> '%f' % 0.23  
'0.230000'  
  
>>> '%.3f' % 0.23  
'0.230'
```

The diagram illustrates two examples of formatted substitution. In the first example, a blue arrow points from the label "standard float marker" to the "%f" placeholder in the code. Another blue arrow points from the label "six decimal places" to the six zeros in the output string. In the second example, a blue arrow points from the label "modified float marker: '.3'" to the ".3" placeholder in the code. A blue arrow also points from the label "three decimal places" to the three digits in the output string.

116

We can go further. It is possible to specify formatting in the marker to specify exactly how a float, for example, should be rendered. If we interpose a “.3” between the “%” and the “f” then we get three decimal places shown. (The default for the %f marker is to display the floating point number to six decimal places.)

Note that you can use the substitution markers to convert your values into different types as the value is substituted into the string. For example, using %d with a floating point number would cause the floating point number to be substituted into the string as an integer, with the corresponding loss of precision:

```
>>> '%d' % 1.98  
'1'
```

## More complex formatting possible

```
'23'    '23.4567'    ' 23.46'  
' 23'  '23.456700' '23.46 '  
'0023' '23.46'      '+23.46'  
' +23' ' +23.4567' '+23.46 '  
'+023' '+23.456700'  
'23  ' '+23.46'     'Bob'  
'+23  ' '0023.46'   'Bob '  
     '+023.46'     '  Bob'
```

117

There is a very wide variety of formatting. A guide to it is appended to the notes but we're not going to tediously plough through it all here.

# Uses of tuples

1. Functions
2. Related data
3. String substitution

118

So, as we have seen, there are three main uses for tuples:

- **Functions:** for holding the input and output of functions;
- **Related data:** for grouping together related data that doesn't form a natural sequence and/or is of differing types; and
- **String substitution:** for holding the values that will be substituted into the string.

They are not to be confused with lists, which although similar, are not the same. Lists are to be used for sequential data, where that data is all of the same type.

```
#!/usr/bin/python

# The keys of this dictionary are the
# symbols for the atomic elements.
# The values are tuples:
# (name, atomic number, boiling point).
chemicals = {...}

# For each key in the chemicals
# dictionary, print the name and
# boiling point (to 1 decimal place),
# e.g. for the key 'H', print:
#     hydrogen: 20.3K
```

What goes here?

chemicals3.py

119

Time for another exercise.

In your course home directories you will find an incomplete script called `chemicals3.py`.

Complete this script so that for each key in the `chemicals` dictionary, the script prints out the name of the atomic element and its boiling point in Kelvin, formatted as shown on the slide above.

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

Also, if you have any questions on anything we've done so far, now would be a good time to ask them.

(By the way, the authors know that the dictionary really contains data about the *atomic elements* rather than chemicals, and so it would be better to call this dictionary “elements” rather than “chemicals”. However, people talk of “elements of lists” or “elements of dictionaries” to refer to the individual items in lists or dictionaries and we would rather not cause unnecessary confusion.)

# This page intentionally left blank

Deze bladzijde werd met opzet blanco gelaten.

このページは計画的にブランクを残している

Ta strona jest celowo pusta.

Esta página ha sido expresamente dejada en blanco.

Эта страница нарочно оставлена пустой.

Denne side med vilje efterladt tom.

Pañon intence vaka.

این صفحه خالی است

An leathanach seo fágtha folamh in aon turas.

120

This page intentionally left blank: nothing to see here. If you're stuck for an answer to the exercise, have a look at the next page.

## Answer

```
#!/usr/bin/python

# The keys of this dictionary are the
# symbols for the atomic elements.
# The values are tuples:
# (name, atomic number, boiling point).
chemicals = {...}

# For each key in the chemicals
# dictionary, print the name and
# boiling point (to 1 decimal place),
# e.g. for the key 'H', print:
#     hydrogen: 20.3K
for symbol in chemicals:
    (name, number, boil) = chemicals[symbol]
    print "%s: %.1fK" % (name, boil)
del name, number, boil
del symbol
```

chemicals3.py

121

And here's one possible solution to the exercise you were just set. Note how I access all the items in the tuple at once, and then only use the ones I actually want.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

(As already mentioned, the authors know that the dictionary in this example really contains data about the *atomic elements* rather than chemicals, and so it would be better to call this dictionary “elements” rather than “chemicals”. However, people talk of “elements of lists” or “elements of dictionaries” to refer to the individual items in lists or dictionaries and we would rather not cause unnecessary confusion.)

# Accessing the system

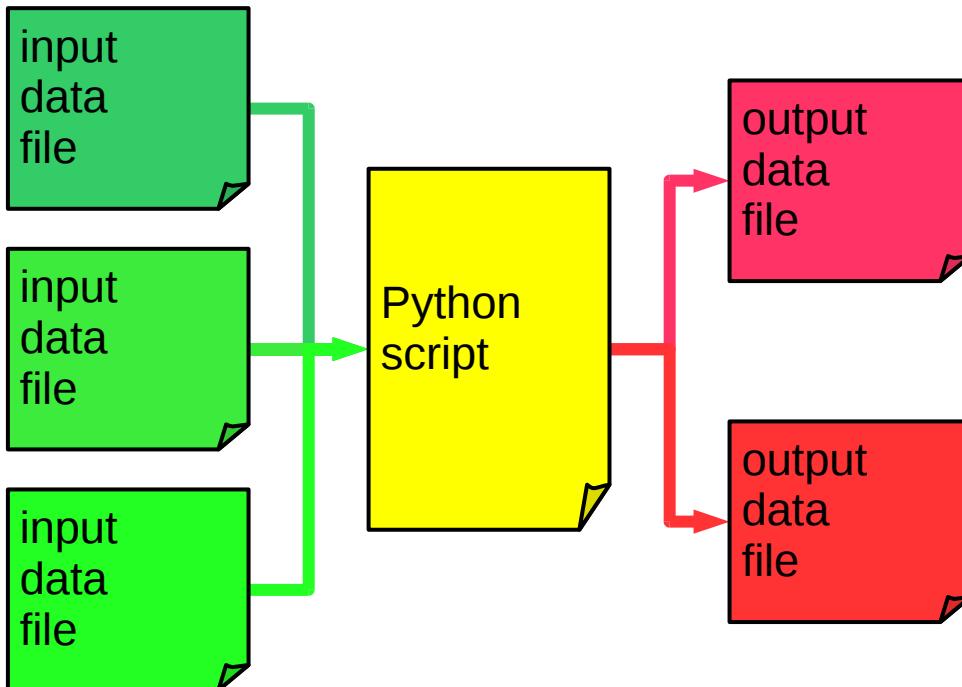
1. Files
2. Standard input & output
3. The command line

122

Next we're going to look now at three aspects of how a Python script can access the system.

First we will consider access to files and then move on to how files redirected into and out of the script work. Finally we will examine access to the command line itself.

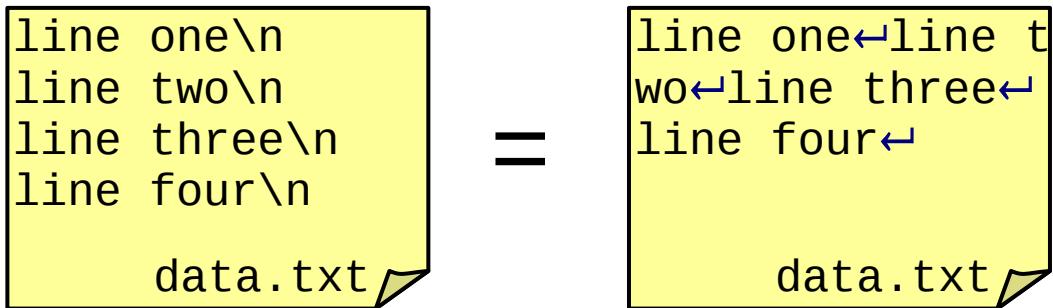
## May want to access many files



123

The usual Unix approach to input and output is to redirect files into and out of the script using the shell's redirection operations. We will move on to this model shortly but first we want to handle files directly.

In particular, we want to be able to cope with the situation where there is more than one input and/or output file, a situation which *can* be dealt with using shell redirection but which stretches its flexibility.



\n



124

There is a very important point to note about reading input from a file in Python. The input passed to Python consists of *strings*, regardless of any intended meaning of the data in the file. If we supply Python with a file that we know contains numerical values, Python doesn't care. It reads in a series of strings which just so happen to only use a few of the usual characters (digits, decimal points, minus signs and new lines). Python can't be told "this is a file of numbers"; it only reads in strings.

In keeping with its string fixation for file input, Python expects all the files it reads to consist of *lines*. As far as Python is concerned, a line is a string that ends with an "end of line" (EOL) marker (usually the "new line" character, conventionally represented as "\n"). Python can cope with operating systems that use different EOL markers, but we won't cover that in this course. (This is covered in the "Python: Further Topics" course, details of which are available at:

<http://training.csx.cam.ac.uk/course/pythonfurther>

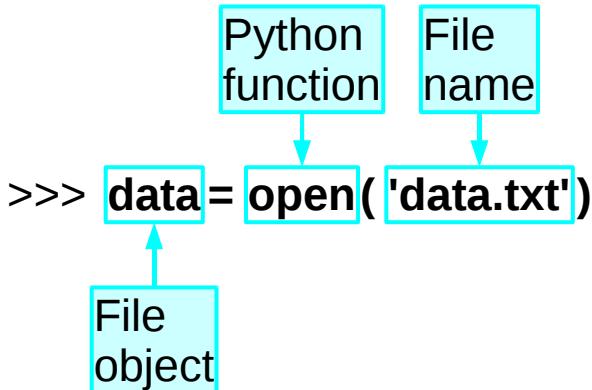
)

When Python reads a line from a file it will return the string it has read, complete with the "new line" character at the end. If we don't want this trailing "new line" character at the end of our string, we need to get rid of it ourselves. We'll see how to do this shortly.

When you create a file in a text editor, you move to a new line by just pressing the return or enter key on your keyboard. When the file is written to disk these "line breaks" are stored as the appropriate EOL marker for your operating system (under Unix, this is the "new line" character, "\n").

In your course home directories there is a file called "data.txt" which we'll be working with as we investigate how Python reads files. The contents of this file are shown on the slide above.

124



All access to the file is via the file object

125

Opening a file involves taking the file name and getting some Python object whose internals need not concern us; it's another Python type, called "file" logically enough. If there is no file of the name given, or if we don't have permission to get at this file, then the opening operation fails. If it succeeds we get a `file` object that has two properties of interest to us. It knows the file on disc that it refers to, obviously. But it also knows how far into the file we have read so far. This property, known as the "offset", obviously starts at the beginning of the file when it is first opened. As we start to read from the file the offset will change.

We open a file in Python with the "open" command.

In your course home directories there is a file called "data.txt". If you enter Python interactively and give the command

```
>>> data = open('data.txt')
```

then you should get a `file` object corresponding to this file inside Python.

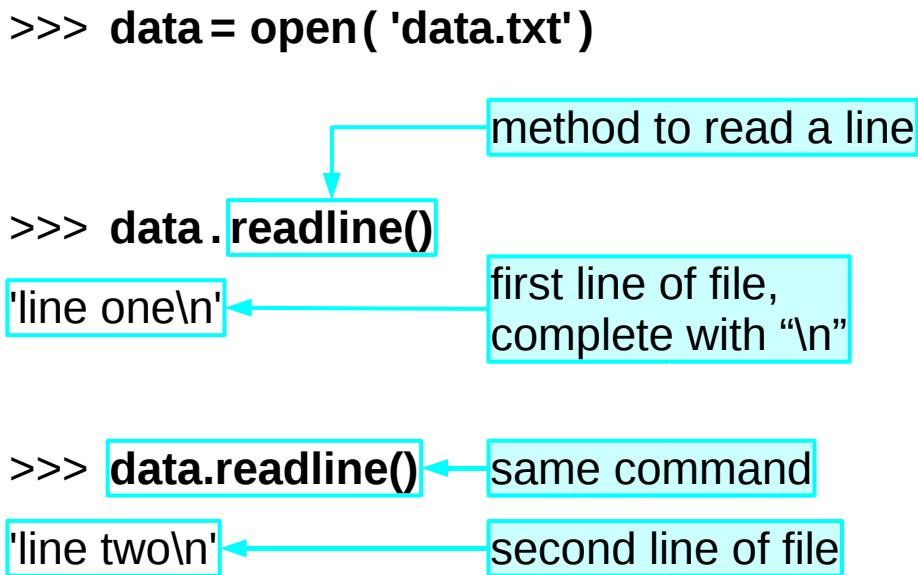
Note that we just gave the name of the file, we didn't say where it was. If we don't give a path to the file then Python will look in the current directory. If we want to open a file in some other directory then we need to give the path as well as the name of the file to the `open` command. For instance, if we wanted to open a file called "data.txt" in the /tmp directory, we would use the `open` command like this: `open('/tmp/data.txt')`.

If you want to know which directory is your current directory, you can use a function called `getcwd()` ("get current working directory") that lives in the `os` module:

```
>>> import os
>>> os.getcwd()
'/home/x241'
```

(If you try this on the computer in front of you, you will find that it displays a different directory to the one shown in these notes.)

You can change your current directory by using the `chdir()` ("change directory") function, which also lives in the `os` module. You give the `chdir()` function a single argument: the name of the directory you want to change to, e.g. to change to the /tmp directory you would use `os.chdir('/tmp')`. However, **don't try this now**, as if you change the current directory to something other than your course home directory then many of the examples in this section of these notes will no longer work! (If you have foolishly ignored my warning and changed directory, and don't remember what your course home directory was called (and so can't change back to it), the easiest thing to do is to quit the Python interpreter and then restart it.)



126

To read a file line by line (which is typical for text files), the `file` object provides a method to read a single line. (Recall that methods are the “built in” functions that objects can have.) The method is called “`readline()`” and the `readline()` method on the `data` object is run by asking for “`data.readline()`” with the object name and method name separated by a dot.

There are two important things to notice about the string returned. The first is that it’s precisely that: one line, and the first line of the file at that. The second point is that, as previously mentioned, it comes with the trailing “new line” character, shown by Python as “`\n`”.

Now observe what happens if we run exactly the same command again. (Python on PWF Linux has a history system. You can just press the up arrow once to get the previous command back again.) This time we get a different string back. It’s the second line.

```
>>> data = open( 'data.txt' )
>>> data.readline()
'line one\n'
>>> data.readline()
'line two\n'

>>> data.readlines()
[ 'line three\n', 'line four\n' ]
```

127

There's one other method which is occasionally useful. The "readlines()" method gives all the lines from the current position to the end of the file as a list of strings.

We won't use `readlines()` much as there is a better way to step through the lines of a file, which we will meet shortly.

Once we have read to the end of the file the position marker points to the end of the file and no more reading is possible (without moving the pointer, which we're not going to discuss in this course).

```
>>> data = open( 'data.txt' )
>>> data.readline()
'line one\n'
>>> data.readline()
'line two\n'
>>> data.readlines()
[ 'line three\n', 'line four\n' ]
```

```
>>> data.close() ← disconnect
>>> del data ← delete the variable
```

128

The method to close a file is, naturally, “close( )”.

It’s only at this point that we declare to the underlying operating system (Linux in this case) that we are finished with the file. On operating systems that lock down files while someone is reading them, it is only at this point that someone else can access the file.

Closing files when we are done with them is important, and even more so when we come to examine writing to them.

We should practice good Python variable hygiene and delete the `data` variable if we aren’t going to use it again immediately.

## Treating file objects like lists:

```
for line in data.readlines():
    do stuff
```

reads the lines  
all at once



```
for line in data:
    do stuff
```

reads the lines  
as needed

129

We have seen before that some Python objects have the property that if you treat them like a list they act like a particular list. `file` objects act like the list of lines in this case, but be warned that as you run through the lines you are running the offset position forward.

# Very primitive input

`line.split()`

Very simplistic splitting  
No way to quote strings

Comma separated values:      **csv** module  
Regular expressions:      **re** module

**“Python: Further Topics” course**

**“Python: Regular Expressions” course**

130

We often use the `split()` method of strings to process the line we've just read in from a file. Note, though, that the `split()` method is very, very primitive. There are many better approaches, and some of these are covered in the “Python: Further Topics” and “Python: Regular Expressions” courses.

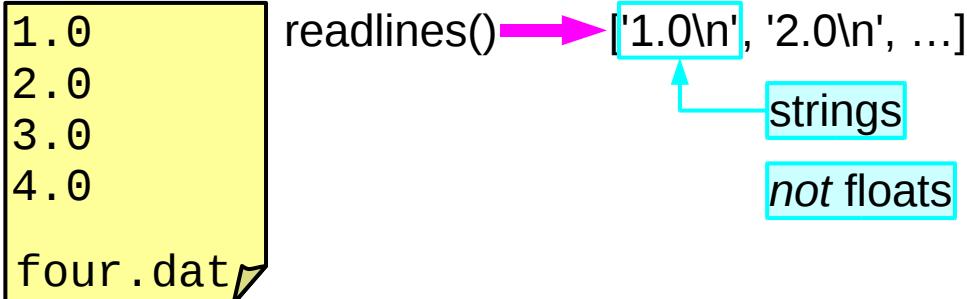
For details of the “Python: Further Topics” course, see:

<http://training.csx.cam.ac.uk/course/pythonfurther>

...and for details of the “Python: Regular Expressions” course, see:

<http://training.csx.cam.ac.uk/course/pythonregexp>

# Reading data gets you strings



```
>>> '1.0\n'.strip()  
'1.0'
```

A blue box labeled "Method to clear trailing white space" has an arrow pointing to the `.strip()` method call.

Still need to convert string to other types

131

As mentioned before, input from a file is read by Python as a string, complete with a trailing “new line” character.

One method of getting rid of unwanted “white space” (spaces, tabs, “new line” characters, etc.) is to use the `strip()` method of strings. `strip()` returns a copy of a string with all leading and trailing “white space” removed. This is often useful when dealing with strings read in from a file.

```
>>> '1.0\n'.strip()  
'1.0'  
>>> ' 1.0  \n'.strip()  
'1.0'
```

# Converting from one type to another

## In and out of strings

>>> <b>float('0.25')</b>	↔	>>> <b>str(0.25)</b>
0.25		'0.25'

>>> <b>int('123')</b>	↔	>>> <b>str(123)</b>
123		'123'

132

We need to be able to convert from the strings we read in to the numbers that we want. Python has some basic functions to do exactly this. Each is named after the type it generates and converts strings to the corresponding values. So `float()` converts strings to floating point numbers and `int()` converts strings to integers. Similarly, the `str()` function converts values into their string representations. The `float()` and `int()` functions will also strip any leading or trailing white space characters (spaces, tabs or new lines) from the string before converting it, which is very useful when working with numbers that were read in as strings from a file.

# Converting from one type to another

Between numeric types

```
>>> int(12.3)
```

```
12
```

loss of  
precision

```
>>> float(12)
```

```
12.0
```

133

The functions are slightly more powerful than just converting between strings and numeric types. They attempt to convert any input to the corresponding type so can be used to convert between integers and floating point numbers, and between floating point numbers and integers (truncating the fractional part in the process).

# Converting from one type to another

If you treat it like a list...

```
>>> list('abcd')
['a', 'b', 'c', 'd']

>>> list(data)
['line one\n', 'line two\n', 'line three\n', 'line four\n']

>>> list({'H':'hydrogen', 'He':'helium'})
['H', 'He']
```

134

Even more impressive is the `list()` function which converts things to lists. We have repeated the mantra several times that where Python expects a list it will treat an object like a list. So `file` objects are treated as lists of lines, dictionaries are treated as lists of keys and strings can even be treated as lists of characters. The `list()` function makes this explicit and returns that list as its result.

```
#!/usr/bin/python

# This script reads in some
# numbers from the file 'numbers.txt'.
# It then prints out the smallest
# number, the arithmetic mean of
# the numbers, and the largest
# number.
```

**What goes here?  
(Use the function  
you wrote in an  
earlier exercise.)**



135

Time for another exercise.

Write a script that reads in some numbers from the file “`numbers.txt`” in your course home directories. (It will obviously need to convert the strings it has read in into the appropriate numeric types.)

It should then print out the smallest number, the arithmetic mean (average) of the numbers, and the largest number. (You should use the function you wrote in one of the earlier exercises to do this.)

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

Also, if you have any questions on anything we’ve done so far, now would be a good time to ask them.

# This page intentionally left blank

Deze bladzijde werd met opzet blanco gelaten.

このページは計画的にブランクを残している

Ta strona jest celowo pusta.

Esta página ha sido expresamente dejada en blanco.

Эта страница нарочно оставлена пустой.

Denne side med vilje efterladt tom.

Pañon intence vaka.

این صفحه خالی است

An leathanach seo fágtha folamh in aon turas.

136

This page intentionally left blank: nothing to see here. If you're stuck for an answer to the exercise, have a look at the next page.

## Answer

```
#!/usr/bin/python

# This script reads in some
# numbers from the file 'numbers.txt'.
# It then prints out the smallest
# number, the arithmetic mean of
# the numbers, and the largest
# number.

import utils

data = open('numbers.txt')

numbers = []
for line in data:
    numbers.append(float(line))
del line

data.close()
del data

print utils.stats(numbers)
```

function you wrote  
in earlier exercise

137

Here's my answer to the exercise. Note that I'm using the `stats()` function I wrote in one of the earlier exercises, which I defined in my `utils` module – you should use the name of whatever function you created as your answer to the earlier exercise.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

# Output to files

```
input    = open('input.dat' )  
          ^  
          | default: read-only  
  
input    = open('input.dat', 'r')  
          ^  
          | read-only  
  
output  = open('output.dat','w')  
          ^  
          | write-only
```

138

To date we have been only reading from files. What happens if we want to write to them?

The `open()` function we have been using actually takes two arguments. The second specifies whether we want to read or write the file. If it is missing then the default is to open the file for reading only.

(We haven't described how to write functions with optional arguments, nor are we going to in this course – this is covered in the "Python: Further Topics" course; for details of this course see:

<http://training.csx.cam.ac.uk/course/pythonfurther>  
)

The explicit value you need to open a file for **reading** is the single letter string '`r`'. That's the default value that the system uses. The value we need to use to open a file for **writing** is '`w`'.

# Output to files

```
>>> output = open('output.dat', 'w')  
>>> output.write('alpha\n')           explicit "\n"  
>>> output.write('bet')            write(): writes  
>>> output.write('a\n')           lumps of data  
>>> output.writelines(['gamma\n', 'delta\n'])  
>>> output.close()                Flushes to  
                                    file system
```

139

As ever, a newly opened file has its position pointer (“offset”) pointing to the start of the file. This time, however, the file is empty. **If the file previously had content then it gets completely replaced.**

Apart from the explicit second argument, the `open()` function is used exactly as we did before.

Now that we’ve written our file ready to be written to we had better write something to it. There is no “`writeline()`” equivalent to `readline()`. What there is is a method “`write()`” which might be thought of as “`writelump()`”. It will write into the file whatever string it is given whether or not that happens to be a line.

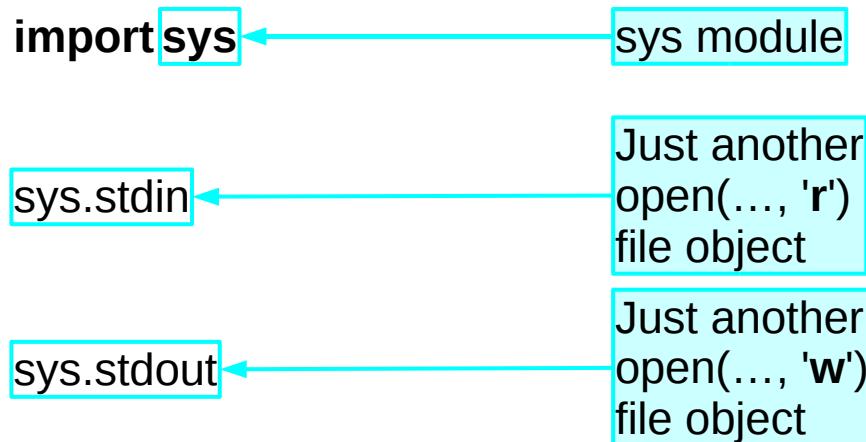
When we are writing text files it tends to be used to write a line at a time, but this is not a requirement.

There is a writing equivalent of `readlines()` too: “`writelines()`”. Again, the items in the list to be written do not need to be whole lines.

Closing the file is particularly important with files opened for writing. As an optimisation, the operating system does not write data directly to disc because lots of small writes are very inefficient and this slows down the whole process. When a file is closed, however, any pending data is “flushed” to the file on disc. This makes it particularly important that files opened for writing are closed again once finished with.

**It is only when a file is closed that the writes to it are committed to the file system.**

# Standard input and output



140

Let's move on to look at a couple of other ways to interface with our scripts. These will involve the use of a particular module, provided on all platforms: "sys", the system module.

First let's quickly recap what we mean by "standard input" and "standard output".

When a Unix command is run and has its input and output set up for it by the shell, e.g.

```
$ command.py < input.dat > output.dat
```

we refer to the data coming from `input.dat` as the standard input and the data going to `output.dat` as the standard output. It is critically important to understand that the shell doesn't just pass in the file names to the command. Instead, the shell does the opening (and closing) of the files and hands over file objects to the command.

(Note that above Unix command line is just an example command line we might use, it will not work if you actually try typing it in on the machines in front of you.)

So, what do standard input and output look like inside the Python system?

The `sys` module gives access to two objects called "`sys.stdin`" and "`sys.stdout`". These are `file` objects just as we got from the `open()` command. These come pre-opened, though, and will be closed for us automatically when the script ends.

# So, what does this script do?

Read lines in from standard input

Write them out again to standard output

It copies files, line by line

```
#!/usr/bin/python  
  
import sys  
  
for line in sys.stdin:  
    sys.stdout.write(line)
```

stdin-stdout.py

141

So let's look at a very simple script. It imports the `sys` module and runs through the lines of `sys.stdin`, the standard input. (Recall that if you treat a file object like a list you get the list of lines.) For each line it simply writes that line to `sys.stdout`, the standard output.

In essence it is a copier. Every line it reads in it writes out.

Obviously in practice we would do something more interesting with "line" between reading it in and writing it out. This functionality should be carved off into a function.

One approach is for the function to run a test on the line and return a Boolean according to whether or not the line passed. Then the line either is or isn't written out according to the answer.

```
for line in sys.stdin:  
    if test(line):  
        sys.stdout.write(line)
```

The other approach is for the function to transform the line in some fashion and the script then writes out the transformed line.

```
for line in sys.stdin:  
    sys.stdout.write(transform(line))
```

Or we can combine them:

```
for line in sys.stdin:  
    if test(line):  
        sys.stdout.write(transform(line))
```

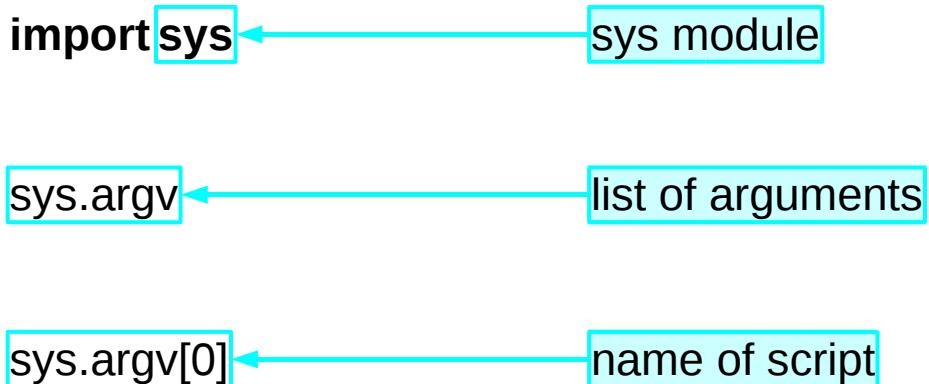
or:

```
for line in sys.stdin:  
    newline = transform(line)  
    if test(newline):  
        sys.stdout.write(newline)
```

As you may know, this model of reading in lines, possibly changing them and then possibly printing them out is called "filtering" and such scripts are often called "filters". Filters are covered in more detail in the "Python: Regular Expressions" course. For details of this course, see:

<http://training.csx.cam.ac.uk/course/pythonregexp>

# Command line



142

The next interaction with the system is to get at the command line. To date our interaction with the user has been with files, either opened explicitly inside the script or passed pre-opened by the calling shell. We want to exchange this now to allow the user to interact via arguments on the command line.

There's another object given to us by the `sys` module called "argv", which stands for "**a**rgument **v**alues".

Item number zero in `sys.argv`, i.e. `sys.argv[0]` is the name of the script itself.

Item number one is the first command line argument (if any).

Item number two is the second command line argument (if any), and so on.

Note that all the items in `sys.argv` are *strings*, regardless of whether the command line arguments are meant to be interpreted as strings or as numbers or whatever. If the command line argument is not intended to be a string, then you need to convert it to the appropriate type.

```
#!/usr/bin/python  
  
print sys.argv[0]  
print sys.argv
```

args.py

```
$ python args.py 0.25 10  
  
args.py  
['args.py', '0.25', '10']
```

NB: list of *strings*

143

There is a script called `args.py` in your course home directories. You can use this to investigate what the command line arguments of your script look like to Python.

Again, the most important thing to note is that Python stores these arguments as a list of *strings*.

```
#!/usr/bin/python

# This script takes some numbers as
# arguments on the command line.
# It then prints out the smallest
# number, the arithmetic mean of
# the numbers, and the largest
# number.
```

← **What goes here?**

144

Time for another exercise.

Write a script that takes some numbers as command line arguments. (Your script will obviously need to do some string conversion.)

It should then print out the smallest number, the arithmetic mean (average) of the numbers, and the largest number. (You should use the function you wrote in one of the earlier exercises to do this.)

Make sure you test your script.

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

Also, if you have any questions on anything we've done so far, now would be a good time to ask them.

## Answer

```
#!/usr/bin/python

# This script takes some numbers as
# arguments on the command line.
# It then prints out the smallest
# number, the arithmetic mean of
# the numbers, and the largest
# number.

import sys
import utils

numbers=[]
for arg in sys.argv[1:]:
    numbers.append(float(arg))
del arg
print utils.stats(numbers)
```

function you wrote earlier

145

Here's my answer to the exercise. Note that I'm using the `stats()` function I wrote in one of the earlier exercises, which I defined in my `utils` module – you should use the name of whatever function you created as your answer to the earlier exercise.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

```
def find_root(  
    ...  
):  
    """find_root(function, lower, upper, tolerance)  
    finds a root within a given interval to within a  
    specified tolerance. The function must take  
    values with opposite signs at the interval's ends."""  
  
    while upper - lower < tolerance:  
        middle = (lower + upper) / 2.0  
        if function(middle)*function(upper) > 0.0:  
            upper = middle  
        else:  
            lower = middle  
  
    return (lower, upper)
```

Inserted string

utils.py

146

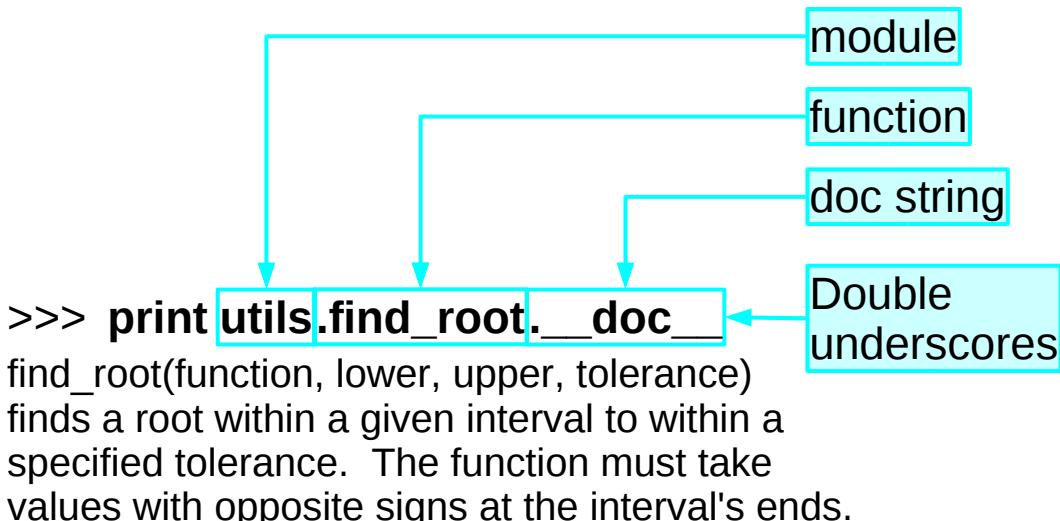
We are going to end with a little frill that is enormously useful. We are going to add some extra documentation to our functions and modules which they can then carry around with them and make accessible to their users.

We edit `utils.py` to insert some text, as a string, immediately after the “`def`” line and before any actual code (the body of the function). This string is long so is typically enclosed in triple double quotes, but this isn’t required; any string will do. The text we include is documentation for the *user*. This is not the same as the comments in the code which are for the programmer.

A string placed here does not affect the behaviour of the function.

# Doc strings for functions

```
>>> import utils
```



147

So how do we get at it?

If we import the module containing the function we can print the “`__doc__`” attribute of the module’s function. (Just as methods are built in *functions* of Python objects, so “*attributes*” are *variables* that are built in to Python objects. We use the same dot (.) syntax that we’ve used for methods to get at attributes of objects.)

The “`__doc__`” attribute of the function gives us access to the string we added to the function’s definition. Ultimately, we will be sharing the modules with other people. This is how we can share the documentation in the same file.

Note that it is a ***double underscore*** on each side of “`__doc__`”. This isn’t clear in all fonts.

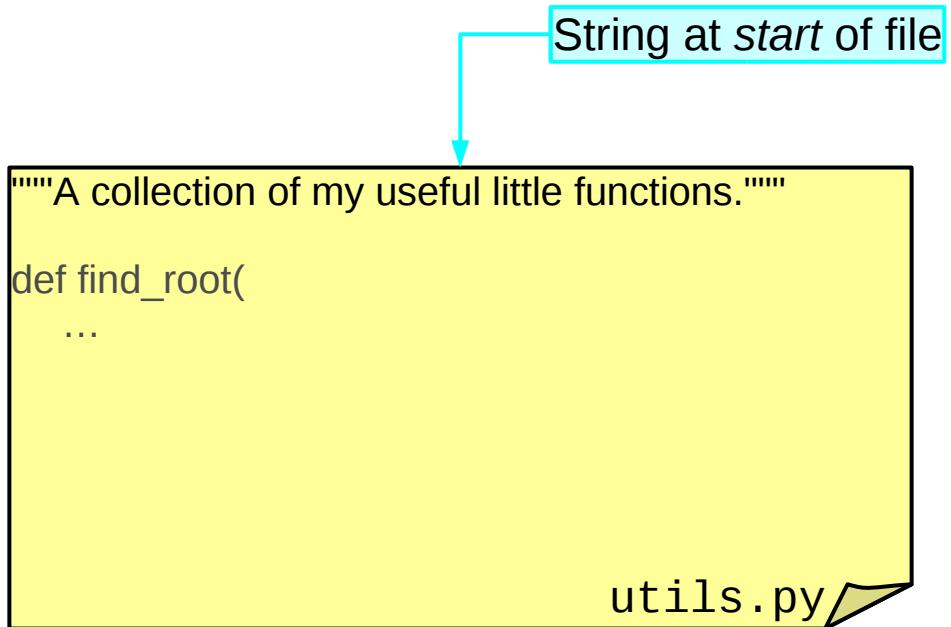
All the system modules come copiously documented:

```
>>> import sys
>>> print sys.exit.__doc__
exit([status])
```

Exit the interpreter by raising `SystemExit(status)`.  
If the status is omitted or `None`, it defaults to zero (i.e., success).  
If the status is numeric, it will be used as the system exit status.  
If it is another kind of object, it will be printed and the system  
exit status will be one (i.e., failure).

If you need to work with a module, the doc strings are a good place to look first.

# Doc strings for modules

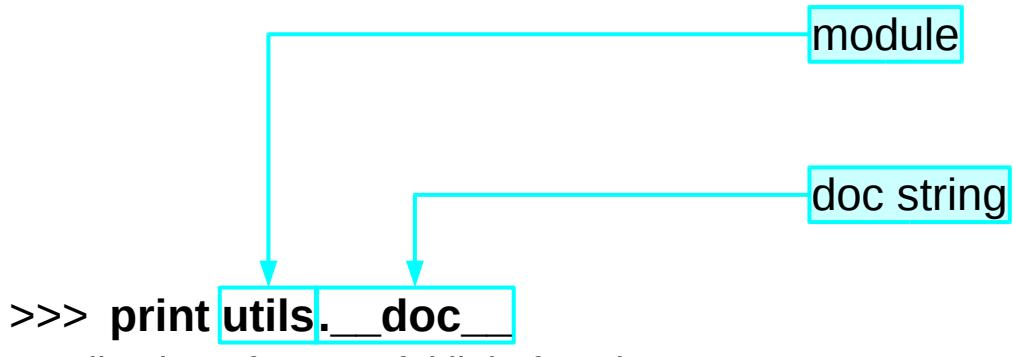


If we can document a function, can we document a module? Yes.

We insert a similar string at the top of the module file, before any Python code.

# Doc strings for modules

```
>>> import utils
```



A collection of my useful little functions.

149

We get at it in the exact same way we got at a function's doc string, except that this time no function name is involved.

```
>>> import sys  
>>> print sys.__doc__
```

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

```
argv -- command line arguments; argv[0] is the script pathname if known  
path -- module search path; path[0] is the script directory, else ''  
modules -- dictionary of loaded modules
```

...

All system modules have doc strings describing their purpose and contents.

## Final exercise

```
#!/usr/bin/python
```

```
# This script takes some atomic symbols  
# on the command line. For each symbol,  
# it prints the atomic element's name, and  
# boiling point (to 2 decimal places),  
# e.g. for the symbol 'H', print:  
# hydrogen has a boiling point of 20.3K  
# Finally, it tells you which of the given  
# atomic elements has the lowest atomic  
# number.  
  
# The keys of this dictionary are the  
# symbols for the atomic elements.  
# The values are tuples:  
# (name, atomic number, boiling point)  
chemicals = {...}           chemicals4.py
```

**Write the rest of  
this script.**

150

We'll leave you with one final exercise and some references for further information about Python.

In your course home directories you will find an incomplete script called `chemicals4.py`.

Complete this script so that it accepts the symbols for atomic elements on the command line. For each symbol it gets from the command line, it should print out the name of the atomic element and its boiling point in Kelvin to 2 decimal places in the manner given in the example on the slide above.

It should then tell you which of the specified atomic elements has the lowest atomic number.

Make sure you test your script.

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

You can take a copy of the `chemicals4.py` file away with you and work on it at your leisure, but that does mean that we won't be able to help you should you run into any problems.

Finally, if you have any questions on anything we've covered in the course, now is your last opportunity to ask them.

(And again, for the pedants reading, the authors know that the `chemicals` dictionary in this script really contains the atomic elements rather than chemicals, and so it would be better to call this dictionary "elements" rather than "chemicals". However, as we said, we want to avoid confusion with the use of the word "elements" in "elements of lists" or "elements of dictionaries" to refer to the individual items in lists or dictionaries.)

# References and further courses

## Dive Into Python

Mark Pilgrim

Apress

ISBN: 1-59059-356-1

<http://diveintopython.org/>

Best book on Python your course presenter has found. (It was written for Python 2.3, though. Luckily, Python 2.4, 2.5 and 2.6 are very similar to Python 2.3.)

Official Python documentation: <http://docs.python.org/>

“**Python: Further Topics**” is the follow-on course from this one. For details of this and other University Computing Service courses on Python, see:

<http://training.csx.cam.ac.uk/theme/scicomp?scheduled=all>

151

If you need to learn more Python than that covered in this course, have a look at the other Python courses in the “Scientific Computing” series:

<http://training.csx.cam.ac.uk/theme/scicomp?scheduled=all>

The course which directly follows on from this course is the “**Python: Further Topics**” course:

<http://training.csx.cam.ac.uk/course/pythonfurther>

The official Python documentation can be found on-line at:

<http://docs.python.org/>

You can either browse it on-line or download it in a variety of formats. It is extremely comprehensive and covers all aspects of the language and all the core modules supplied with Python. It also includes a tutorial on Python.

There are also a number of good books on Python out there that you may wish to consult. Probably the best book on Python your course presenter has found is “**Dive Into Python**” by Mark Pilgrim. It also has the advantage of being available free, in a variety of electronic formats, at:

<http://diveintopython.org/>

If, however, you find this book as useful as your course presenter does, then you should really consider buying a paper copy as such excellent free resources deserve our support. The one caveat regarding this book is that when it was written the major release of Python then current was 2.3. At the time of writing these notes, the current major release of Python in widespread use is 2.6. Fortunately, Python 2.4, 2.5 and 2.6 are not wildly different to Python 2.3, so this shouldn’t cause too many problems. It does mean, though, that some of the external modules referred to in the book have either been superseded or are no longer maintained/available.

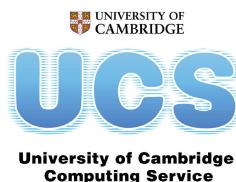
# Chapter 4

## Advanced Python

# Python: Further Topics

## Day One

Bruce Beckles  
University of Cambridge Computing Service



1

Note that this course covers Python 2.4 to 2.7, which are the most common versions currently in use – it does **NOT** cover the recently released Python 3.0 (or 3.1) since that version of Python is so new. Python 3.0 is significantly different to previous versions of Python, and this course will be updated to cover it as it becomes more widely used.

The official UCS e-mail address for all scientific computing support queries, including any questions about this course, is:

[scientific-computing@ucs.cam.ac.uk](mailto:scientific-computing@ucs.cam.ac.uk)

# Introduction

- Who:
  - Bruce Beckles, e-Science Specialist, UCS
- What:
  - Python: Further Topics course, *Day One*
  - Part of the **Scientific Computing** series of courses
- Contact (questions, etc):
  - [scientific-computing@ucs.cam.ac.uk](mailto:scientific-computing@ucs.cam.ac.uk)
- Health & Safety, etc:
  - Fire exits
- **Please switch off mobile phones!**

2

As this course is part of the Scientific Computing series of courses run by the Computing Service, all the examples that we discuss will be more relevant to scientific computing than to other programming tasks.

This does not mean that people who wish to learn about Python for other purposes will get nothing from this course, as the techniques and underlying knowledge taught are generally applicable. However, such individuals should be aware that this course was not designed with them in mind.

Note that there are various versions of Python in use, the most common of which are releases of Python 2.2, 2.3, 2.4, 2.5 and 2.6. (The material in this course is applicable to versions of Python in the 2.4 to 2.7 releases.)

On December 3rd, 2008, Python 3.0 was released. Python 3.0 is significantly different to previous versions of Python, is not covered by this course, and breaks backward compatibility with previous Python versions in a number of ways. As Python 3.0 and 3.1 become more widely used, this course will be updated to cover them.

# Related/Follow-on courses

## “Python: Operating System Access”:

- Accessing the underlying operating system (OS)
- Standard input, standard output, environment variables, etc

## “Python: Regular Expressions”:

- Using *regular expressions* in Python

## “Programming Concepts: Pattern Matching Using Regular Expressions”:

- Understanding and constructing regular expressions

## “Python: Checkpointing”:

- More robust Python programs that can save their current state and restart from that saved state at a later date
- “Python: Further Topics” is a pre-requisite for the “Python: Checkpointing” course

## “Introduction to Gnuplot”:

- Using *gnuplot* to create graphical output from data

3

For details of the “Python: Operating System Access” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonopsys>

For details of the “Python: Regular Expressions” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonregexp>

For details of the “Programming Concepts: Pattern Matching Using Regular Expressions” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-regex>

If you are unfamiliar with regular expressions, the following Wikipedia article gives an overview of them:

[http://en.wikipedia.org/wiki/Regular\\_expression](http://en.wikipedia.org/wiki/Regular_expression)

...although that article does not express itself as simply as it might, so it may be most useful for the references it gives at the end. If you have met regular expressions before, but haven't yet used them in Python, then the “Python: Regular Expressions” course will teach you how to use them in Python. Alternatively, the Python “*Regular Expression HOWTO*” introductory tutorial also provides a good introduction to using regular expressions in Python:

<http://docs.python.org/howto/regex>

For details of the “Python: Checkpointing” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonchkpt>

For the notes of the “Introduction to Gnuplot” course, see:

<http://www-uxsup.csx.cam.ac.uk/courses/Gnuplot/>

If you are unfamiliar with gnuplot, you may wish to have a look at its home page:

<http://www.gnuplot.info/>

# Pre-requisites

- Ability to use a text editor under Unix/Linux:
  - Try gedit if you aren't familiar with any other Unix/Linux text editors
- Basic familiarity with the Python language (as would be obtained from the “[Python: Introduction for Absolute Beginners](#)” or “[Python: Introduction for Programmers](#)” course):
  - Interactive and batch use of Python
  - Basic concepts: variables, flow of control, functions, Python's use of indentation
  - Simple data manipulation
  - Simple file I/O (reading and writing to files)
  - Structuring programs (using functions, modules, etc)

4

For details of the “Python: Introduction for Absolute Beginners” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-python>

For details of the “Python: Introduction for Programmers” course, see:

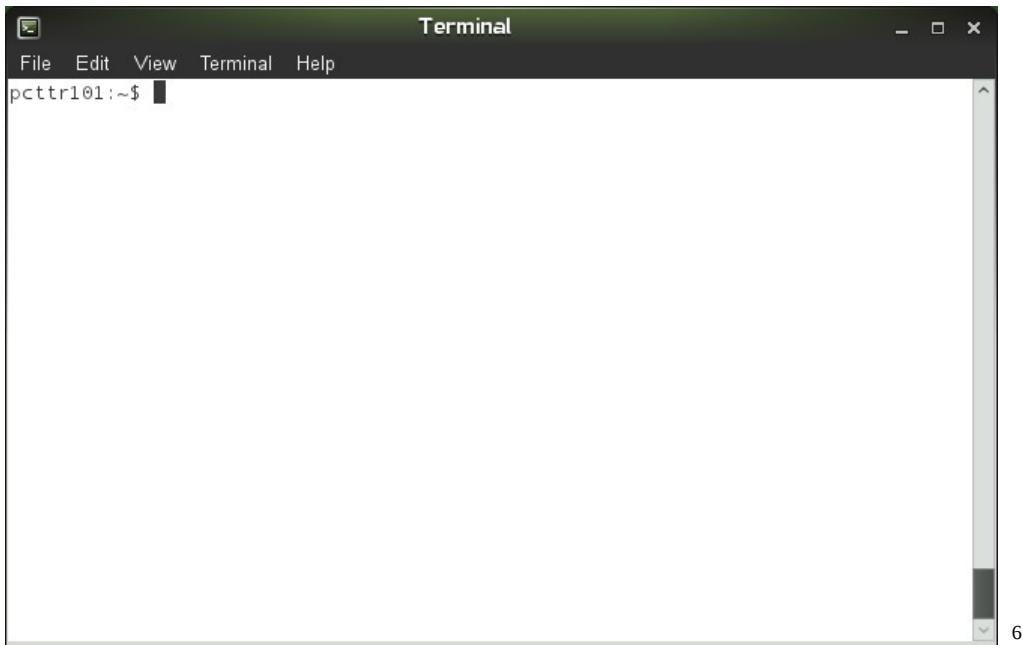
<http://www.training.cam.ac.uk/ucs/course/ucs-python4progs>

# Start a shell



5

# Screenshot of newly started shell



# None



**None** is a special value in Python, with its own data type (**NoneType**). It is Python’s way of representing “nothing”. Its “truth value” is **False** (i.e. for the purpose of tests it is equivalent to **False**).

It is often used as “placeholder” value, or to mean that there is “no data”.

```
>>> None          >>> not None  
>>>                   True  
  
>>> type(None)  
<type 'NoneType'>
```

7

The value **None** is a Python special value (Python calls it a “null object”) designed for situations when you need a value, but that value should not represent anything. For instance, it is often used as a “placeholder” for variables that will be assigned a value later in the script. It has its own separate type (**NoneType**). For the purpose of tests, it is equivalent to **False** (i.e. its “truth value” is **False**).

Many Python functions use **None** to mean that there are no appropriate value(s) for whatever they were asked to do. We will see some examples of how it can be used later in this course.

It is also the value that is returned by a function that doesn’t explicitly **return** anything else. So if your function does not explicitly **return** anything, then it actually returns **None**.

# Controlling loops in Python: **break**

```
while x % 2 == 0 :  
    print x, 'still even'  
    x = x/2  
    break  
  
for prime in primes :  
    print prime  
    if prime > 5 :  
        break
```

**break** Stop executing loop and go to statement immediately after loop

8

The **break** statement causes Python to stop executing whatever loop it might be executing and jump to the first statement immediately after that loop. If you have nested loops (i.e. loops within loops, e.g. a **for** loop within a **while** loop), the **break** statement will terminate the current innermost loop, transferring control to the next statement in the containing loop.

If your loop has an **else** block (**while** and **for** loops can have **else** blocks, although these are seldom used) then **break** will skip any statements in the **else** block, and jump to the first statement after the loop *and* its **else** block.

(If a **while** loop has an **else** block, the statements in this block will be executed when the test in the **while** loop evaluates to **False**, after which the rest of the script will be executed. If a **for** loop has an **else** block, the statements in the **else** block will be executed after the **for** loop completes, after which the rest of the script will be executed.)

## Controlling loops in Python: **continue**

```
while x % 2 == 0 :  
    if x == 4 :  
        continue  
    x = x/2  
  
for prime in primes :  
    print prime  
    continue  
    print 'This line never executed' 9
```

Stop executing this iteration of the loop and go to the next iteration

The **continue** statement causes Python to stop executing the current iteration of whatever loop it might be executing and start on the next iteration of that loop. If you have nested loops (i.e. loops within loops, e.g. a **for** loop within a **while** loop), the **continue** statement will start the next iteration of the current innermost loop.

## Function Arguments

```
>>> import utils  
>>> utils.greet("Afternoon", "Bruce")  
Good Afternoon, Bruce.
```

1st argument

```
>>> utils.greet(person="Bruce", time="Afternoon")  
Good Afternoon, Bruce.
```

2nd argument

named argument: person

named argument: time

10

We have already seen that when we use a function in Python we can give it some arguments (parameters). Up to now we have specified the function's arguments (its input) by *position*. We have called the function, specifying its arguments, and the **position** of each argument determines what the function does with that argument. Such arguments are called *positional arguments*, and the order in which they are given to the function is crucial. In the example above, the **greet()** function (a function I have specially written for this course and put in the **utils** module in your course home directory) prints out a greeting on the screen – the first argument is the time of day (morning/afternoon/evening/etc) and the second argument is the name of the person to whom the greeting is addressed.

Python also has another way of organising a function's arguments: rather than the position of an argument being used to determine what the function should do with it, the argument is given a *name* (Python calls the name a “*keyword*”), and that **name** determines what the function does with the argument. This allows the arguments to be given to the function in any order. Such arguments are called *named arguments* (or *keyword arguments*). In Python, the functions that the programmer writes automatically support both positional arguments and named (or keyword) arguments, and we can use either mechanism to specify the functions arguments when we use the function. The advantage of using named arguments is that we can specify them in any order we like, and the purpose of each argument is immediately clear (assuming the programmer chose sensible names for the arguments).

Note that some of the built-in functions in Python do not support named arguments, but the functions that you write will do so automatically.

# Default Values and Optional Arguments

**person** is now an  
*optional* argument

Default value for **person**

```
def greet(time, person='user'):  
    print "Good %s, %s." % (time, person)
```

utils.py

```
>>> import utils  
>>> utils.greet('Afternoon')  
Good Afternoon, user.
```

11

Open up the **utils.py** file in your course home directories with a text editor and edit the definition of the **greet()** function as shown above (note that to fit everything on the slide we haven't shown the function's doc string), i.e. add:

**'user'**

immediately after "person" on the first line (the line with the def keyword) of the greet() function's definition. Thus first line should now look like this:

```
def greet(time, person='user'):
```

The rest of the function is unchanged. Make sure you save the file after you've made this change. (If you have been following along in the Python interpreter and still have a copy of the interpreter running after you've edited the **utils.py** file, quit the interpreter and restart it before trying the changed **greet()** function.)

This function now has what is called "a *default value*" for the **person** argument (parameter). If we import the **utils** module, and then call the **greet()** function without specifying a value for the **person** argument, then **person** will be set to this default value, i.e.

```
>>> utils.greet('Afternoon')  
Good Afternoon, user.
```

is exactly the same as

```
>>> utils.greet('Afternoon', 'user')  
Good Afternoon, user.
```

This means that **person** is now an *optional* argument: we no longer have to specify a value for it when we call the function; if we don't specify a value, Python will use the default value we've given in the function definition.

**VERY IMPORTANT:** Once you've defined a default value for one argument in the function definition, you need to define default values for *all* the *following* arguments taken by the function. So, in the **greet()** function, if we had given a default value for the **time** argument, we would **have** to also specify one for the **person** argument as well.

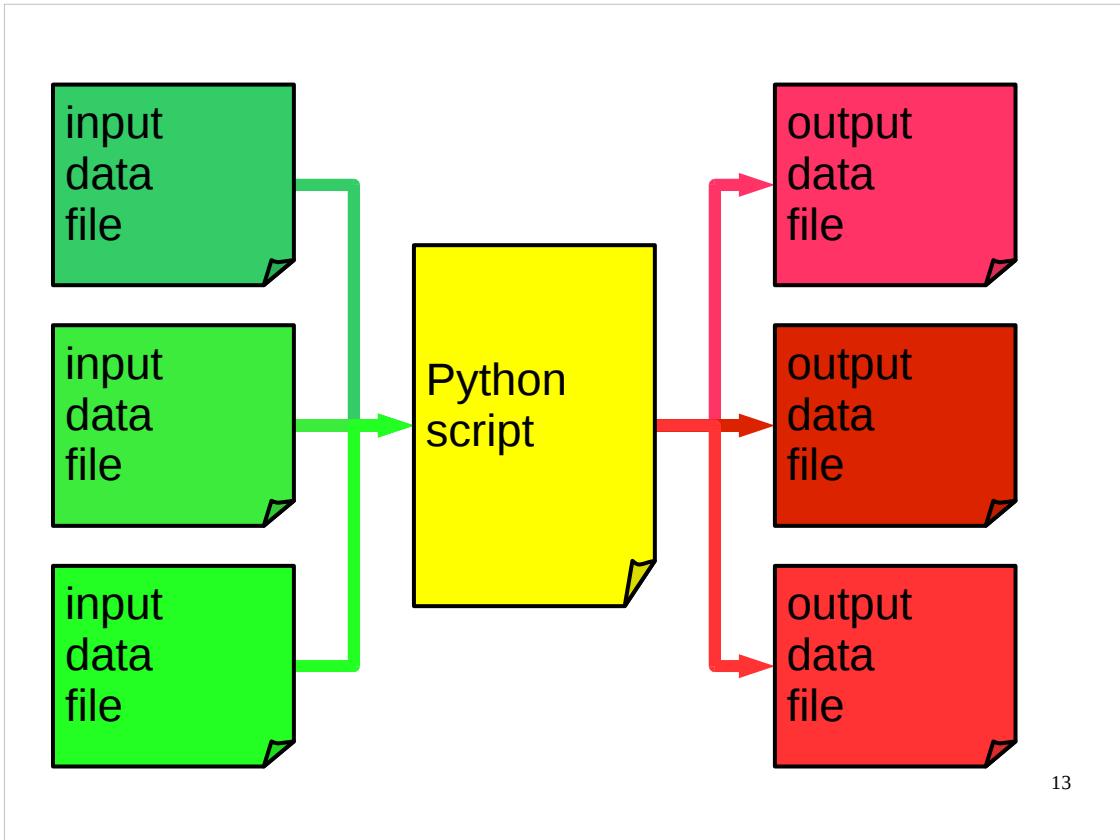
# Accessing files

1. Direct access to files
2. Structured files: **csv** module

12

Today, we're going to examine file I/O (input and output) in Python. We'll start off with a quick recap of the basics (as was covered in the "Python: Introduction for Absolute Beginners" course, and in a more abbreviated fashion in the "Python: Introduction for Programmers" course) and then move on to more advanced topics.

First we will consider directly accessing files ourselves, and then move on to how we can access structured files with the help of the **csv** module.



We want to be able to directly access files from Python. In particular, we want to be able to cope with the situation where there is more than one input and/or output file.

# Reading a file

1. Opening a file
2. Reading from the file
3. Closing the file

14

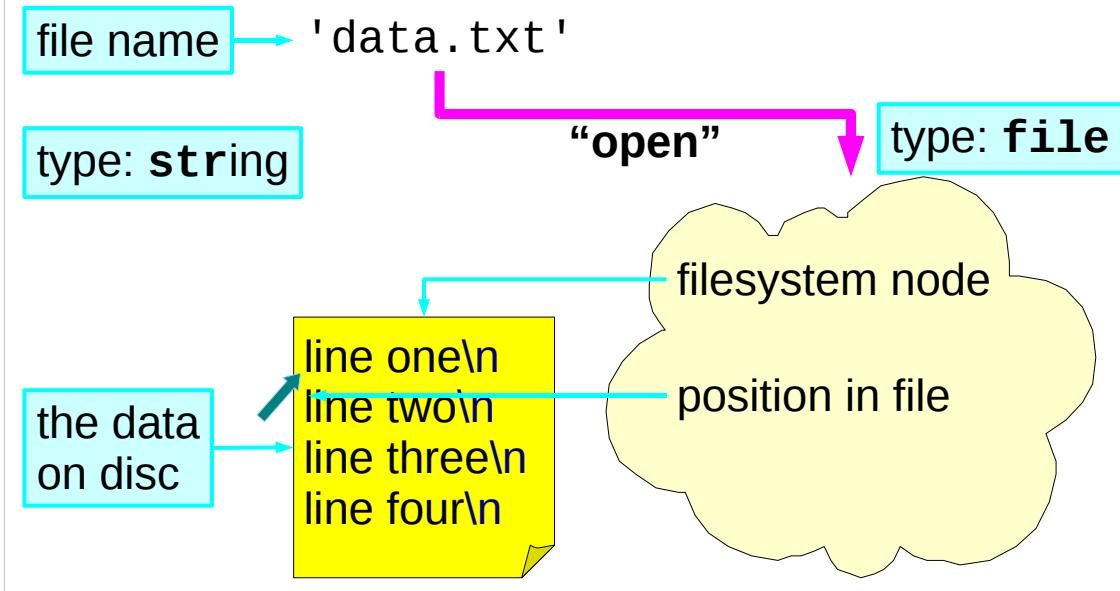
So, we will start by reminding ourselves how we read a file in Python. There are three phases if we start with a file name.

We “open” the file. This takes the name of the file, checks to see if it exists and gives us a “handle” – a special type of data type known as a **file** object – on the contents of the file itself.

Then we use that **file** object to read the file’s contents.

Then we disconnect from the file by declaring that we are done with it now. This is called “closing” the file (in contrast to “opening” it in the first place).

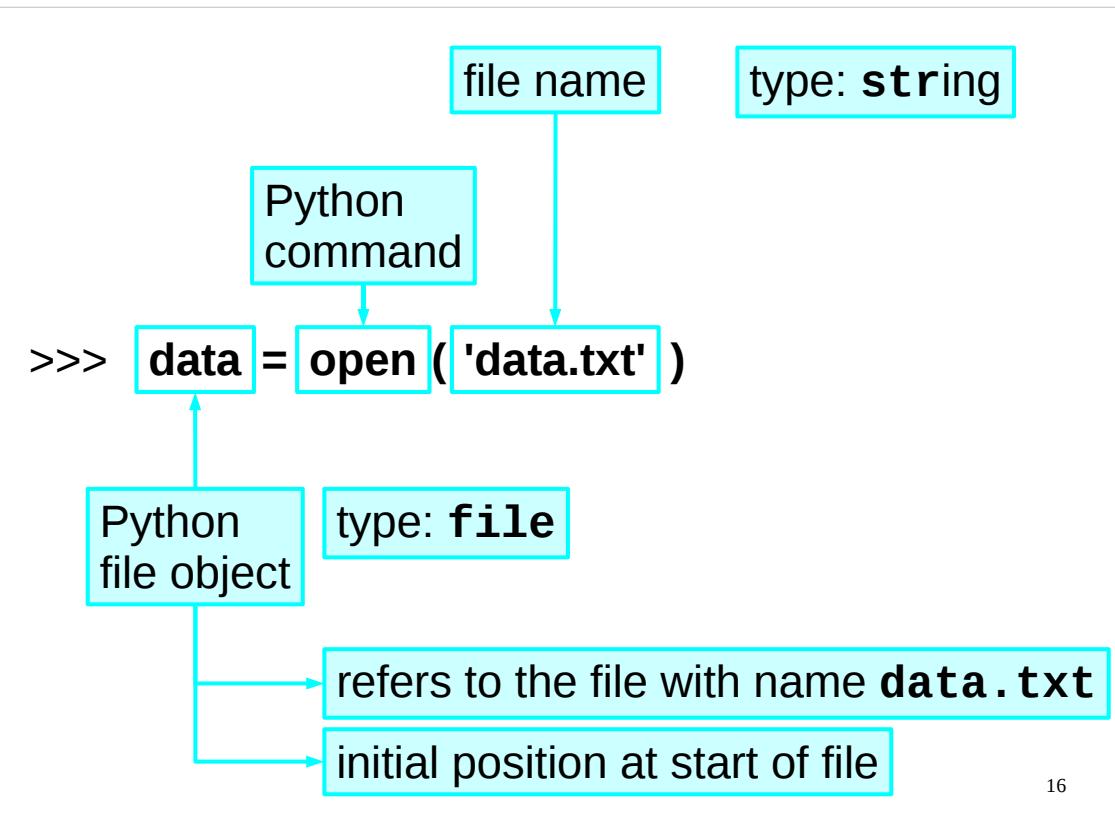
# Opening a file



15

Opening a file involves taking the file name and getting some Python object whose internals need not concern us – it's a Python type called “**file**” (logically enough). If there is no file of the name given, or if we don't have permission to get at this file, then the opening operation fails. If it succeeds we get a **file** object that has two properties of interest to us. It knows the file on disc that it refers to, obviously. But it also knows how far into the file we have read so far. This property, known as the “offset”, obviously starts at the beginning of the file when it is first opened. As we start to read from the file the offset will change.

Think of opening a file, given its name, as being in a library. You are given a book's name, you fetch the book from the shelves (if it exists and you have permission), you open the book and place your finger under the first letter of the first word of the book, ready to read.



In Python, we open a file with the “**open**” command. (You may also meet some scripts that use the “**file**” command instead of the “**open**” command. The “**file**” command is used to create **file** objects and so can be used to do the same thing as the “**open**” command, but this is not really a very good idea. Stick to the “**open**” command for opening files.)

In your Python directories there is a file called “**data.txt**”. If you enter Python interactively and give the command

```
>>> data = open('data.txt')
```

then you should get a **file** object for this file inside the Python interpreter.

Note that we just gave the name of the file, we didn’t say where it was. If we don’t give a path to the file then Python will look in the current directory. If we want to open a file in some other directory then we need to give the path as well as the name of the file to the **open** command. For instance, if we wanted to open a file called “**data.txt**” in the **/tmp** directory, we would use the **open** command like this: **open('/tmp/data.txt')**.

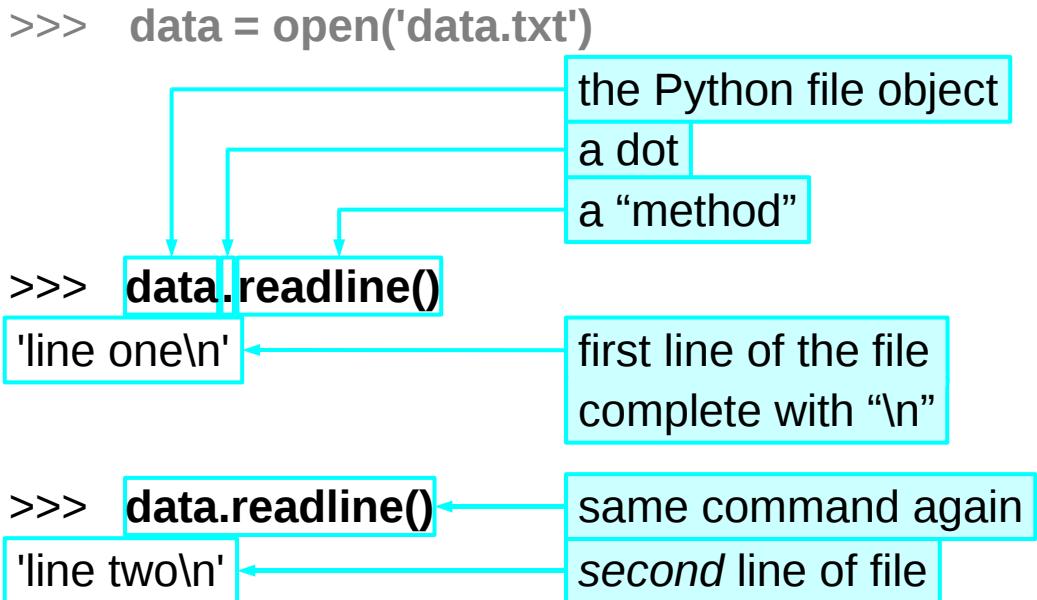
If you want to know which directory is your current directory, you can use a function called **getcwd()** (“**get current working directory**”) that lives in the **os** module:

```
>>> import os
>>> os.getcwd()
'/home/x282'
```

(If you try this on the computer in front of you, you will find that it displays a different directory to the one shown in these notes.)

You can change your current directory by using the **chdir()** (“**change directory**”) function, which also lives in the **os** module. You give the **chdir()** function a single argument: the name of the directory you want to change to, e.g. to change to the **/tmp** directory you would use **os.chdir('/tmp')**. However, **don’t try this now**, as if you change the current directory to something other than your course home directory then many of the examples in these notes will no longer work! (If you have foolishly ignored my warning and changed directory, and don’t remember what your course home directory was called (and so can’t change back to it), the easiest thing to do is to quit the Python interpreter and then restart it.)

# Reading a file



17

To read a file line by line (which is typical for text files), the **file** object provides a method to read a single line. (Recall that methods are the “built in” functions that objects can have.) The method is called “**readline()**” and the **readline()** method on the **data** object is run by asking for “**data.readline()**” with the object and method name separated by a dot.

There are two important things to notice about the string returned. The first is that it’s precisely that: one line, and the first line of the file at that. The second point is that it comes with the trailing “new line” character, shown by Python as “`\n`”.

Now observe what happens if we run exactly the same command again. (Remember that Python on PWF Linux has a history system. You can just press the up arrow once to get the previous command back again.) This time we get a different string back. It’s the second line.

```
>>> data = open('data.txt')
```

position:  
start of file

data

line one\n  
line two\n  
line three\n  
line four\n

18

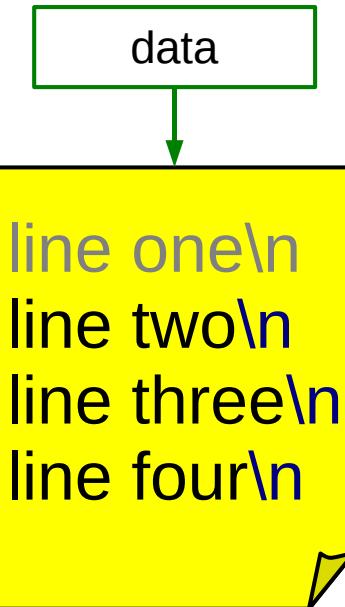
Let's take a closer look at what just happened to be sure we understand how reading from a file works. We started with the **file** object for the “**data.txt**” file having its offset point to the start of that file. That's what we always get immediately after an **open()**.

```
>>> data = open('data.txt')
```

```
>>> data.readline()
```

```
'line one\n'
```

position:  
after end of first line,  
at start of second line



19

Then we ran **data.readline()**. This read one line from the file and returned it to us in a string. As it did so it moved the offset along to just beyond the last character read. It's now at the start of the second line.

In our book analogy, as you read the words you slide your finger along. It's now at the start of the second line ready for you to read that.

```
>>> data = open('data.txt')
```

```
>>> data.readline()
```

```
'line one\n'
```

```
>>> data.readline()
```

```
'line two\n'
```

position:  
after end of read data,  
at start of unread data

data

line one\n  
line two\n  
line three\n  
line four\n

20

Next time we call **readline()** we get the line following on from the current position in the file.

(It is possible to put the position in the middle of a line using a method that I'm not going to mention yet. What **readline()** generates is everything from the current position to the end of the line, including the new line character.)

```
>>> data.readline()
```

```
'line one\n'
```

```
>>> data.readline()
```

```
'line two\n'
```

```
>>> data.readlines()
```

```
[ 'line three\n', 'line four\n' ]
```

remaining unread  
lines in the file

21

Just so you know, there's another method which is occasionally useful. The “**readlines()**” method gives all the lines from the current position to the end as a list of strings.

We won't use **readlines()** much as there is a better way to step through the lines of a file.

```
>>> data = open('data.txt')
```

data

```
>>> data.readline()
```

'line one\n'

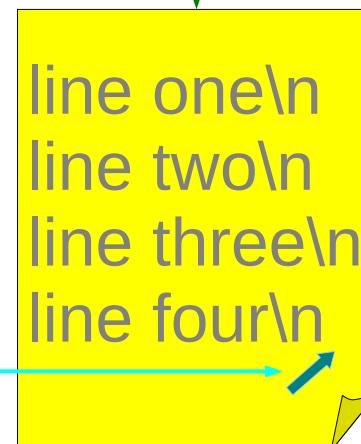
```
>>> data.readline()
```

'line two\n'

```
>>> data.readlines()
```

[ 'line three\n', 'line four\n' ]

position:  
at end of file



22

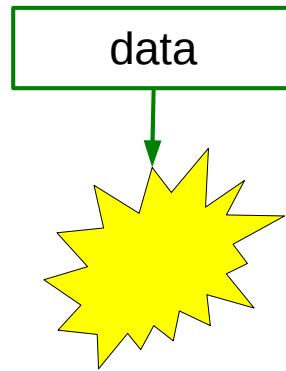
Once we have read to the end of the file the position marker points to the end of the file and no more reading is possible (without changing our position in the file, which we're not going to discuss yet).

# Closing a file

```
>>> data.readlines()  
[ 'line three\n', 'line four\n' ]
```

```
>>> data.close()
```

```
disconnect
```



23

The method to close a file is, naturally, “**close()**”.

It’s only at this point that we declare to the underlying operating system (Linux in this case) that we are finished with the file. On operating systems that lock down files while someone is reading them, it is only at this point that someone else can access the file.

Closing files when we are done with them is important, and even more so when we come to examine writing to them.

```
>>> data.readlines()  
[ 'line three\\n', 'line four\\n' ]
```

```
>>> data.close()
```

```
>>> del data
```

delete the variable if we  
aren't going to use it again

24

We should practice good Python variable hygiene and delete the **data** variable if we aren't going to use it again immediately.

# Common trick

```
for line in data.readlines():
    stuff
```

```
for line in data:
    stuff
```

Python “magic”:  
treat the file like  
a list and it will  
behave like a list

25

Some Python objects have the property that if you treat them like a list they act like a particular list. **file** objects act like the list of lines of the file, but be warned that as you run through the lines you are running the offset position forward, i.e. you won’t get the same results twice (in fact, most likely you won’t get anything the second time) unless you move the offset back to where it was. (We’ll see how to do this later.)

# Putting it all together in a function

1. Take a file name as the function argument.
2. Read a file of “key/value” lines.
3. Create the equivalent dictionary.
4. Return the dictionary.

```
H hydrogen
He helium
Li lithium
Be beryllium
B boron
```

chemicals.txt

26

So let’s look at a sensible example. In the “Python: Introduction for Absolute Beginners” course we created a function – that we put in a module called “**utils**” – that took a file name as its argument, read in the lines, expecting each line to consist of two simple words, and returned a Python dictionary where the first word on each line is a key of the dictionary and the second word is the corresponding value.

In your course home directory you will find this **utils** module (the module is in a file called **utils.py**), containing the function that does this. The function is called “**file2dict()**” and we’ll examine it now to see what it does. Then we’ll try improving it.

(For the pedants reading, the author knows that the file we’re using this function on really contains the atomic elements rather than chemicals, and so it would be better to call it “**elements.txt**” rather than “**chemicals.txt**”. However, the word “element” is often used to refer to the individual items in lists or dictionaries (as in “an element of a list” or “an element of a dictionary”), and I’d rather avoid any confusion.)

# The function

1. Create an empty dictionary.
2. Open the file.
3. For each line in the file:
  - 3a. Split the line into two strings (key & value).
  - 3b. Add the key and value to the dictionary.
4. Close the file.
5. Return the dictionary.

27

This is what the function does in words...

Simple, really.

```
def file2dict(filename):
    dict = {}
    data = open(filename)
    for line in data:
        [ key, value ] = line.split()
        dict[key] = value
    data.close()
    return dict
```

utils.py

28

...and this is what it does in Python.

We know how to create a function – chiefly, we need to decide on its name and the name(s) of its argument(s). This function is called “**file2dict()**” and its single argument is “**filename**”. Remember that the body of the function must be indented.

This function takes the approach to creating a dictionary of starting with an empty one and adding entries as it proceeds. So first it creates an empty dictionary.

It needs access to the content of the file so it **opens** the file to read it.

Next it needs to step through the lines of the file. It uses the standard Python trick of treating something like a list and having it behave like a list.

Then it needs to split the line into two words. It does this using the **split()** method (with which you should be familiar). Note that **split()** discards all white space, including the new line character as well as the spaces between the words. Observe that it assigns the list of words generated to a list of two variable names. If any lines in the file don't consist of exactly two words, this line (and the function) will fail.

Having got two words it adds them into the dictionary.

Once it finishes with the file it **closes** it.

Now that it is done it hands back the dictionary, and the function ends.

```
#!/usr/bin/python
import utils

chemicals = utils.file2dict('chemicals.txt')
utils.print_dict(chemicals)
```

mkdict.py

29

So we can use it to create a dictionary from a text file structured in a particular way.

(Again, for the pedants reading, the author knows that the text file really contains the atomic elements rather than chemicals, and so it would be better to call the dictionary we create “**chemicals**” rather than “**elements**”. However, as I said, I want to avoid confusion with the use of the word “elements” in “elements of lists” or “elements of dictionaries” to refer to the individual items in lists or dictionaries.)

# What if something goes wrong?

```
>>> data = open('output')
```

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

  IOError: [Errno 2] No such file or directory: 'output'

output

```
>>> data = open('data.txt')
```

```
>>> data.readlines()
```

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

  IOError: [Errno 13] Permission denied

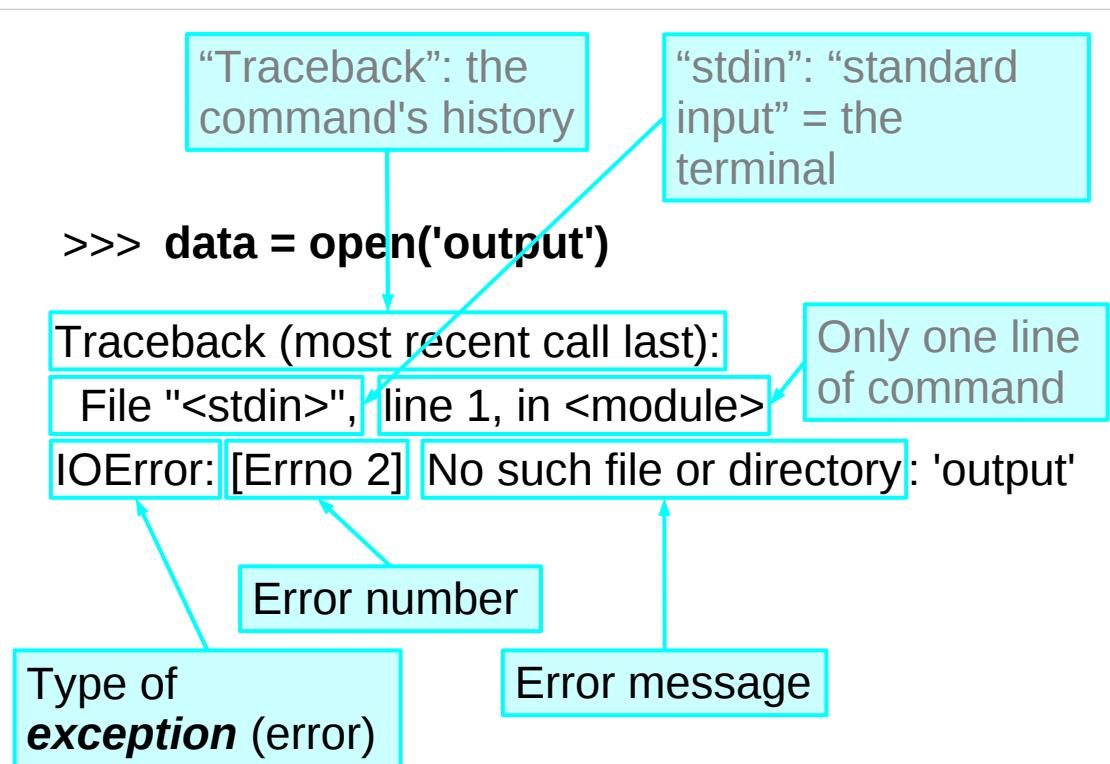
data.txt

30

What happens if we try to open a file that doesn't exist? Or what if something happens to the file whilst we're reading it? – perhaps it is on a network filesystem and there's a network problem, or maybe there's a bad sector on the disk where the file is stored.

Dealing with files is one of the places where we are most likely to encounter *unexpected* errors that are beyond our control or impossible to predict in advance (or avoid). So how can we handle such unexpected errors?

(Note that the errors shown above are *examples* of what *could* happen while you are performing, or attempting to perform, file I/O operations in Python. You won't necessarily get those results if you try the Python commands above now in class. If you do open the file **data.txt** make sure you **close()** it before continuing.)



31

Errors in Python are called *exceptions*. When something goes wrong, Python will “raise an exception”, i.e. generate an error. It is up to your script to *handle* (deal with) that error. If your script does not have a mechanism for handling the exception, then Python’s default exception handler is used, which normally halts your script and prints out some error messages similar to those above.

The first line of the error message above declares that what we are seeing is a “traceback” with the “most recent call last”. A traceback is the command’s history: how we got to be here making this mistake, if you like. The error itself will come at the end. It will be preceded with how we got to be there. (If our script has jumped through several hoops to get there we will see a list of the hoops.) In the above example the error occurs as soon as we try to open the file, so the traceback is pretty trivial.

The next two lines are the traceback. In more complex examples there would be more than two but they would still have the general structure of a “file” line followed by “what happened” line.

The file line says that the error occurred in a file called “`<stdin>`”. What this actually means is that the error occurred on Python being fed to the interpreter from “standard input”. Standard input means our terminal. We typed the erroneous line and so the error came from us.

Each line at the “`>>>`” prompt is processed before the prompt comes back. Each line counts as “line 1”.

If the error had come from a script we would have got the file name instead of “`<stdin>`” and the line number in the script.

The “`<module>`” refers to what function (and module) we were in. This command wasn’t in a function (or a module), so we get “`<module>`” as the function name (indicating we weren’t in a function, or a module for that matter).

The third line gives information about the error itself. First comes a description of what type of error (*exception*) has occurred. This is followed by a more detailed error message – some errors (such as this one) also have an error number. If the error had come from a script the line of Python that generated this error would also be reproduced here.

# Exception handling

```
try:  
    Python commands  
except:  
    Exception handler
```

Python exception handling:

0. **try** some commands
1. if there's an error...
2. ...execute the **except** block...
3. ...but if there's no error, don't execute the **except** block.

(Similar to **if...else** statements)

32

Exception handling in Python is done using the **try...except** construct. Essentially, whenever I think that some commands may fail, I create an exception handler for the errors I expect using the **try...except** construct. If I don't specify a specific exception (error) to handle, then my exception handler is used for *any* errors that occur while executing those commands. If my exception handler only handles certain exceptions and my script produces an exception that my handler wasn't written to deal with, Python's default exception handler will handle that error for me.

Using the **try...except** construct will become clearer as we do some examples.

You can get a list of all the built-in exceptions that Python knows about in the Python documentation here:

<http://docs.python.org/library/exceptions.html>

Note that those are the exceptions that are part of Python itself – however, it is also possible to define new exceptions (these are known as “user-defined exceptions”), and many Python modules do this. If a module defines a new exception, it should document this in its documentation, and, if it is a well written module, in its doc string.

```

def file2dict(filename):
    import sys
    dict={}
    try:
        data = open(filename)
        for line in data:
            [ key, value ] = line.split()
            dict[key] = value
        data.close()
    except IOError:
        print "Problem with file %s" % filename
        print "Aborting!"
        data.close()
        sys.exit(1)
    return dict

```

utils.py

33

Modify the **file2dict()** function as shown above. *Note that the commands in the **try** and **except** blocks have to be indented.*

As already mentioned, when dealing with files, it is very important that you **close()** any files you were using when you are finished with them. This is especially true if something has gone wrong. So, if we run into problems while reading the file, we print an error message and close the file (or try to). Since we don't really know what it would be safe to do at this point, we just make whatever script called us exit with an error. Note that although this may seem a draconian response, it is no more than what Python would do anyway if we didn't handle this error.

Apart from the **try...except** construct, you should be familiar with all the Python in the function above with the exception of the **exit()** function.

The **exit()** function lives in the **sys** module – which is why we have to put “**import sys**” at the start of the **file2dict()** function – and causes your script to stop what it is doing and return to the operating system (or whatever program called it). If you give the **exit()** function an integer as input, then that integer will be the *exit status* of the program. If you don't supply an integer, then the **exit()** function behaves as though it had been called with the integer 0 (i.e. the exit status will be 0). By convention, the exit status of a program or script should be 0 if it completed successfully, and non-zero if it didn't. If we get to the **except** block then there's been a problem, so we should exit with a non-zero exit status.

If you are unfamiliar with the exit status concept (also called *exit code*, *return code*, *return status*, *error code*, *error status*, *errorlevel* or *error level*), the following Wikipedia article gives a bit more detail:

[http://en.wikipedia.org/wiki/Exit\\_status](http://en.wikipedia.org/wiki/Exit_status)

Apart from the **exit()** function and the **try...except** construct, if there is any Python above which you don't understand please put up your hand now and ask the course giver to explain.

Don't forget to save the file after you've finished it or your changes won't take effect.

```
>>> import utils
>>> mydict = utils.file2dict('output')
Problem with file output
Aborting!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "utils.py", line 110, in file2dict
      data.close()
UnboundLocalError: local variable 'data' referenced before assignment
>>>
```

34

Well, that didn't work quite as expected.

Observe that the error is in our **except** block, and so Python's default exception handler takes over to handle it (well, an **except** block can hardly be expected to handle an error within itself).

The problem here is that, since the file doesn't exist, the variable **data** never gets assigned a value, and so it doesn't exist when we reference it to **close()** the file in our **except** block. The easiest way of fixing this problem is to assign **data** a value before we enter the **try** block. What value should we use? Well, it doesn't really matter so long as it is not another **file** object. We'll use the value **None** that we met earlier, as it is a Python special value designed for such purposes. Many Python functions use it to mean that there are no appropriate values for whatever they were asked to do.

So let's fix this error and try again.

```
def file2dict(filename):
    import sys
    dict={}
    data = None
    try:
        data = open(filename)
        for line in data:
            [ key, value ] = line.split()
            dict[key] = value
        data.close()
    except IOError:
        print "Problem with file %s" % filename
        print "Aborting!"
    if type(data) == file:
        data.close()
        sys.exit(1)
    return dict
```

utils.py

35

Modify the **file2dict()** function as shown above. *Remember to indent the “**data.close()**” line for the **if** statement.*

You should be able to see why the above modifications will fix the error we encountered before. If you are confused, please ask the course giver to explain.

Don’t forget to save the file after you’ve finished it or your changes won’t take effect.

```
>>> import utils  
>>> mydict = utils.file2dict('output')  
Problem with file output  
Aborting!  
$
```

36

Note that the `exit()` function not only causes our function to stop running, it throws us out of the Python interpreter as well.

Now, our error message is a little vague (“Problem with file”) and it would be nice if we could be a bit more specific. Recall that Python’s default exception handler not only tells you the type of error but gives you some detail in the form of an “error message”. How can we get access to this?

```

def file2dict(filename):
    import sys
    dict={}
    data = None
    try:
        data = open(filename)
        for line in data:
            [ key, value ] = line.split()
            dict[key] = value
        data.close()
    except IOError, error:
        (errno, errdetails) = error
        print "Problem with file %s: %s" % (filename, errdetails)
        print "Aborting!"
        if type(data) == file:
            data.close()
        sys.exit(1)
    return dict

```

utils.py

37

Modify the **file2dict()** function as shown above.

Don't forget to save the file after you've finished it or your changes won't take effect.

As well as specifying what type of exception we want to handle, we can also get the error information produced by Python put into a variable of our choice. We do this by specifying the name of our chosen variable, separated by a comma, after the type of exception we want to handle. So a more complete syntax for the **try...except** construct is:

```

try:
    commands
except ExceptionType, errorvariable:
    exception handler commands

```

where **ExceptionType** is the type of exception we want to handle (**IOError** is the type of exception raised if there is a problem with file I/O) and **errorvariable** is the optional variable in which we want information about the error to be placed.

The information about the error may actually contain several pieces of information: the error number, the error message, etc. If so, we would normally want to use these separately, so we would “unpack” this variable using a tuple of variables to hold its constituent parts (recall that we can do exactly this sort of “unpacking” for lists or tuples to get the values in the list or tuple into individual variables). The way the error information is stored for errors that result in the **IOError** exception being raised is that the first value is the error number and the second is the actual error message.

The Python documentation gives some details about the error information available with the different sorts of exception:

<http://docs.python.org/library/exceptions.html>

```
>>> import utils  
>>> mydict = utils.file2dict('output')  
Problem with file output: No such file or directory  
Aborting!  
$
```

38

Now we have a much better exception handler. It tells us quite specifically what the problem is, and then **exits** our program.

We now have a reasonably well-written Python function that reads a dictionary from a file, exiting with an informative error message if something goes wrong with the file I/O. Can we make improve this function's error handling even further?

```
>>> line = "Too many values"
>>> [ key, value ] = line.split()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> line = "notenough!"
>>> [ key, value ] = line.split()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 1 value to unpack
>>>
```

39

What happens if the data in the file isn't in the format we were expecting?

Using the Python interpreter, we can simulate this error quite easily.

The variable **line** gets set to the next line of data in the file we are reading. We then use **split()** on this variable, which returns a list of *words* in the line we've just read, i.e. a list of collections of characters separated by *whitespace* (spaces, tabs, etc). We then “unpack” this list into two variables, **key** and **value**. If the list does not contain **exactly** two items (i.e. if line does not contain exactly two “words”), then Python will complain.

We can simulate such errors by setting **line** to a string that contains more than two words, and one that contains fewer than two words, and then using **split()** and trying to unpack the resulting list, exactly as we do in our function. If we do this, we see that the exception that Python raises (in both error cases) is a **ValueError** exception. (Note that the information about the error that we get with a **ValueError** exception is just the error message, there is *no* error number as there was with **IOError** exceptions.)

So, perhaps we could modify our function to handle this type of exception?

# The plan

0. Find the part of the function where the **ValueError** may occur.

1. Place it in a **try...except** construct:

1a. **try** block contains statement(s) that may fail.

1b. **except ExceptionType** block says what to do on failure:

i). print a message saying what has gone wrong.

ii). Set the dictionary to **None**.

iii). Stop processing the file.

40

This is the approach we will take.

We first identify the part of the function where the exception we intend to handle (the **ValueError** exception) occurs.

Then we surround this part of the function with a **try...except** construct.

We have to decide what we want to do on failure. The sensible thing is to tell the user that there is something wrong with the file. We'll use a **print** statement for that.

We have two options at this point: we can either halt the Python program using the **exit()** function, as we have done before, or we can carry on.

Arguably, a file that is in the wrong format is not so serious an error that we should automatically force the program to quit. However, we don't want to return a garbled or incomplete dictionary either. The sensible thing would be to return something that couldn't possibly be a valid dictionary (the special **None** value, for instance). The program or user who called our function can then look at what they got back and, if it is not a dictionary, then *they* can decide whether to quit or do something else (for example, they could try reading from a different file).

We then need to stop processing the file (well, we could continue, but there would be no point reading any more of the file as at this point we've decided to return **None** anyway). In this function we read and process data from the file in a **for** loop, so we need a way of telling Python to quit the **for** loop. (Recall that the Python statement that forces a **for** (or **while**) loop to terminate is the **break** statement.)

# Exercise

Add exception handling to the **file2dict()** function for the **ValueError** exception.

```
def file2dict(filename):  
  
    try:  
        statement(s) from original function  
    except ...  
        print ...  
        ...  
    break      utils.py
```

41

Now I want you to modify the **file2dict()** function in the **utils.py** file in your course home directory to add exception handling for the **ValueError** exception.

In case you get stuck, here are a couple of hints/pointers:

- Remember that the statement(s) from the original function that you are putting into the **try** block need to be indented!
- Remember to specify the type of exception your **except** block should handle (if you don't it will try to handle *all* exceptions).
- If you've positioned your exception handler properly within the function then you don't need to worry about closing the file, since your exception handling for this exception should occur before the file would normally be closed. Thus you don't need to worry about closing the file in your exception handler as this will happen anyway.
- The last line of your **except** block will be a **break** statement.

If (and only if!) you really can't manage it take a look at the page after next.

After you've done this exercise take a short break (i.e. **stop** using the computer) and then we'll continue.

# This page intentionally left blank

Deze bladzijde werd met opzet blanco gelaten.

このページは計画的にブランクを残している

Ta strona jest celowo pusta.

Esta página ha sido expresamente dejada en blanco.

Эта страница нарочно оставлена пустой.

Denne side med vilje efterladt tom.

Pañon intence vaka.

این صفحه خالی است

An leathanach seo fágtha folamh in aon turas.

42

This page intentionally left blank: nothing to see here. If you're stuck for an answer to the exercise, have a look at the next page.

```

def file2dict(filename):
    Beginning of function unchanged

    for line in data:

        try:
            [ key, value ] = line.split()
        except ValueError:
            print "File %s is not in the correct format." % filename
            dict = None
            break

        dict[key] = value

    data.close()

Rest of function unchanged

```

utils.py

43

You should have modified the `file2dict()` function in a similar way to the modifications shown above.

(Don't forget to save the file after you've finished it or your changes won't take effect.)

There are two points worth mentioning regarding the above exception handler. Firstly, note that we not only tell the user what the problem is, but we tell them what file has caused the problem – this will help them track down the problem. Secondly, note that if our exception handler for the `ValueError` exception is called we will exit the `for` loop and the file will then be closed, thus we don't need to worry about closing it in the exception handler.

You can now try this function on two files in your course home directory that aren't in the correct format: `bad-format1.txt` and `bad-format2.txt`.

```

>>> import utils
>>> dict1 = utils.file2dict('bad-format1.txt')
File bad-format1.txt is not in the correct format.
>>> print dict1
None
>>> dict2 = utils.file2dict('bad-format2.txt')
File bad-format2.txt is not in the correct format.
>>> print dict2
None

```

If you had problems with the exercise, or if you don't understand the Python above, please let the course giver know.

# Handling multiple exceptions

**try:**

*Python commands*

**except *Exception1*:**

*Exception handler1*

**except *Exception2*:**

*Exception handler2*

...

**except:**

*Handler for all other exceptions*

0. **try** some commands
1. if there's an error...
2. ...examine the **except** blocks...
3. ...if the error is **Exception1** use that **except** block...
4. ...if it's **Exception2** use that **except** block...
5. ...and so on...
6. ...if it's not any of the listed exceptions, use the final **except:** block if it exists.

If you need to handle multiple exceptions in the same block of code, you simply list each of the exception handlers in separate **except** blocks one after the other with each **except** block indicating what type of exception it is supposed to handle in its **except** statement. You can have as many of these **except** blocks as you want.

After you've finished defining exception handlers for specific exceptions, you can then have a final **except** block which will handle any other exceptions not previously handled – this **except** block will be written just as “**except:**” since it is supposed to handle multiple types of exception.

For example, consider the code snippet below:

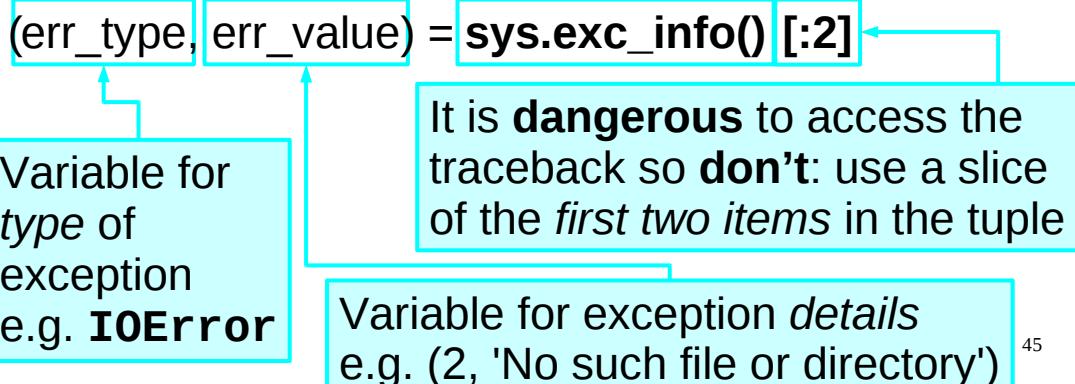
```
try:  
    my_function(data)  
except IOError:  
    print "There was an I/O error."  
except ValueError:  
    print "There was something wrong with a value."  
except:  
    print "There was some other error!"
```

This code snippet will call the **my\_function()** function with **data** as its argument. If an **IOError** exception occurs, it will print “There was an I/O error.”. If a **ValueError** exception occurs, it will print “There was something wrong with a value.”. If any other type of exception occurs it will print “There was some other error!”.

# Exception handling: `exc_info()`

```
import sys
```

`exc_info()` returns a *tuple* of three items of information about the current exception:  
(*ExceptionType*, *ExceptionDetails*, *Traceback*)



45

The `exc_info()` function (which lives in the `sys` module) returns **exception information**. It returns a tuple of three items relating to the current exception. (If no exception has been raised then it returns the tuple (`None`, `None`, `None`)).

The first item is the *type* of the exception, e.g. `IOError`, `ValueError`, etc.

The second item contains the exception *details*, e.g. for an `IOError` exception it might contain the tuple (2, 'No such file or directory'). These details are the information about the error that we've accessed earlier using a `try...except` construct for a specific type of exception.

The third item contains the *traceback*, which is the listing of the lines of our script that have gotten us to this error. We've seen Python display the traceback before when we've made errors, but we have not tried to access it. In fact, it can be quite dangerous to interfere with the traceback, and, in particular, **assigning it to a local variable in a function will cause a circular reference**. Therefore it is best not even to access this item. Consequently you normally call the `exc_info()` function like this:

```
sys.exc_info()[:2]
```

The “[`:2`]” tells Python to take a slice of the first two items of the tuple returned by the `exc_info()` function, i.e. (the type of the exception, the exception details), and so it ignores the traceback.

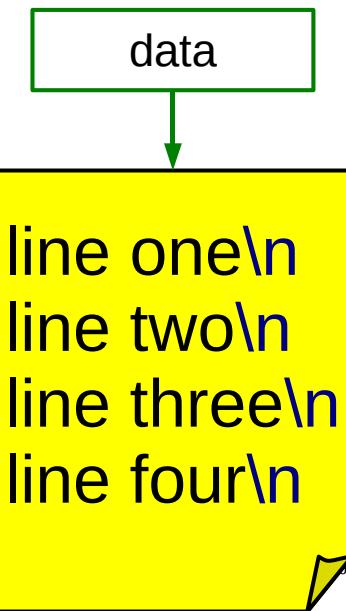
Given that we already know how to get the exception details when handling a specific type of exception, why might we want to use the `exc_info()` function?

Recall that we can have an exception handler that handles *all* types of exception. Inside such an exception handler, we might want to know what type of exception we were handling, as well as the details of the exception. In such circumstances the best way to get this information is with the `exc_info()` function.

# Moving around a file

```
>>> data = open('data.txt')
```

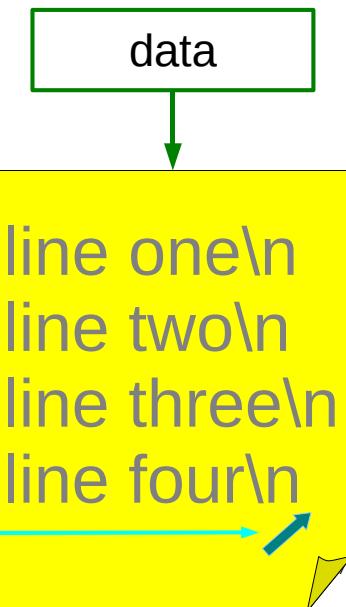
position:  
start of file



As we know, when we **open()** a file, we start at the beginning of the file: the offset of our file object is set to the start of the file (unsurprisingly, as we will see in a moment, this position is offset **0** of the file).

```
>>> data = open('data.txt')  
  
>>> data.readlines()  
['line one\n', 'line two\n', 'line three\n', 'line four\n']  
  
>>> data.readlines()  
[]  
  
>>> data.readline()  
''
```

position:  
at end of file



We can move to the end of the file by reading all the data in it using the `readlines()` method. Once we are at the end, if we try to use the `readlines()` or `readline()` methods we don't get any more data, we just get an empty list or an empty string (respectively).

Suppose we want to read from somewhere else in the file, how can we do that? We could always close the file and then open it again, but that seems a bit silly – is there a better way?

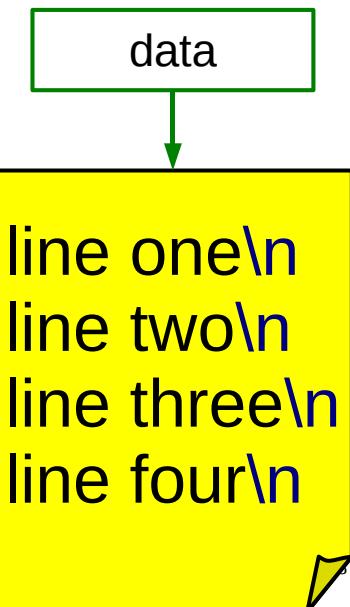
# Moving to the start of a file

```
>>> data = open('data.txt')  
>>> data.readlines()  
['line one\n', 'line two\n', 'line three\n', 'line four\n']
```

```
>>> data.seek(0)
```

offset in file

position:  
start of file



The better way is to use the `seek()` method. The `seek()` method is a method that moves the offset of the `file` object to the specified location in the file without reading (or writing) any data. You need to be very careful with this method, as it is very easy to accidentally move yourself to a random position in a file, which can then play havoc with subsequent read or write operations.

As we see here, the start of the file has an offset of 0, and so, to move to the start of a file we would call the `seek()` method with an argument of `0`, as shown above.

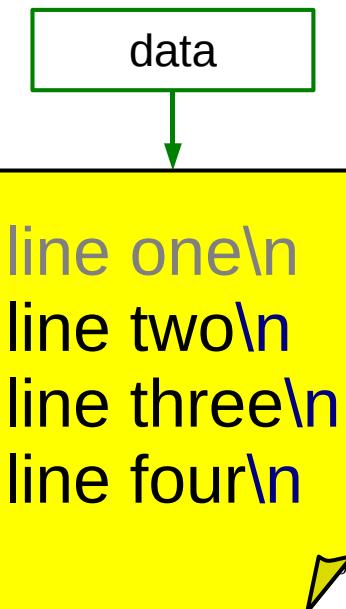
You can find out what the current offset of the `file` object is using the `tell()` method, as we will see shortly.

**A very important point to note is that not all offsets you can give the `seek()` method are valid!** This is one reason why the `seek()` method is rarely used except to move to the beginning or an end of a file (and even that is relatively uncommon). (In case you are curious, what offsets are valid depends on the type of file (whether it is text or binary) and the operating system that Python is running on. In this course we are only going to deal with text files (although we'll mention binary files briefly a little later), and we're not going to use the `seek()` method except to show you how to move to the start and end of a file.)

There are also certain sorts of `file` object that do not support the `seek()` method – however, if you are reading from a normal file that is stored on a file system somewhere, the `seek()` method should work.

```
>>> data = open('data.txt')  
  
>>> data.readlines()  
['line one\n', 'line two\n', 'line three\n', 'line four\n']  
  
>>> data.seek(0)  
  
>>> data.readline()  
'line one\n'
```

position:  
after end of first line,  
at start of second line



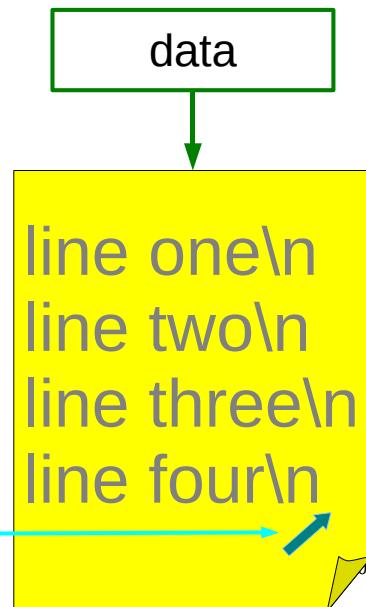
We can verify that we are indeed at the start of the file by trying to read a line from it. As we see, we indeed at the start of the file, and have now moved to the start of the second line of the file.

# Moving to the end of a file

```
>>> data = open('data.txt')  
>>> data.readline()  
'line one\n'  
>>> data.seek(0, 2)
```

specifies that offset is  
***relative to the end of the file***

position:  
at end of file



We can move to the end of the file using the **seek()** method as well. If we know exactly what value corresponded to the offset of the end of the file, we could instruct the **seek()** method to move to that offset, but that depends on us knowing exactly what the offset of the end of the file is.

So the **seek()** method can be instructed to move to an offset ***relative*** to the end of the file (or relative to the current position in the file). The syntax is as follows:

**seek(offset) or seek(offset, 0)**

move to the specified **offset**  
(*absolute* position, i.e. relative  
to the start of the file)

**seek(offset, 1)**

move to the specified **offset**  
**relative to the current**  
**position in the file**

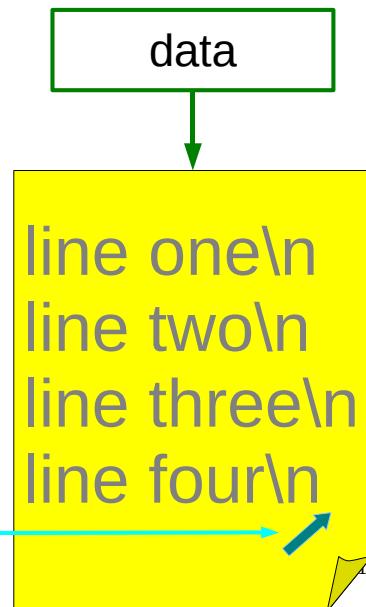
**seek(offset, 2)**

move to the specified **offset**  
**relative to the end of the file**

So to move to the end of the file, we call the **seek()** method with the arguments **0, 2**. **0** for the offset, and **2** to signify that this offset is relative to the end of the file.

```
>>> data = open('data.txt')  
>>> data.readline()  
'line one\n'  
>>> data.seek(0, 2)  
>>> data.readline()  
''
```

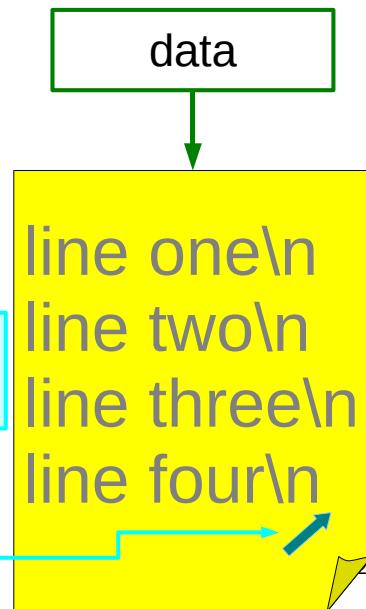
position:  
at end of file



We can verify that we are at the end of the file by trying to use the **readline()** method (or the **readlines()** method, if you prefer). As we are at the end of the file, if we try to use the **readline()** or **readlines()** methods we don't get any more data, we just get an empty string or an empty list (respectively).

# Finding your position in a file

```
>>> data = open('data.txt')  
>>> data.readline()  
'line one\n'  
>>> data.seek(0, 2)  
>>> data.tell() current offset as a  
39L long integer  
>>> data.close() position:  
at end of file
```



We can determine the value of the offset of a **file** object (i.e. our position within the file) using the **tell()** method. The offset is an integer, and because files can be very large, it is a *long integer* (in Python long integers can be of arbitrary size, limited only by the amount of memory the computer possesses), hence the ‘L’ that is printed after the ‘39’ above.

(Note that the **tell()** method only returns *valid* offsets, *except* when you have opened a Unix text file as a text file on the Windows platform, when it is **unreliable** – the solution to this is to open Unix text files as binary files on the Windows platform (we’ll see how to open a file in binary mode later). However, apart from in this pathological case, any offset returned by the **tell()** method can be used as the offset argument for the **seek()** method.)

We’ve finished playing around with this file now, so make sure you **close()** it. If you’re being good you get rid of the **data** variable as well:

```
>>> del data
```

# What about output?

```
input = open('input.txt')
```

)

equivalent

```
input = open('input.txt',
```

'r')

open for  
reading

```
output = open('output.txt',
```

'w')

open for  
writing

53

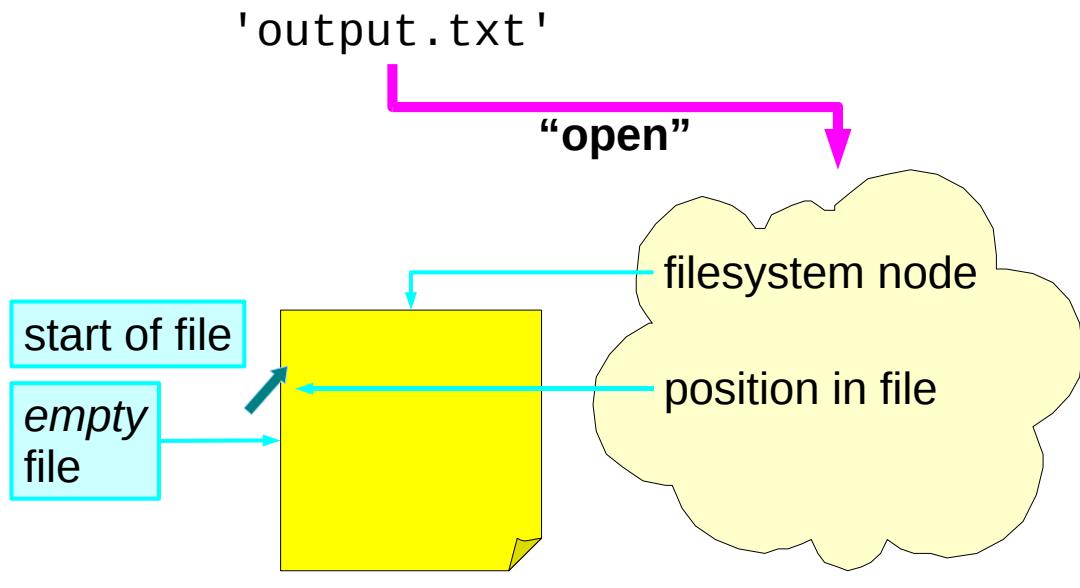
To date we have been only reading from files. What happens if we want to write to them?

The **open()** function we have been using actually takes more than one argument. The second argument specifies the *mode* in which we want to open the file. Amongst other things, the mode specified whether we want to read or write the file. If the mode is not specified, then the default is to open the file for reading only.

The explicit value you need to open a file for **reading** is the single letter string '**r**'. That's the default value that the system uses. The value we need to use to open a file for **writing** is '**w**'.

There are other modes apart from the above two simple ones, some of which we'll meet later.

# Opening a file for writing



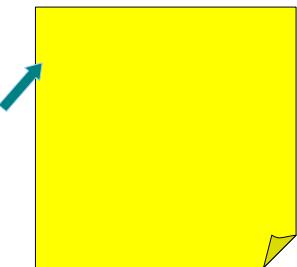
54

As ever, a newly opened **file** has its position pointer (“offset”) pointing to the start of the file. This time, however, the file is empty. **If the file previously had any content then it would get completely replaced.**

```
>>> output = open('output.txt', 'w')
```

file name

open for  
writing



55

Apart from the explicit second argument, the **open()** function is used exactly as we did before.

```
>>> output = open('output.txt', 'w')
```

```
>>> output.write('alpha\n')
```

method to write  
a lump of data

lump of data  
to be written

lump: not  
necessarily  
a whole line

alpha\n

56

Now that we've opened our file ready to be written to we had better write something to it. There is no “**writeline()**” equivalent to **readline()**. What there is is a method “**write()**” which might be thought of as “writelump()”. It will write into the file whatever string it is given whether or not that happens to be a line.

When we are writing text files it tends to be used to write a line at a time, but this is not a requirement.

```
>>> output = open('output.txt', 'w')  
>>> output.write('alpha\n')  
>>> output.write('bet')
```

*lump of data  
to be written*



57

For example, we could write three characters in one `write()`...

```
>>> output = open('output.txt', 'w')  
>>> output.write('alpha\n')  
>>> output.write('bet')  
>>> output.write('a\n')
```

remainder  
of the line

alpha\n  
beta\n

58

...and the last two characters (the new line counts as one character) in a second **write()**.

```
>>> output = open('output.txt', 'w')  
>>> output.write('alpha\n')  
>>> output.write('beta')  
>>> output.write('a\n')  
>>> output.writelines(['gamma\n', 'delta\n'])
```

method to write  
a list of lumps

the list of lumps  
(typically lines)

alpha\n  
beta\n  
gamma\n  
delta\n

There is a writing equivalent of **readlines()** too: “**writelines()**”. Again, the items in the list to be written do not need to be whole lines.

```
>>> output = open('output.txt', 'w')  
>>> output.write('alpha\n')  
>>> output.write('bet')  
>>> output.write('a\n')  
>>> output.writelines(['gamma\n', 'delta\n'])  
>>> output.close()
```

Python is done  
with this file.

Only at this point is it  
guaranteed that the  
data is on the disc!

60

Closing the file is particularly important with files opened for writing. As an optimisation, the operating system does not write data directly to disc because lots of small writes are very inefficient and this slows down the whole process. When a file is closed, however, any pending data is “flushed” to the file on disc. This makes it particularly important that files opened for writing are closed again once finished with.

**It is only when a file is closed that the writes to it are committed to the file system.**

# Exercise

In `utils.py`, write a function called `dict2file()` that takes a dictionary and an *optional* filename as its arguments and writes the contents of the dictionary to the file:

1. Open a file for writing.
2. For each key in the dictionary:
  - 2a. Write the key to the file.
  - 2b. Write a tab ('\t') to the file.
  - 2c. Write the value corresponding to the key, followed by a newline ('\n'), to the file.
3. Close the file.

61

So now let's try creating a function that writes to a file. You should create your function in the `utils.py` file in your course home directory. We have covered all the Python I/O you need to write this function (*hint*: you can use either the `write()` method or the `writelines()` method (or both if you're particularly creative)). And you should already know the basic Python you'll need (*hint*: recall that if you treat a dictionary like a list (say in a `for` loop) it behaves like a list of its *keys*).

**Make sure you put in exception handling for any `IOError` exceptions that may be raised when writing to the file. Your function should also use a *default* filename (i.e. a filename to use if the user does not specify one when they call the function) of `dictionary.txt`.**

Once you've finished writing this function (make sure you saved the file) you can test it out on the dictionary of atomic symbols and element names in the file `chemicals.txt` in your course home directory:

```
>>> import utils  
>>> chemicals = utils.file2dict('chemicals.txt')  
>>> utils.dict2file(chemicals)
```

Now exit the Python interpreter, and have a look at the file `dictionary.txt`:

```
$ more dictionary.txt  
Ru      ruthenium  
Re      rhenium  
Ra      radium  
Rb      rubidium
```

(For space reasons, only an excerpt of the `dictionary.txt` file is shown above.)

If – and only if – you really get stuck, have a peek at the answer on the page after next.

When you've finished make sure and take at least a 5 minute break (preferably at least a 10 minute break) – and that means a *break* from the computer, not checking your e-mail.

# This page intentionally left blank

Deze bladzijde werd met opzet blanco gelaten.

このページは計画的にブランクを残している

Ta strona jest celowo pusta.

Esta página ha sido expresamente dejada en blanco.

Эта страница нарочно оставлена пустой.

Denne side med vilje efterladt tom.

Pañon intence vaka.

این صفحه خالی است

An leathanach seo fágtha folamh in aon turas.

62

This page intentionally left blank: nothing to see here. If you're stuck for an answer to the exercise, have a look at the next page.

```

def dict2file(dict, filename='dictionary.txt'):
    import sys
    data = None
    try:
        data = open(filename, 'w')
        for key in dict:
            output = "%s\t%s\n" % (key, dict[key])
            data.write(output)
        data.close()
    except IOError, error:
        (errno, errdetails) = error
        print "Problem with file %s: %s" % (filename, errdetails)
        print "Aborting!"
        if type(data) == file:
            data.close()
        sys.exit(1)
    return

```

utils.py

63

Your **dict2file()** function should look similar to the one above. If it doesn't, or if you had problems with the exercise, please let the course giver know.

Also, if there is anything in the above function that you don't understand please ask the course giver.

Note that in my version of the function above I only use the **write()** method once as I process each key, value pair from the dictionary. This makes my code fairly compact, but not as readable as it might be. It is perfectly acceptable to use the **write()** method multiple times, e.g.

```

data.write(key)
data.write('\t')
data.write(dict[key])
data.write('\n')

```

Note that one problem with the above sequence of Python code is that there is no guarantee that either the key or the value is a string and the **write()** method *only* accepts a string as an argument. This is why it is better to use the string formatting operator (%) as I do here:

```
output = "%s\t%s\n" % (key, dict[key])
```

and then use the **write()** method like this:

```
data.write(output)
```

The use of the string formatting operator guarantees that **output** will be a string (unless, of course, there is something wrong with **key** or **dict[key]**, say if **dict** is not actually a dictionary, in which case the function will fail anyway). An alternative would be to use **str()** function to convert the key and value to strings before giving them to the **write()** method.

# Checking whether a file exists

```
>>> import os.path  
>>> os.path.exists('chemicals.txt')  
True  
>>> os.path.exists('rubbish.txt')  
False  
>>>
```

64

Those of you who have ever accidentally overwritten a file may have spotted the glaring flaw with our otherwise well-behaved `dict2file()` function: it happily overwrites (or attempts to overwrite) whatever file it is given as a file name (or `dictionary.txt` if it is not given a file name).

Clearly, this is dangerous and we should fix this glaring bug. How do we do this? We need to check whether the file already exists before we try to write to it.

The `exists()` function – which lives in the `os.path` module – returns `True` if the specified file exists, `False` if it doesn't (or if the specified file is a broken *symbolic link*).

(A symbolic link (also known as a *symlink* or a *soft link*) is similar to a shortcut in the Microsoft Windows operating system (if you are familiar with those) – essentially, a symbolic link points to another file elsewhere on the system. When you try and access the contents of a symbolic link, you actually get the contents of the file to which that symbolic link points. If the file to which the symbolic link points does not exist, then the symbolic link is said to be *broken*. For a more detailed explanation of symbolic links see the following Wikipedia article:

[http://en.wikipedia.org/wiki/Symbolic\\_link](http://en.wikipedia.org/wiki/Symbolic_link)

)

```
def dict2file(dict, filename='dictionary.txt'):
    import sys, os.path

    if os.path.exists(filename):
        print "File %s exists." % filename
        print "Not overwriting it."
        return

    data = None
```

*Rest of function unchanged*

utils.py

65

Modify the **dict2file()** function as shown above.

Remember to save the file after you've made your modifications.

Now the first thing our function does is to see whether the file name it has been given is of a file that already exists. If it does, it prints out a message to that effect, says that it is not overwriting the file, and simply **returns** without doing anything.

If we wanted to, we could make the function cause our program to **exit()**, but that seems an overly draconian response – it makes more sense to just let the user know that we haven't overwritten the file and let the program continue.

Of course it would be even better if the *program* (and not just the user) could tell whether the function had succeeded or not. To do that we need to make our function do something that a program calling it could easily check to see whether the function had succeeded or not.

```
def dict2file(dict, filename='dictionary.txt'):
    import sys, os.path

    if os.path.exists(filename):
        print "File %s exists." % filename
        print "Not overwriting it."
        return False

    data = None

Rest of function unchanged except last line:

    return True
```

utils.py

66

Modify the **dict2file()** function as shown above.

Remember to save the file after you've made your modifications.

Now, if the passed file name is of a file that already exists, our function returns the Boolean value **False**, as well as printing out a message for the user. If the function succeeds, it returns the Boolean value **True**. A well-written program can now test whether the function succeeded or not and behave accordingly.

We can now try out our greatly improved function:

```
>>> import utils
>>> chemicals = utils.file2dict('chemicals.txt')
>>> writeout = utils.dict2file(chemicals, 'dict1.txt')
>>> print writeout
True
>>> writeout = utils.dict2file(chemicals, 'dict1.txt')
File dict1.txt exists.
Not overwriting it.
>>> print writeout
False
```

# Renaming a file

```
>>> import os.path  
>>> os.path.exists('data1.txt')  
True
```

**rename( )** renames files.  
It lives in the **os** module.

```
>>> import os  
>>> os.rename('data1.txt', 'data2.txt')
```

Under Unix/Linux if the new name is a file that already exists, then that file is **deleted**, i.e. **rename( )** behaves like the Unix **mv** command.

```
>>> import os.path  
>>> os.path.exists('data1.txt')  
False
```

67

So what do you do if there already exists a file with the name you want to use? You could use a different name for your file, or you could rename the existing file to avoid overwriting it.

You can rename a file (or directory) using the **rename( )** function that lives in the **os** module.

**IMPORTANT:** If the new name of the file or directory you want to rename is a file that already exists, then under Unix/Linux that file will be **deleted** and then the renaming will be done. If the new name is the name of an existing directory, then an **OSError** will be raised. Under Windows, if the new name is an existing file or directory then an **OSError** will be raised.

Under Unix/Linux **rename( )** basically behaves like Unix's **mv** command (which means that on Unix/Linux you can use **rename( )** to **move** files around, not just rename them in the same directory). Note though, that on some versions of Unix **rename( )** will fail if the new "name" is on a different file system to the original file, i.e. if, instead of just renaming the file, you use the **rename( )** function to **move** the file to a **different** file system.

# Appending output to a file

```
$ cat output.txt
```

```
alpha  
beta  
gamma  
delta
```

open for  
appending

```
>>> output = open('output.txt', 'a')  
>>> output.write('epsilon\n')  
>>> output.close()  
>>> del output  
>>>
```

```
$ cat output.txt
```

```
alpha  
beta  
gamma  
delta  
epsilon
```

68

So we know how to read from files, and how to write to a file. However, if we write to a file that already exists then we will *overwrite* it. Now suppose we don't want to overwrite the file, but just add some more data to the end of it – how can we do this?

Another value for the mode of the **open()** function that we can use is '**a**'. This means that we should open the file for writing, but we want to *append* our writes to it, i.e. we don't want to destroy the data already in the file, and instead we want to add whatever we write to the end of the file.

Note that on some systems (most commonly some (but not all) versions of Unix) if you open a file in append mode then *all* writes to the file are *always appended* to the end of the file, regardless of the position you may have moved to in the file using **seek()**. This means that if you use **seek()** on a file opened in append mode, you should not rely on it working in the way you might hope.

(Note that in the above slide the **cat** command is given at the Unix/Linux prompt. We then issue some Python commands in the Python interpreter. Finally, we quit the interpreter and issue the **cat** command (at the Unix/Linux prompt) again to see what effect our Python commands have had.)

# Checking whether a file is open

```
>>> data = open('data.txt')
```

```
>>> data.closed
```

```
False
```

the Python file object

a dot

an “attribute”

Boolean indicating  
whether or not the  
file is closed

```
>>> data.close()
```

```
>>> data.closed
```

```
True
```

69

Some of you may be wondering how we can tell whether a file is open or closed.

**file** objects have an attribute called **closed**, which is set to **True** if the file is closed and set to **False** if the file is open. (Attributes are “built in” variables that objects can have.) If we need to tell whether a file is open or closed, we can do so by examining the **closed** attribute of the corresponding **file** object to see whether it is **True** or **False**.

# Accessing binary files

```
input = open('input.dat', 'rb')
```

open for reading  
a binary file

```
input.read(1)
```

read some *bytes* from a file:  
bytes are returned as a **string**

maximum number of bytes  
to read from file: omit to read  
all remaining bytes of file

```
output = open('output.dat', 'wb')
```

open for writing  
in binary mode

70

To date we have only been accessing text files. What about binary files?

The first thing to understand is that many operating systems (such as Unix/Linux) **do not** treat text and binary files differently: so on these platforms we can carry on much as before. Some platforms (such as Windows) do treat text and binary files differently however, and on those platforms it is **very important** to know what sort of file you are dealing with.

Recall that the **open()** function for opening a file takes more than one argument, and the second argument specifies the mode in which we want to open the file. The mode tells Python whether to treat the file as a text file or a binary file. If we don't explicitly say it is a binary file, then the default is to treat the file as a text file.

The explicit value you need to use to open a file in binary mode is the single letter string '**b**' which you add to the end of the file mode. So for **reading** a **binary** file, use '**rb**' and for **writing** in **binary** mode, use '**wb**' (and to **append** to a **binary** file you would use '**ab**').

If you are working with binary files, then the concept of lines probably no longer applies, so we need another method to read data from such files. That method is **read()**. When you use the **read()** method you can specify the *maximum* number of bytes you wish to read – **read()** will return that number of bytes or fewer if it comes to the end of the file. If you don't specify the maximum number of bytes then **read()** will read all the bytes in the file from the current offset in the file to the end of the file. Just as with **readline()** or **readlines()**, using **read()** advances the position in the file (the offset). And also as with **readline()** and **readlines()**, **read()** returns the bytes it reads as a **string**. If your binary file does not contain strings, it is up to you to convert the data in the string returned by **read()** to the correct format.

We already have a method (**write()**) that allows us to write arbitrary length strings, rather than whole lines, to a file, so we don't need a new method for writing to a binary file.

# Reading files from other OSes

```
>>> data = open('Win-text.txt')  
>>> data.readline()  
'line one\r\n'  
>>> data.close()
```

open as a text file in *Universal newline mode* (for reading)

```
>>> data = open('Win-text.txt', 'rU')  
>>> data.readline()  
'line one\n'  
>>> data.close()
```

71

Those of you who frequently work on both Unix/Linux and Windows platforms, or old Macintosh (pre-MacOS X) and Windows platforms, etc will probably have come across the annoying fact that text files are handled differently on all these platforms.

On Unix/Linux systems, each line of text in a text file is terminated with a single line feed character ('\n').

On Windows systems, each line of text in a text file is terminated with a single carriage return character ('\r') followed by a single line feed character ('\n'), i.e. the *two* character combination '\r\n'.

On old Macintosh systems (up to MacOS 9), each line of text in a text file is terminated with a single carriage return character ('\r').

As long as you are always working on the same platform, or always working with binary files, this is irrelevant and you are unlikely to care. However, when working with text files on different platforms, it can be a problem, since some applications will not recognise a file as text if it doesn't have the correct *end-of-line* (EOL) character combination for that platform, or else they may get confused.

Python has a special mode – *universal newline support* – for handling such files. Instead of just opening such files as we have been doing up to now, we use the special mode '**rU**' (or just '**U**') as shown above. In this mode, Python treats any of the EOL combinations as a single newline character, which it represents as '\n'. (Note that the copy of Python you are running needs to have been compiled with universal newline support enabled – this is the default, so normally you shouldn't need to worry about whether or not universal newline support is enabled on your particular copy of Python.) This mode is only for *reading* text files – when Python writes to text files it uses the EOL character combination for the platform on which it is running whenever you specify the '\n' character.

The file `Win-text.txt` in your course home directory is a Windows text file version of the file `data.txt`. As you can see, if we open it normally it doesn't look right: there's an extra '\r' character hanging around near the end of each line. If we open it using Python's universal newline support mode then it behaves "normally".

If you want to know more about line endings for text files on different platforms, see the following Wikipedia article:

<http://en.wikipedia.org/wiki/Newline>

# Accessing files

1. Direct access to files
2. Structured files: **csv** module

72

We've already met the limitations of the **split()** function for handling input, and we've already come across two structured files that we can't read because they aren't in the simple "whitespace" delimited format we're used to. How can we handle such files?

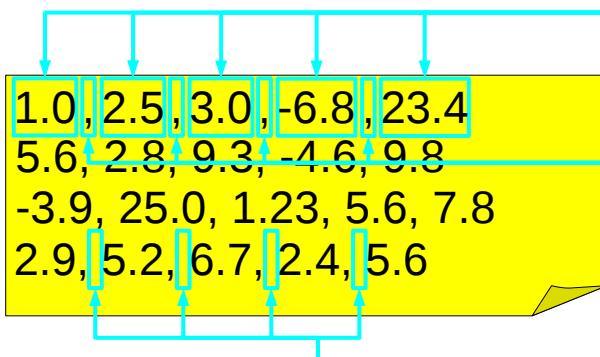
Obviously, we could write our own functions for making sense of (*parsing*) every single file format we came across, but there is an easier way, at least for a large class of text files. We use one of the standard Python modules (introduced in Python 2.3): the **csv** module.

This module allows us to read and write so-called CSV (comma separated value) files. These are files where each line has the same structure, consisting of a number of values (called *fields*) separated by some specified character (or sequence of characters), typically a comma (,). The character (or characters) that separates the fields is called a *delimiter*.

This module also defines its own special sort of exception that is used when something goes wrong with the functions and methods it provides. This exception is "**Error**", but as it is defined in the **csv** module, you would normally refer to it by prefixing it with "**csv.**", e.g.

```
try:  
    ...  
except csv.Error:  
    print "Can't read CSV file!"
```

# CSV files



**Fields** containing data

Fields are separated by **delimiters**.  
A comma (,) is often used as a delimiter.

Sometimes spaces may follow delimiter between fields, or may be used as “padding” to make fields a particular size (*width*).

73

Although CSV stands for “comma separated values”, as far as Python is concerned CSV files are any type of text file which have a particular structure.

For Python, the CSV file will consist of a number of lines, each of which will have a number of items of data (called *fields*), separated from each other by a particular character known as a *delimiter*. (A comma (,) is often used as a delimiter, hence the name “comma separated values”).

There may also be spaces after (or before) the delimiter to separate the fields, or the fields may be “padded” out with spaces to make them all have the same number of characters in them. (The number of characters in a field is called the field’s *width*.)

Some programs that work with CSV files will only accept a comma as a delimiter, or else will only accept a comma, a space or a tab as delimiters. Python, however, will accept any single character as a delimiter.

Also, some programs that work with CSV files require each line of the file to have the same number of fields (although some of the fields may be empty). Python doesn’t care how many fields there are on each line, provided that the same delimiter is used throughout the file.

# Quoting in CSV files

```
"Fred Smith", red, 56.9  
"Joe Bloggs", blue, 27.8  
"Jill East", brown, 28.9
```

Data with spaces may be *quoted* (surrounded by quotation marks).

```
"Smith, Fred", red, 56.9  
"Bloggs, Joe", blue, 27.8  
"East, Jill", brown, 28.9
```

Data containing special characters, e.g. the delimiter, is also quoted.

```
"Fred Smith", "red", "56.9"  
"Joe Bloggs", "blue", "27.8"  
"Jill East", "brown", "28.9"
```

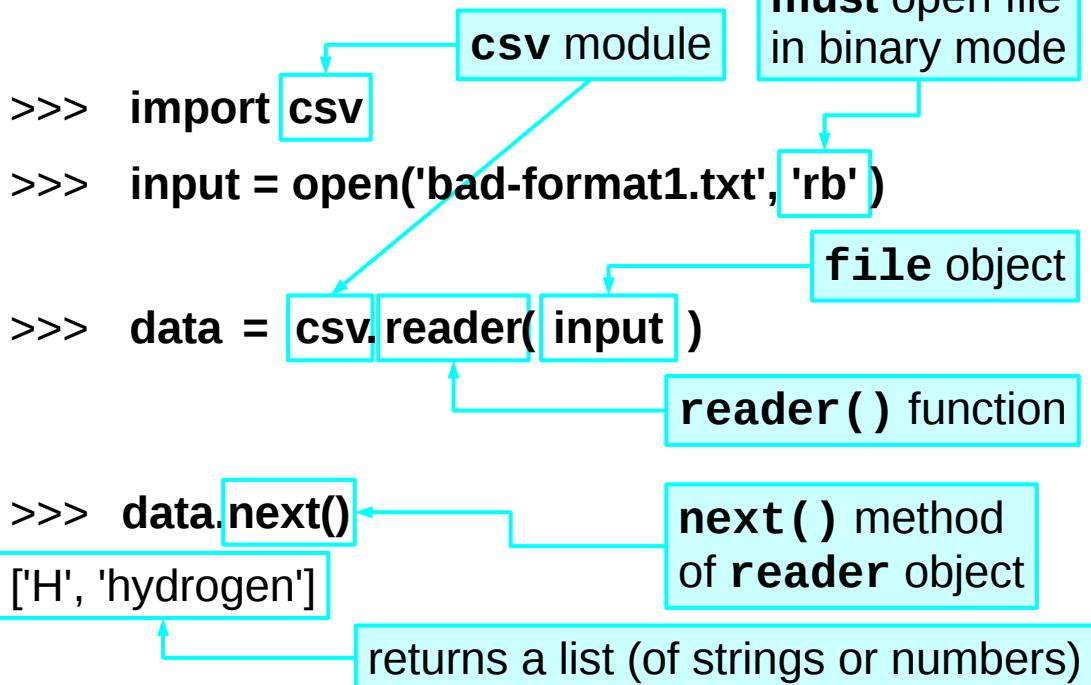
We can even quote *all* data (or all *text* data) if we wish.

In CSV files, data with spaces in it is often *quoted* (surrounded by quotation marks) to make clear that it is one single item of data and should be treated as such. In Python this is not necessary unless you are using a space as your delimiter, but you will often find that programs that produce CSV files automatically quote data with spaces in it.

If your data contains special characters, such as the delimiter or a new line ('\n') character, then you will need to quote that data or Python will get confused when it reads the CSV file.

If you want, you can quote all the data in the file, or all the text data. Python doesn't mind if you quote data even when it is not strictly necessary.

# Reading CSV files



Make sure you close the file (`data.close()`) after you've finished reading from it!

You read a CSV file using the `reader()` function in the `csv` module. This function requires a `file` object as input. **The file object must be opened for reading in binary mode.**

The `reader()` function returns a special sort of object, called a `reader` object. The `reader` object has a `next()` method which reads the next line of the CSV file and returns it as a list of strings or numbers. The first field on the line is the first item in the list, the second field is the second item in the list, and so on.

It is *your* responsibility to delete the `reader` object, close the CSV file and delete its `file` object when you are finished with the file.

The `reader()` function can take a number of optional formatting parameters that specify what sort of CSV file it is reading. See the `csv` module's documentation for a list of these parameters and their default values:

<http://docs.python.org/library/csv.html#csv-fmt-params>

You may also find the examples given in the module's documentation useful:

<http://docs.python.org/library/csv.html#csv-examples>

Treat a **reader** object like a list...

...it behaves like a list of the *lines of the CSV file*, where each line is itself a list (of fields of the CSV file).

```
import csv  
import utils  
  
input = open('bad-format1.txt','rb')  
data = csv.reader(input)  
  
chemicals = {}  
  
for line in data:  
    [ key, value ] = line  
    chemicals[key] = value  
del key, value, line, data  
input.close()  
del input  
  
utils.print_dict(chemicals) 76  csv1.py
```

...so here is a typical example of using the **csv** module to read a CSV file. This script is in the file **csv1.py** in your course home directory.

Recall that in Python very often if you treat an object like a list, it will behave like a list of something, e.g. a dictionary will behave like a list of its keys. If we treat a **reader** object like a list, it behaves like a list of the lines of the CSV file, where each of those lines is itself a list (of strings or numbers). (Of course, just as when we treat a **file** object like a list we move our position in the file forward until we get to the end of the file, the same thing happens when we treat a **reader** object like a list. This means that we can only really treat a **reader** object like a list once, unless we then use the **seek()** method of the underlying **file** object to reset our position in the file.)

The **print\_dict()** function is not a standard Python function. It is a function that prints out the keys and values of a dictionary that we wrote in the “Python: Introduction for Absolute Beginners” and the “Python: Introduction for Programmers” courses. For this course the function has been put in the **utils** module in your course home directories.

Finally, note that the bits of Python that read the CSV file and set up the dictionary would normally be hived off as a separate function in the script – that way we wouldn’t have to worry about deleting the loop variable and all the other temporary variables. I haven’t bothered with structuring this script that way as its purpose here is just to demonstrate the simple use of **reader** objects. Similarly, I haven’t put in any exception handling. If this script was actually meant to be used for any serious task, it would be better structured and would have some exception handling.

# Writing CSV files

```
>>> import csv  
>>> output = open('csv-file.txt', 'wb')  
>>> data = csv.writer(output)  
>>> data.writerow(['H', 'hydrogen'])
```

must open file in binary mode  
file object  
writer() function  
writerow() method of writer object  
list (of strings or numbers)

77

Make **sure** you close the file (`data.close()`) after you've finished writing to it!

**It is only when a file is closed that the writes to it are committed to the file system.**

You write a CSV file using the `writer()` function in the `csv` module. This function requires a `file` object as input. **The file object must be opened for writing in binary mode.**

The `writer()` function returns a special sort of object, called a `writer` object. The `writer` object has a `writerow()` method which writes takes a list of *strings or numbers* and writes them out as a complete line of the CSV file, formatted appropriately. The first item in the list will be the first field of the line that is written to the CSV file, the second item will be the second field of the line, and so on. Python will handle quoting any data that needs to be quoted, you do not have to do it yourself.

Note that it is **your** responsibility to delete the `writer` object, close the CSV file and delete its `file` object when you are finished with the file. It is **only** when you **close** the `file` object that the data you've written to the CSV file will be committed to the file system.

The `writer()` function can take a number of optional formatting parameters that specify what sort of CSV file it is reading (these are the same as the optional formatting parameters for the `reader()` function). See the `csv` module's documentation for a list of these parameters and their default values:

<http://docs.python.org/library/csv.html#csv-fmt-params>

You may also find the examples given in the module's documentation useful:

<http://docs.python.org/library/csv.html#csv-examples>

```
import csv

symbol_to_properties = {...}

output = open('chem_props.txt','wb')

data = csv.writer(output)

for symbol in symbol_to_properties:
    (name, anum, boil) = symbol_to_properties[symbol]
    data.writerow([symbol, name, anum, boil])
del symbol, name, anum, boil
del data

output.close()
del output
```

csv2.py

78

Above is an example of using the **csv** module to write to a CSV file. This example is in the file **csv2.py** in your course home directories. Again, for simplicity I have not included any exception handling in this script, but if it was intended for serious use then it would include some exception handling.

We can try out this script if we wish:

```
$ python csv2.py
$ more chem_props.txt
Ru,ruthenium,44,4423.0
Re,rhenium,75,5900.0
Ra,radium,88,2010.0
Rb,rubidium,37,961.0
Rn,radon,86,211.3
Rh,rhodium,45,3968.0
```

(For space reasons, only an excerpt of the **chem\_props.txt** file is shown above.)

You should understand all the Python in the script above. If there is anything you do not understand, please ask the course giver now.

# Formatting options for CSV files

1.0, 2.5, 3.0, -6.8, 23.4  
5.6, 2.8, 9.3, -4.6, 9.8  
-3.9, 25.0, 1.23, 5.6, 7.8  
2.9, 5.2, 6.7, 2.4, 5.6

**delimiter = ','**  
**delimiter** is a string that specifies the character being used as the delimiter.  
Default value: ',', '

**skipinitialspace = True**  
will ignore any whitespace immediately after the delimiter. Default value: **False**

79

We'll look at some of the most common of these optional formatting parameters now.

Note that these parameters are *optional*, so you don't have to specify them: if you don't specify a parameter then Python will use that parameter's default value. When you call the **reader()** or **writer()** functions you can specify as many, or as few, of these optional parameters as you wish. If you specify any of these optional parameters, they must come *after* the **file** object you give to the **reader()** or **writer()** function, separated by a comma (,) as is usual for multiple arguments for a function in Python..

The **delimiter** parameter specifies the delimiter that separates the fields in the CSV file. It is a one-character string and defaults to the comma (',').

The **skipinitialspace** parameter is a Boolean that controls whether any spaces immediately following a delimiter should be ignored or considered part of the data in the field. When set to **True** any whitespace immediately following the delimiter is ignored, when set to **False** any whitespace is considered part of the data in the field. Its default value is **False**.

These optional parameters are specified as named (or keyword) arguments that you give when you call the **reader()** or **writer()** functions, e.g. if **data** is a **file** object for a CSV file that has been opened for writing, and the CSV file uses the tab character ('\t') as its delimiter, you would call the **writer()** function like this:

```
writer(data, delimiter='\t')
```

Similarly, if you had a **file** object for a CSV file that was opened for reading in **data**, and that CSV file used a space as its delimiter, and you wanted to ignore any extra spaces between fields, you would call the **reader()** function like this:

```
reader(data, delimiter=' ', skipinitialspace=True)
```

# Formatting options for CSV files

```
["Fred Smith", red, 56.9  
 "Joe Bloggs", blue, 27.8  
 "Jill East", brown, 28.9]
```

**quotechar = ''**  
**quotechar** is a string that specifies the character to be used for quoting.  
Default value: ''

The **quoting** parameter controls when things should be quoted.

The default is to only quote fields that contain special characters (the delimiter, etc).

80

The **quotechar** parameter specifies the character to be used for quoting. It is a one-character string and defaults to the double quote character ('"').

It is another optional parameter that you can specify when you call the **reader()** or **writer()** function. For example, if **data** is a **file** object for a CSV file that has been opened for writing, and you want to use the single quote (') character for quoting, you would call the **writer()** function like this:

```
writer(data, quotechar="'")
```

When things are quoted is controlled by the optional **quoting** parameter. The default value for this tells Python only to quote a field when the data in that field contains a special character, such as the delimiter. We'll look at the values the **quoting** parameter can take next.

# Controlling quoting in CSV files

```
"Smith, Fred", red, 56.9  
"Bloggs, Joe", blue, 27.8  
"East, Jill", brown, 28.9
```

default behaviour

```
quoting = csv.QUOTE_MINIMAL
```

```
"Fred Smith", "red", 56.9  
"Joe Bloggs", "blue", 27.8  
"Jill East", "brown", 28.9
```

```
quoting = csv.QUOTE_NONNUMERIC
```

```
"Fred Smith", "red", "56.9"  
"Joe Bloggs", "blue", "27.8"  
"Jill East", "brown", "28.9"
```

```
quoting = csv.QUOTE_ALL
```

81

The **quoting** parameter can take one of four values. These values are special constants that are defined in the **csv** module. This means that to use one of these special values we have to prefix it by “**csv.**” so that Python knows where to find it. The four values (and their meanings) are:

<b>QUOTE_ALL</b>	Makes <b>writer</b> objects quote all fields.
<b>QUOTE_MINIMAL</b>	Makes <b>writer</b> objects only quote those fields which contain special characters in the data, such as the delimiter.
<b>QUOTE_NONNUMERIC</b>	Makes <b>writer</b> objects quote all non-numeric fields (i.e. fields whose data cannot be represented as a number). Makes <b>reader</b> objects convert all non-quoted fields to <b>floats</b> .
<b>QUOTE_NONE</b>	Makes <b>writer</b> objects never quote fields. If the field contains a special character, then the <b>writer</b> object will try to <i>escape</i> it. For further details on this behaviour see the <b>QUOTE_NONE</b> entry in the <b>csv</b> module’s documentation:  <a href="http://docs.python.org/library/csv.html#csv-contents">http://docs.python.org/library/csv.html#csv-contents</a> <b>QUOTE_NONE</b> makes <b>reader</b> objects do no special processing of quote characters, i.e. any quote characters will be treated as part of the data in the field.

For example, if **data** is a **file** object for a CSV file that has been opened for writing, and you want all text fields (i.e. all non-numeric fields) to be quoted, you would call the **writer()** function like this:

```
writer(data, quoting=csv.QUOTE_NONNUMERIC)
```

# Any questions?

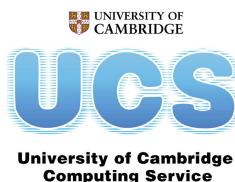
82

If there are any questions about what I have said today I'll (try to) answer them now. There will be another opportunity to ask questions at the start of the next day.

# Python: Further Topics

## Day Two

Bruce Beckles  
University of Cambridge Computing Service



1

Note that this course covers Python 2.4 to 2.7, which are the most common versions currently in use – it does **NOT** cover the recently released Python 3.0 (or 3.1) since that version of Python is so new. Python 3.0 is significantly different to previous versions of Python, and this course will be updated to cover it as it becomes more widely used.

The official UCS e-mail address for all scientific computing support queries, including any questions about this course, is:

[scientific-computing@ucs.cam.ac.uk](mailto:scientific-computing@ucs.cam.ac.uk)

# Introduction

- Who:
  - Bruce Beckles, e-Science Specialist, UCS
- What:
  - Python: Further Topics course, *Day Two*
  - Part of the **Scientific Computing** series of courses
- Contact (questions, etc):
  - [scientific-computing@ucs.cam.ac.uk](mailto:scientific-computing@ucs.cam.ac.uk)
- Health & Safety, etc:
  - Fire exits
- **Please switch off mobile phones!**

2

As this course is part of the Scientific Computing series of courses run by the Computing Service, all the examples that we discuss will be more relevant to scientific computing than to other programming tasks.

This does not mean that people who wish to learn about Python for other purposes will get nothing from this course, as the techniques and underlying knowledge taught are generally applicable. However, such individuals should be aware that this course was not designed with them in mind.

Note that there are various versions of Python in use, the most common of which are releases of Python 2.2, 2.3, 2.4, 2.5 and 2.6. (The material in this course is applicable to versions of Python in the 2.4 to 2.7 releases.)

On December 3rd, 2008, Python 3.0 was released. Python 3.0 is significantly different to previous versions of Python, is not covered by this course, and breaks backward compatibility with previous Python versions in a number of ways. As Python 3.0 and 3.1 become more widely used, this course will be updated to cover them.

# Related/Follow-on courses

## “Python: Operating System Access”:

- Accessing the underlying operating system (OS)
- Standard input, standard output, environment variables, etc

## “Python: Regular Expressions”:

- Using *regular expressions* in Python

## “Programming Concepts: Pattern Matching Using Regular Expressions”:

- Understanding and constructing regular expressions

## “Python: Checkpointing”:

- More robust Python programs that can save their current state and restart from that saved state at a later date
- “Python: Further Topics” is a pre-requisite for the “Python: Checkpointing” course

## “Introduction to Gnuplot”:

- Using *gnuplot* to create graphical output from data

3

For details of the “Python: Operating System Access” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonopsys>

For details of the “Python: Regular Expressions” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonregexp>

For details of the “Programming Concepts: Pattern Matching Using Regular Expressions” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-regex>

If you are unfamiliar with regular expressions, the following Wikipedia article gives an overview of them:

[http://en.wikipedia.org/wiki/Regular\\_expression](http://en.wikipedia.org/wiki/Regular_expression)

...although that article does not express itself as simply as it might, so it may be most useful for the references it gives at the end. If you have met regular expressions before, but haven't yet used them in Python, then the “Python: Regular Expressions” course will teach you how to use them in Python. Alternatively, the Python “*Regular Expression HOWTO*” introductory tutorial also provides a good introduction to using regular expressions in Python:

<http://docs.python.org/howto/regex>

For details of the “Python: Checkpointing” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonchkpt>

For the notes of the “Introduction to Gnuplot” course, see:

<http://www-uxsup.csx.cam.ac.uk/courses/Gnuplot/>

If you are unfamiliar with gnuplot, you may wish to have a look at its home page:

<http://www.gnuplot.info/>

# Pre-requisites

- Ability to use a text editor under Unix/Linux:
  - Try gedit if you aren't familiar with any other Unix/Linux text editors
- Basic familiarity with the Python language (as would be obtained from the “[Python: Introduction for Absolute Beginners](#)” or “[Python: Introduction for Programmers](#)” course):
  - Interactive and batch use of Python
  - Basic concepts: variables, flow of control, functions, Python's use of indentation
  - Simple data manipulation
  - Simple file I/O (reading and writing to files)
  - Structuring programs (using functions, modules, etc)

4

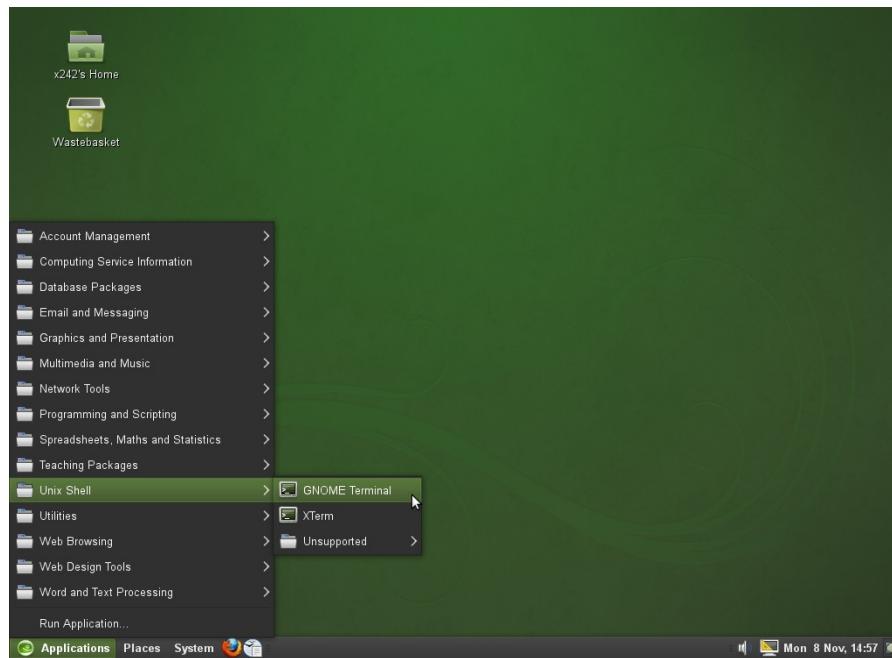
For details of the “Python: Introduction for Absolute Beginners” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-python>

For details of the “Python: Introduction for Programmers” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-python4progs>

# Start a shell



5

# Screenshot of newly started shell



# Recap: previous day

- **File I/O:**
  - Reading and writing files
  - Using the **csv** module to access structured text files
- **Exception handling**

7

On the previous day of the course, we’re examined two aspects of file I/O (input and output) in Python. In each of those areas we started off with a quick recap of the basics (as was covered in the “Python: Introduction for Absolute Beginners” and “Python: Introduction for Programmers” courses). We first looked at access to files and, after covering the basics, moved on to more advanced topics.

In the course of doing this we also looked at exception handling – which is how one copes with errors in Python – principally in the context of file I/O, but we also looked at exception handlers in other contexts, and how we could use them to make more robust functions.

Finally, we had a quick look at how we could use the **csv** module to access certain sorts of structured text files.

# Any questions?

8

If there are any questions about what I have said on the previous day of the course I'll (try to) answer them now. There will be another opportunity to ask questions at the end of today.

# Working with modules and functions

```
>>> import utils  
>>> reload(utils)  
<module 'utils' from 'utils.pyc'>
```

**reload( )** reloads an *already loaded* module from the file containing the module.

```
>>> dir(utils)  
['__builtins__', '__doc__', '__file__', '__name__', 'dict2file', 'file2dict',  
'find_root', 'greet', 'print_and_return', 'print_dict', 'reverse']
```

**dir( )** displays all the *names* defined within a module (or indeed in any type of object).

```
>>> callable(utils.file2dict)  
True  
>>> callable(utils.__doc__)  
False
```

**callable( )** tells us whether or not we can call something.

9

We already know how to load a module in Python using the `import` statement. We've also seen that if we make changes to the module we need to reload it by using the `reload()` function. If we try to import the module again, Python will not do anything since it knows it has already loaded (imported) the module. We have to explicitly tell it to `reload()` it.

How can we find out what functions are defined in a module? This is unfortunately not straightforward, although we can easily find out all the *names* that are defined in the module using the `dir()` function. These names will not be just the functions defined in the module though, they will be a mixture of any variables defined in the module, any functions defined in the module and also some special things created by Python (such as `__doc__` which contains the module's doc string). The special things created by Python will always be called something like `__name__`, i.e. they will be prefixed and followed by two underscore (`_`) characters. In general you disregard these, apart from the doc string (`__doc__`) which should contain useful information about the module.

Note that we can use the `dir()` function not just on modules, but on *any* object and it will tell us all the names that are defined within that object. (In case you were wondering, *everything* in Python is an object: modules, functions, variables, everything. What do we mean by "object" here? Basically it's a programming jargon term for a special sort of structure that can have both variables and functions defined within it.)

So how *can* we tell whether one of those names is a function or not? Well, we could try using the name as a function and seeing what happened, but that would quickly get tedious (as well as possibly giving false negatives). There's a better way: use the `callable()` function. The `callable()` function tells us whether a given name is callable, i.e. whether we can call it, i.e. if we can use it as a function. (However, you should be aware that there are pathological circumstances in which the `callable()` function will tell us that something is callable even when a call to it would fail; however, the converse (telling us something isn't callable when it is) should never happen.)

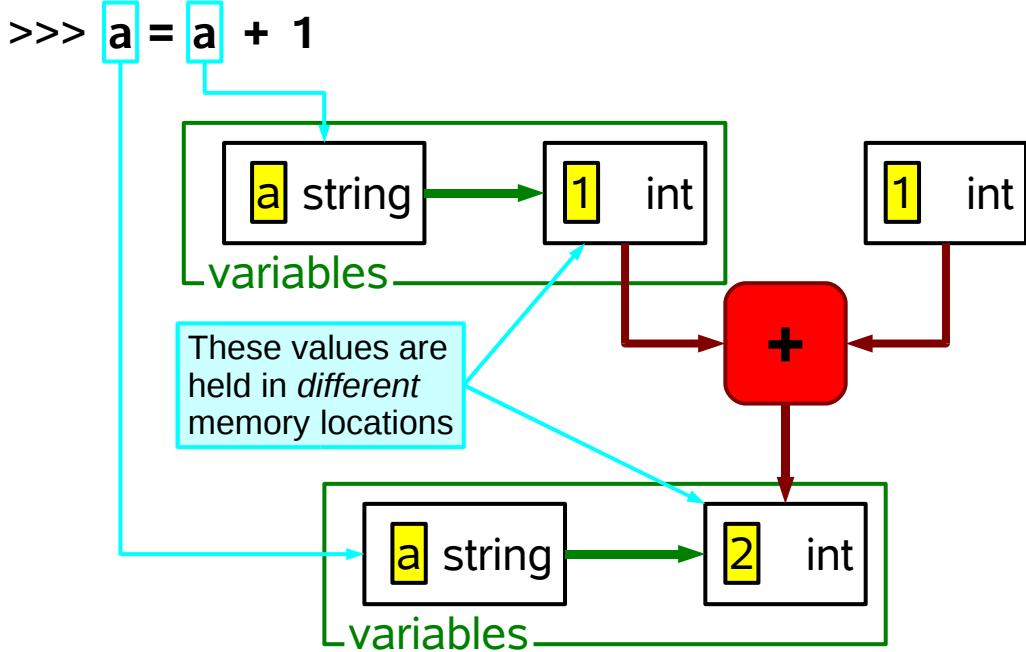
# Augmented assignment

>>> a = 1	>>> a = 1
>>> a += 1	>>> a = a + 1
>>> a	>>> a
2	2
>>> a -= 1	>>> a = a - 1
>>> a	>>> a
1	1
>>> a *= 4	>>> a = a * 4
>>> a	>>> a
4	4

Similarly, we can also use the following for...  
division: /=  
exponentiation: \*\*=  
remainder: %=

10

When we use the forms `+=`, `-=`, `*=`, `/=`, `**=` and `%=` we are doing what is known as *augmented assignment*. Basically, this is a combination of an operation (`+`, `-`, `*`, `/`, `**` or `%` respectively) and an assignment (assigning the result of that operation to a variable). You can also think of it as “assignment in place” because Python will attempt to update the variable’s value rather than creating a temporary value and then “pointing” the variable at that new value (which is what it does when we give it something like `a = a + 1`).



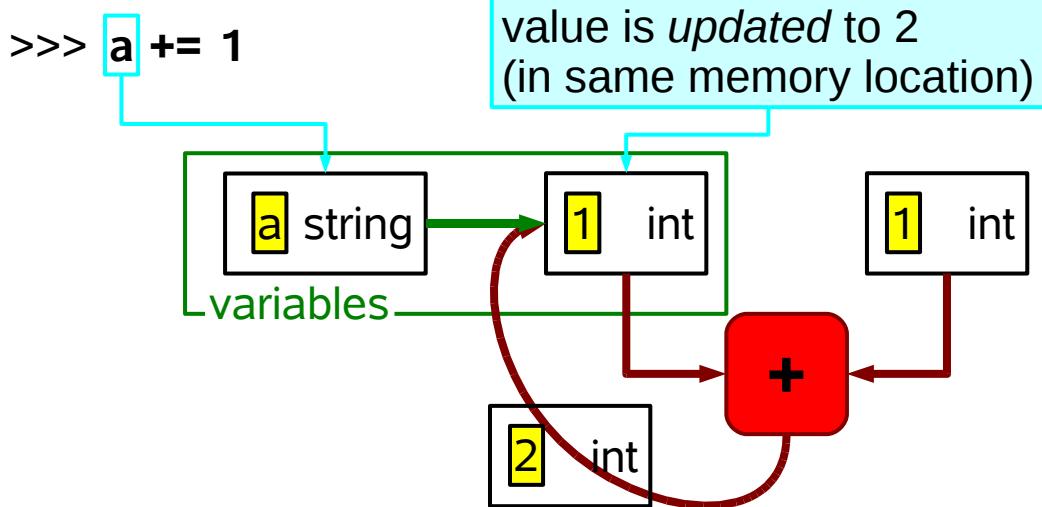
Variable is re-assigned to “point” at the answer,  
which is in a different part of memory

11

When you tell Python to do something like:

`a = a + 1`

what it does is look up the value of `a`, then adds 1 to that value and stores the answer in a *different* memory location. It then updates `a` to “point” to that new memory location and releases the memory that stored the previous value of `a`.



12

However, when you tell Python to do an augmented assignment, such as:

`a += 1`

what it does is look up the value of `a`, then adds 1 to that value and stores the answer in the *same* memory location (if it can), i.e. it updates a “in place”.

# Comparisons and conjunctions

```
>>> import utils  
>>> a = utils.print_and_return(1)  
1  
>>> a  
1  
  
>>> 0 < utils.print_and_return(1) and utils.print_and_return(1) < 3  
1  
1  
True  
  
>>> 0 < utils.print_and_return(1) < 3  
1  
True
```

**print\_and\_return()**  
function evaluated  
twice

...same truth value  
but function only  
evaluated once

13

We've already met the `and` conjunction for joining two comparisons together. However, there is a more compact way of doing something similar for the special case where we are doing something like:

*a compare b and b compare c*

(where “compare” stands for any comparison operator, such as “`<`” or “`>`”; note that the comparison operators used to compare `a` to `b` and `b` to `c` **do not** have to be the same). In this particular case, we can just drop the “`and`”, thus:

*a compare b compare c*

e.g. `a < b < c`, or even `a < b >= c`.

However, there is one important thing to note: in this more compact form, `b` is only evaluated **once**, whilst in “*a compare b and b compare c*”, `b` may be evaluated **twice**. We can easily see this if `b` is a function that has some side-effect (such as printing something on the screen) as in the slide above.

(The `print_and_return()` function is not a standard Python function. It was specially created for this course to illustrate this particular point. You will find it in the `utils` module in your course home directory. It just prints whatever argument it has been given and then returns that argument.)

# How **not** to copy a list

```
>>> list1 = [1, 2, 3, 4]  
>>> list2 = list1  
>>> list2  
[1, 2, 3, 4]
```

Is **list2** a copy of **list1**,  
or does it refer to the same  
*list* as **list1**?

```
>>> list1[2] = 7  
>>> list1  
[1, 2, 7, 4]  
>>> list2  
[1, 2, 7, 4]
```

**list1** and **list2**  
refer to the **same** list

14

If we've assigned a list to a variable (say a variable called `list1`) and we want to make a copy of that list (and assign that copy to another variable, say a variable called `list2`), we might be tempted to do something like this:

```
list2 = list1
```

Unfortunately this does **not** work in the way we might expect!

What happens is that both `list1` and `list2` now refer to the same list in the computer's memory. Changing `list1` will affect `list2` (and vice-versa), since they are both actually the same list. When we "copy" a list like this, we don't actually copy it at all, we just create a new variable that "points" to the same list that we had before. (This is sometimes called a "shallow copy".)

We can see that this is the case if we use the `id()` function. This function returns a constant, unique reference (an "identity") for each *unique* object that has been created. If two variables refer to the same object, then the `id()` function will return the same reference for both variables. (The reference will be an integer or long integer – what the `id()` function actually returns is the memory address at which the object is stored.) If you've typed in the Python on the slide above, you can try this function on `list1` and see what it returns:

```
>>> id(list1)
```

and then on `list2`:

```
>>> id(list2)
```

You should find that `id()` returns the same value for both these variables (whatever that value might happen to be).

So how *can* we make a real copy of a list?...

## Using list slices: copying a list

```
>>> list1 = [1, 2, 3, 4]  
>>> list2 = list1[:]  
>>> list2  
[1, 2, 3, 4]
```

Same question: Is **list2** a *copy* of **list1**, or does it refer to the *same list* as **list1**?

```
>>> list1[2] = 7  
>>> list1  
[1, 2, 7, 4]  
>>> list2  
[1, 2, 3, 4]
```

**list1** and **list2** refer to **different** lists: **list2** was a “genuine” copy of **list1**

Recall that **list1[:]** gives us a “slice” of the list that is the **entire** list (since we have not specified any indices).

15

Recall how we can get sections of a list: list *slices*. If **list1** is a list, we can get a “slice” of it using the syntax **list1[i:j]**, where *i* and *j* are indices of the list. **list1[i:j]** will give us all the items in the list from the item whose index is *i* up to and including the item whose index is *j-1*. We can exclude either or both of the indices in the slice; if we exclude both indices (so **list1[:]**) then the slice we get is the entire list.

In fact, that slice is a **copy** of the entire list. A real, genuine, honest-to-goodness copy that is a *different* list (with the same values in the same order), stored in a *different* memory location. (This is sometimes called a “deep copy”.)

Again, we can see that this is the case using the **id()** function. If you’ve typed in the Python on the slide above, you can try this function on **list1** and see what it returns:

```
>>> id(list1)
```

and then on **list2**:

```
>>> id(list2)
```

You should find that **id()** returns *different* values for each variable (whatever those values might happen to be). That means that they refer to different objects in memory (which may or may not happen to have the same value).

## Using list slices: insert

```
0 1 2 3 4 5 6 7  
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]  
>>> data[2:2] = [16, 17]  
>>> data  
[1, 2, 16, 17, 3, 4, 5, 6, 7, 8]
```

Items are inserted into the list *before* the item whose index is given

16

Note that the items we are inserting have to come from a list, and we can insert as many (or as few) items as we like.

You may wonder why the insertion is *before* the given index rather than after it. Recall that a slice starts from the lower index and goes up to just before (one less than) the higher index. So the slice  $i:i$  at first glance seems nonsensical because it would have to start at item  $i$  and stop just *before* item  $i$ . So Python interprets this as being “the empty space just before item  $i$ ”, which does not actually contain a value, so that, for any list the slice  $i:i$  will evaluate to the empty list, e.g.

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]  
>>> data[2:2]  
[]
```

You can also insert a *single* item into a list using the `insert()` method of the list, which inserts a single item **at** the given index, e.g.

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]  
>>> data.insert(2, 16)  
>>> data  
[1, 2, 16, 3, 4, 5, 6, 7, 8]
```

## Using list slices: replace (and insert)

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> data[2:3] = [24, 32, 17]
```

```
>>> data
```

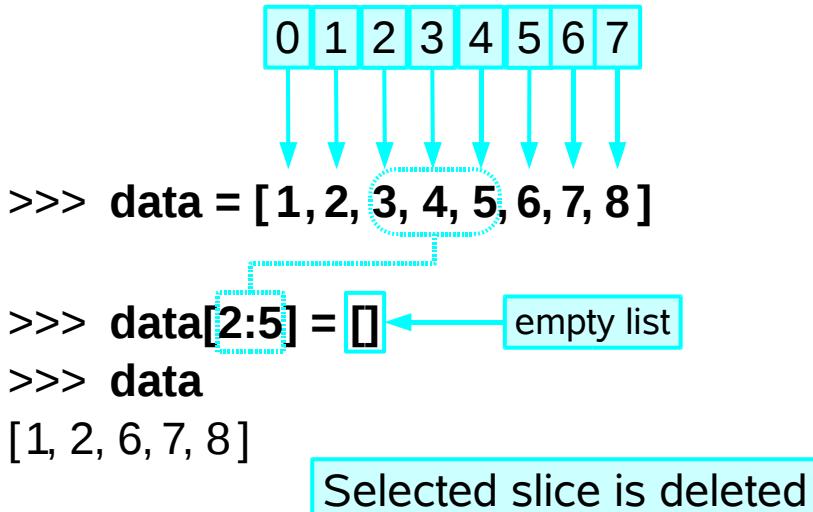
```
[1, 2, 24, 32, 17, 4, 5, 6, 7, 8]
```

Items are inserted into the list *replacing* the selected slice

17

As mentioned earlier, the items we are inserting have to come from a list, and we can replace the slice with as many (or as few) items as we like. Thus, if the slice we're replacing is not the empty list ([ ]), as it was in the previous example, then we will actually be “inserting and replacing” rather than just inserting...

## Using list slices: deletion



18

...which means that if we replace the slice with no items, i.e. the empty list ([]), then we'll actually delete the slice.

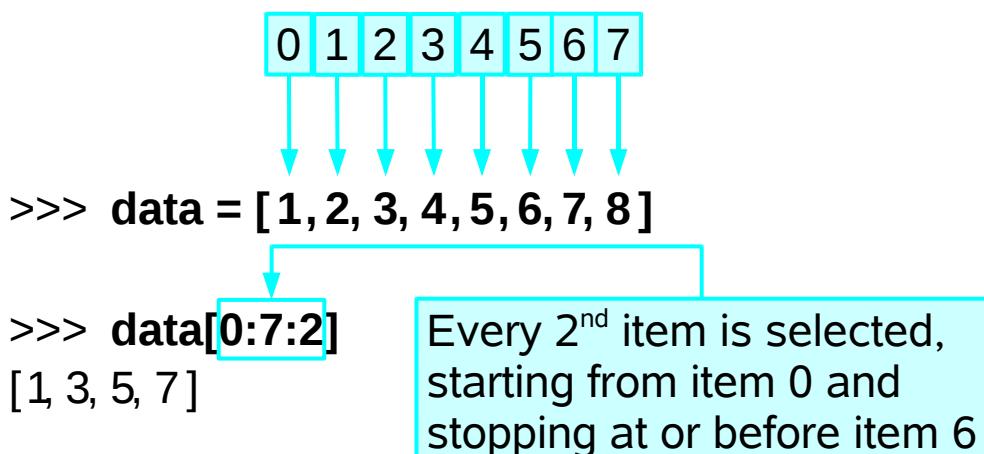
For comparison, remember that you can also delete a single item from a list using the list's `remove()` method. This method removes the *first* matching item in a list (wherever that item might be), e.g.

```
>>> data = [1, 2, 3, 2, 8]
>>> data.remove(2)
>>> data
[1, 3, 2, 8]
```

There's also another way you can delete items from a list: using the `del` operator. This operator can either delete a single item from a list, or an entire slice. `del list[i]` removes the item whose index is *i* from the list, whilst `del list[i:j]` removes the slice *i:j* from the list, e.g.

```
>>> data = [1, 2, 3, 2, 8]
>>> del data[2]
>>> data
[1, 2, 2, 8]
>>> del data[0:2]
[2, 8]
```

## List slices: selecting part of a slice



19

This may seem slightly odd until you get used to it. The way to think of it is that the slice  $i:j:k$  (which you can read as “the slice  $i:j$  in steps of size  $k$ ”) gives you the following items from the slice  $i:j$  –

- item  $i$
- item  $i + k$
- item  $i + 2*k$
- item  $i + 3*k$
- item  $i + 4*k$

...and so on, up to (but **not** including) item  $j$ , i.e. (for the mathematically inclined) we stop at  $i + n*k$ , where

$$i + n*k < j \leq i + (n+1)*k$$

Having selected part of a slice in this way, you can replace the items you’ve selected in a similar manner to the way in which we’ve seen we can replace an ordinary slice of a list, i.e. we set the selected part of the slice equal to another list of items. There is one restriction, though: we must replace this part of a slice with exactly the *same* number of items, e.g.

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]
>>> data[0:7:2] = [3, 9, 15, 21]
>>> data
[3, 2, 9, 4, 15, 6, 21, 8]
```

This restriction means that we can’t remove these sorts of parts of a slice by setting them equal to the empty list ([]), as we can with normal slices. Oh, well, you can’t have everything.

## List repetition

```
>>> data = [1, 2, 3]
```

```
>>> data * 3
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

multiplication operator: \*

```
>>> data * 0
```

```
[]
```

“multiplying” by 0 gives the empty list

```
>>> data * -5
```

```
[]
```

“multiplying” by a negative integer also gives the empty list

20

If we “multiply” a list by an integer (either a normal integer or a long integer) we will get list *repetition*: a new list is generated which consists of the original list repeated the specified number of times. If we “multiply” a list by a negative integer or by zero, then we get the empty list ([]).

Note that we can’t “multiply” a list by a floating point number or a complex number.

Evaluate the following Python statements in your head. What are the items in the list `primes` after each statement?

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19]
>>> primes[0::3] = [1, 6, 16]
>>> del primes[2:5]
>>> primes[2:2] = [5] * 3
>>> primes[0::4] = [2, 11]
>>> primes[3::3] = [7, 17]
```

Now try them interactively in Python and see if you were correct.

21

If you run into problems with this exercise, or if you don't understand any of the Python on the slide above, please ask the course giver or a demonstrator for help.

When you've finished take a short break of a minute or two – that means **stop** staring at the computer screen and move around, relax, etc.

(Note: If you've done it correctly, you should find that the items in `primes` when you've finished are the same (and in the same order) as when you first assigned a list to `primes`.)

## When **not** to use list repetition

```
>>> x = [ [0, 0] ] * 2
```

```
>>> x
```

```
[[0, 0], [0, 0]]
```

```
>>> x[0][0] = 1
```

```
>>> x
```

```
[[1, 0], [1, 0]]
```



probably **not** what we wanted to happen...

List repetition works fine if the list consists of simple data types (integers, floats, complexes, etc.) but with more complicated types (e.g. a list of *lists*) the new list contains “**shallow copies**” of the repeated item(s).

22

And now a very important “gotcha”: list repetition, used in the wrong circumstances, will not behave the way we might expect.

If we use list repetition on a list of *lists*, then the new list consists of a set of “*shallow copies*” of the repeated items, as we see on the slide above. Thus, in the example above, instead of having a list of two items, each of which is a distinct list (that just happen to have the same values in the same order when we first set them up), we have a list of two items, each of which is the *same* list, c.f. what happened earlier when we tried to copy a list without using slices.

(Note that we get this wrong-headed behaviour whenever we use list repetition on a list whose items are themselves complicated types, such as lists or dictionaries.)

You may be wondering why we would want to have a list of lists like the one we want to create on the slide above. Such lists are often used as matrices. Since Python doesn’t have a built-in matrix type, people often use a list of lists instead. So, on the slide above, *x* (if it behaved properly) could represent a  $2 \times 2$  matrix. Similarly, the  $4 \times 4$  identity matrix might be represented by the following list:

```
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
```

If you are going to do serious matrix work in Python, though, you are better off using the NumPy module. This is not a standard Python module, but is freely available from:

<http://www.scipy.org/Download>

For documentation on the NumPy module, see:

<http://docs.scipy.org/>

Using matrices in Python and basic use of the NumPy module are covered in the “Python: Interoperation with Fortran” course. For details of this course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonfort>

So how can we do something like list repetition for a list of lists? Well, first we need to know a little more Python...

# List comprehension

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> x = [3 * d for d in data]
```

```
>>> x
```

```
[3, 6, 9, 12, 15, 18, 21, 24]
```

Operation or function  
on each item in list

for loop over list

23

The Python technique we need is called *list comprehension*.

List comprehension is a handy technique for quickly creating one list from another. Basically, you specify an operation or function to be carried out on each item in an existing list. Python will then construct a new list for you whose items are the results of carrying out the specified operation or function on the items (in order) in the old list.

Note that the old list doesn't, in fact, have to be a list at all: anything that you can legitimately treat as a list for the purposes of a for loop (a dictionary, a file object, etc) can be used.

## List comprehensions to repeat a list

```
>>> x = [ [0, 0] for d in range(0,2) ]
```

```
>>> x  
[[0, 0], [0, 0]]
```

```
>>> x[0][0] = 1
```

```
>>> x  
[[1, 0], [0, 0]]
```

Yay! It works!

To repeat a list of *lists* (or other complicated data types), **don't** use list repetition, use a list comprehension instead.

24

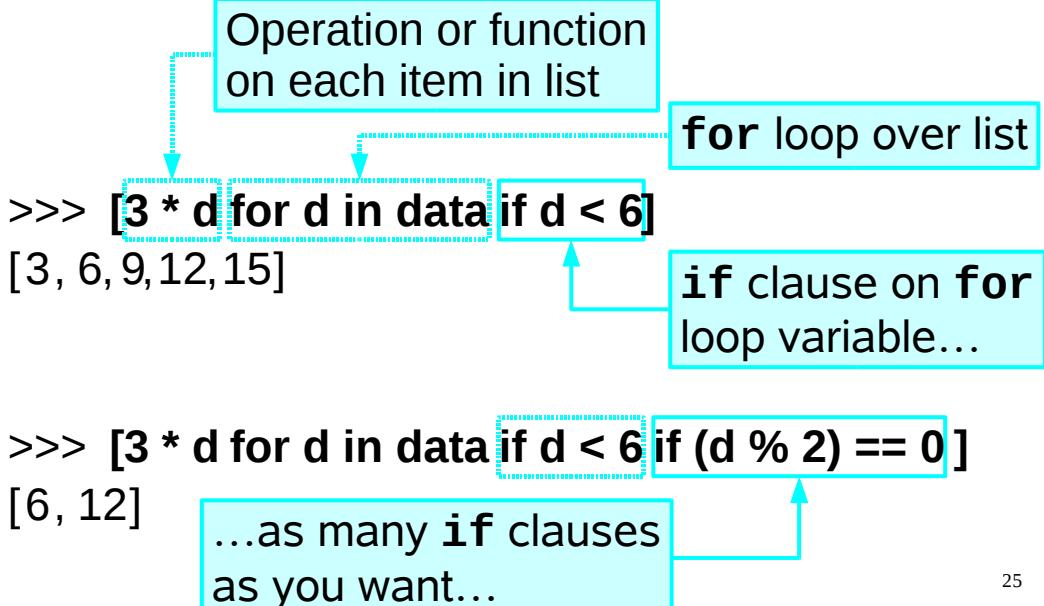
So now we can sensibly repeat a list of lists (or other complicated data types).

As we see above, the “operation” that we carry out in our list comprehension can in fact be a constant value (which can be of any type: integers, floats, even lists (as above)) – this will create a new list, each of whose items is a **copy** (a “deep copy”) of the specified value.

(Recall that the `range()` function gives me a list of integers from the first integer to one less than the second integer, so `range(0,2) = [0, 1]`.)

## More list comprehensions

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]
```



25

List comprehensions are even more versatile than you might at first imagine – as well as looping over an existing list, we can also add one or more `if` clauses to our list comprehension to further limit the items from the original list upon which we want our operation or function to act.

So the list comprehension

```
[3 * d for d in data if d < 6]
```

should be read as something like “for each item in the list `data` whose value is less than 6, multiply 3 by that item and add it to our new list”.

And the list comprehension

```
[3 * d for d in data if d < 6 if (d % 2) == 0]
```

should be read as something like “for each item in the list `data` whose value is less than 6, if that item is divisible by 2, multiply 3 by that item and add it to our new list”.

Recall that for integers `%` means “the (non-negative) remainder when divided by” (usually read as “mod”, short for “modulo”), so the expression “`d % 2`” is only equal to 0 if `d` is even. (We don’t actually need the brackets around the “`d % 2`” in the `if` clause, I’ve just put them in there for clarity.)

## Yet more list comprehensions

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> x = [p for d in data if d < 4 for p in range(0, d)]
```

```
>>> x  
[0, 0, 1, 0, 1, 2]
```

Operation or function on each item in a list

for loop

if clause

Another for loop

26

In fact, they are quite impressively versatile – as well as adding one or more `if` clauses to our list comprehension, we can also add one or more *additional* `for` loops. So, the general form of a list comprehension is:

[ “function or operation” “for loop” “zero or more if clauses and/or for loops” ]

So the list comprehension

```
[p for d in data if d < 4 for p in range(0, d)]
```

should be read as something like “for each *item* in the list `data` whose value is less than 4, loop over the temporary list `range(0, item)`, adding each item from this temporary list to our new list”.

I.e. the following line of Python:

```
x = [p for d in data if d < 4 for p in range(0, d)]
```

is equivalent to:

```
x = []  
for d in data:  
    if d < 4:  
        for p in range(0, d):  
            x.append(p)
```

(Recall that the `range()` function gives me a list of integers from the first integer to one less than the second integer, so, for example, `range(0, 3) = [0, 1, 2]`.)

Evaluate the following Python statements in your head. `primes` is defined as:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

```
>>> [8 / p for p in primes]
```

```
>>> [p for p in primes if p % 3 > 0]
```

```
>>> [[0,0,0] for x in range(2,5)]
```

```
>>> [93 % p for p in primes if 93 % p != 0]
```

```
>>> [5 ** p for p in primes if p % 4 == 0]
```

```
>>> [2 * x for p in primes[0::2] for x in range(p-1,p+2)]
```

Now try them interactively in Python and see if you were correct.

27

If you run into problems with this exercise, or if you don't understand any of the Python on the slide above, please ask the course giver or a demonstrator for help.

Recall that:

- For integers, `a % b` means “the (non-negative) remainder when `a` is divided by `b`” (usually read as “`a mod b`”, “mod” being short for “modulo”); and
- The `range()` function gives me a list of integers from the first integer to one less than the second integer, so `range(6,9) = [6, 7, 8]`.

When you've finished take a short break of one or two minutes – remember that, in this context, “break” means “break from using the computer”.

# How not to copy a dictionary

```
>>> dict1 = {'H':1, 'He':2}  
>>> dict2 = dict1  
>>> dict2  
{'H': 1, 'He': 2}
```

Is **dict2** a copy of **dict1**,  
or does it refer to the *same*  
*dictionary* as **dict1**?

```
>>> dict1['H'] = 1.0079  
>>> dict1  
{'H': 1.0079, 'He': 2}  
>>> dict2  
{'H': 1.0079, 'He': 2}
```

**dict1** and **dict2**  
refer to the **same** dictionary

28

We've seen how we can "properly" copy a list. What about if we want to copy a dictionary? If we've assigned a dictionary to a variable (say a variable called `dict1`) and we want to make a copy of that dictionary (and assign that copy to another variable, say a variable called `dict2`), we might be tempted to do something like this:

```
dict2 = dict1
```

Unfortunately, as with lists, this does **not** work in the way we might expect!

What happens is that both `dict1` and `dict2` now refer to the same dictionary in the computer's memory. Changing `dict1` will affect `dict2` (and vice-versa), since they are both actually the same dictionary. When we "copy" a dictionary like this, we don't actually copy it at all, we just create a new variable that "points" to the same dictionary that we had before. (This is sometimes called a "shallow copy".)

Again, we can see that this is the case if we use the `id()` function. If you've typed in the Python on the slide above, you can try the `id()` function on `dict1` and see what it returns:

```
>>> id(dict1)
```

and then on `dict2`:

```
>>> id(dict2)
```

You should find that `id()` returns the same value for both these variables (whatever that value might happen to be).

So how *can* we make a real copy of a dictionary?...

## How to copy a dictionary

```
>>> dict1 = {'H':1, 'He':2}  
>>> dict2 = dict1.copy()  
>>> dict2  
{'H': 1, 'He': 2}
```

Same question: Is **dict2** a *copy* of **dict1**, or does it refer to the *same dictionary* as **dict1**?

```
>>> dict1['H'] = 1.0079  
>>> dict1  
{'H': 1.0079, 'He': 2}  
>>> dict2  
{'H': 1, 'He': 2}
```

**dict1** and **dict2** refer to **different** dictionaries: **dict2** was a “genuine” copy of **dict1**

29

...Well, fortunately, dictionaries provide a method, the `copy()` method, that allows us to do just that: create a real, genuine, honest-to-goodness **copy** that is a *different* dictionary (with the same key/value pairs), stored in a *different* memory location. (This is sometimes called a “deep copy”.) As `copy()` is a method of dictionaries, we can use it on any dictionary – it returns a *copy* of the dictionary:

```
>>> {'H':1, 'He':2}.copy()  
{'H': 1, 'He': 2}
```

Again, we can see that this is the case using the `id()` function. If you’ve typed in the Python on the slide above, you can try the `id()` function on `dict1` and see what it returns:

```
>>> id(dict1)
```

and then on `dict2`:

```
>>> id(dict2)
```

You should find that `id()` returns *different* values for each variable (whatever those values might happen to be). That means that they refer to different objects in memory (which may or may not happen to have the same value).

## Sorting lists

```
>>> data = [8, 4, 3, 1, 5, 6, 7, 2]
```

```
>>> data.sort()
```

sort( ) method:  
sorts a list “in place”

```
>>> data
```

Note **no** value returned

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

...instead the list is sorted

To reverse the sort order  
use **reverse=True**

```
>>> data.sort(reverse=True)
```

```
>>> data
```

list sorted in  
reverse order

```
[8, 7, 6, 5, 4, 3, 2, 1]
```

30

Another method that lists possess is the `sort()` method. This sorts a list “in place”.

This method also provides a quick way to reverse the sort order: call the `sort()` method setting the `reverse` named argument to the Boolean `True` (i.e. call the method using `sort(reverse=True)` rather than just `sort()`). Note that the `reverse` named argument was introduced in Python 2.4, so you can’t use it in earlier versions of Python.

Note that lists also have a `reverse()` method that does **not** do a reverse sort of the list, but rather reverses (“in place”) the order of the items in the list:

```
>>> data = [8, 4, 3, 1, 5, 6, 7, 2]
>>> data.reverse()
>>> data
[2, 7, 6, 5, 1, 3, 4, 8]
```

(Obviously, this means you could also do a reverse sort of a list by calling the `sort()` method immediately followed by the `reverse()` method, but it is easier and much more efficient to just call the `sort()` method with `reverse=True`.)

The `sort()` method also allows you to define your own sort order for sorting a list – you do this by using defining a comparison function and giving that function to the `sort()` method as an argument. For further details see the Python Library Reference sub-section on “Mutable Sequence Types”:

<http://docs.python.org/library/stdtypes.html#typesseq-mutable>

# Exercise

Write a function that takes a dictionary and **prints** out its values in ascending order.

Dictionary → values in ascending order

```
{'Ar': 39.95,      1.0079  
'H': 1.0079, → 14.007  
'N': 14.007}    39.95
```

...since if we arrange the values of the above dictionary in ascending order, they look like this:  
1.0079, 14.007, 39.95

31

So if the function took as its input the dictionary:

```
{ 'Ar' : 39.95, 'H' : 1.0079, 'N' : 14.007 }
```

it would produce the output:

```
1.0079  
14.007  
39.95
```

If you run into problems with this exercise, ask the course giver or a demonstrator for help.

(An answer is given on the page after next.)

*Hint:* Recall that if `x` is a dictionary then `x.keys()` gives you a list of the dictionary's keys (in a might as well be random order) whilst `x.values()` gives you a list of the values in the dictionary (also in a (possibly different) might as well be random order).

# Exercise redux

Write a function that takes a dictionary and **prints** out its values in *descending* order of the corresponding **keys**.

Dictionary → values in descending order of keys

```
{'H': 1.0079,      14.007  
'N': 14.007, → 1.0079  
'Ar': 39.95}     39.95
```

...since if we arrange the keys of the above dictionary in descending order, they look like this:  
'N', 'H', 'Ar'

32

So if the function took as its input the dictionary:

```
{ 'H': 1.0079, 'N': 14.007, 'Ar': 39.95 }
```

it would produce the output:

```
14.007  
1.0079  
39.95
```

If you run into problems with this exercise, ask the course giver or a demonstrator for help.

After this exercise take at least a 5 or 10 minute break. Remember that this means you should **stop** using the computer, and move around, exercise your arms, wrists, neck, etc.

(An answer is given to this exercise on the page after next.)

*Hint:* Recall that if `x` is a dictionary then `x.keys()` gives you a list of the dictionary's keys (in a might as well be random order) whilst `x.values()` gives you a list of the values in the dictionary (also in a (possibly different) might as well be random order).

# Answer to Exercise

```
def print_dict_values_sorted(dict):  
  
    sorted_values = dict.values()  
    sorted_values.sort()  
  
    for value in sorted_values:  
        print value
```

33

Here is a solution to the first exercise that you were to attempt over the break.

If there is anything in the solution that you do not understand, or if your solution looks utterly different from that shown above, please tell the course giver or demonstrator.

## Answer to Exercise *redux*

```
def print_dict_values_sorted_by_reverse_keys(dict):  
  
    ordered_keys = dict.keys()  
    ordered_keys.sort(reverse=True)  
  
    for key in ordered_keys:  
        print dict[key]
```

34

Here is a solution to the second exercise that you were to attempt over the break.

If there is anything in the solution that you do not understand, or if your solution looks utterly different from that shown above, please tell the course giver or demonstrator.

# Temporary files

*Temporary files: a great way  
to accidentally give access to  
your system to someone who  
shouldn't have it.*

35

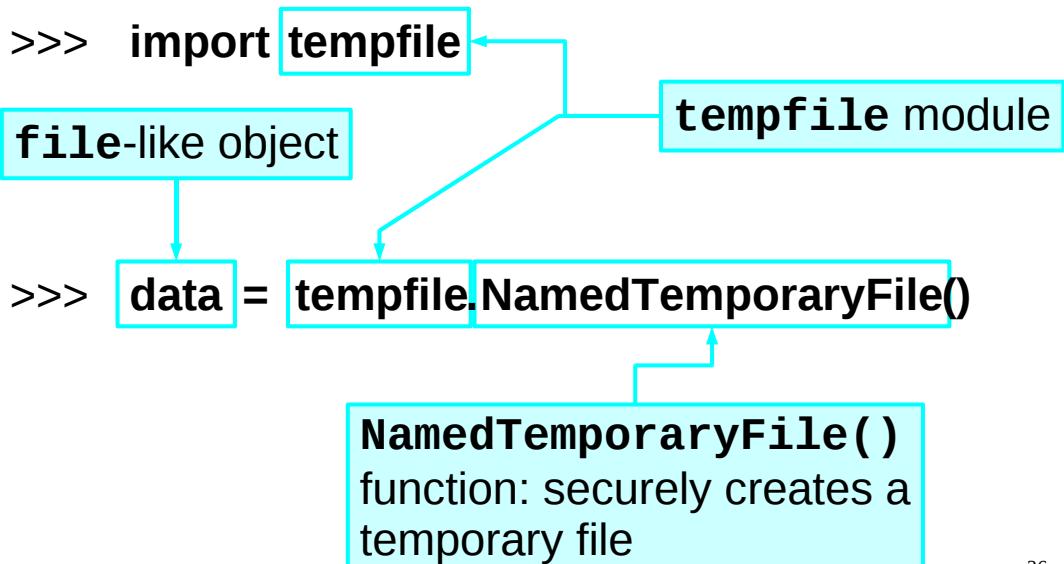
We're going to briefly return now to file I/O to look at one particular aspect of it: temporary files.

Often we need a file to write some data to for a short period of time, which we will then delete. We might need to do this because we need to pass some intermediate data to another program for processing but we don't want to keep that intermediate data.

Some of you may think: "but I already know how to create a file, why don't I just create a temporary file myself?". In general, that's an extremely **bad** idea – on a multi-user system it is very difficult to securely create a temporary file, and very easy to insecurely create one, which, over the years, has led to any number of security holes in systems that have allowed unauthorised people to get access to the system.

Fortunately, there are a number of functions that have been provided which do this for us in a safe, secure manner. We'll look at two of them now.

# NamedTemporaryFile()



36

The `NamedTemporaryFile()` function (which lives in the `tempfile` module) will securely create a temporary file for us, which it will delete when we close the file. It returns a `file`-like object (“like” as in it has all the familiar properties and methods of `file` objects, but it is actually a different type of object). This function was introduced in Python 2.3, so you can’t use it in earlier versions of Python.

The temporary file is opened in binary mode, and also is opened for both reading and writing (this is a new mode we haven't yet met, which is specified by using '`w+b`' – '`w+`' specifies the file should be opened for *both* reading and writing, the '`b`' on the end specifies it should be opened in **binary** mode). Note that if the file already exists, opening it in '`w+`' mode will remove its contents (just as opening it in ordinary '`w`' mode does). Since this is a temporary file specially created for us this doesn't matter.

If, however, you want the temporary file opened in a different mode, then you can specify a mode to the `NamedTemporaryFile()` function, like this:

```
tempfile.NamedTemporaryFile(mode='w')
```

which would create a temporary file for writing (in text mode).

Note that `NamedTemporaryFile()` will *delete* the temporary file when we close it.

# NamedTemporaryFile()

```
>>> import tempfile  
>>> data = tempfile.NamedTemporaryFile()
```

```
>>> data.name
```

'/tmp/tmpXI3Yj7'

**name** attribute  
holds the file's name

```
>>> data.close()
```

File is deleted on **close()**

37

The name of the temporary file, in case this is of interest, lives in the `name` attribute of the `file-like` object created by the `NamedTemporaryFile()` function.

Now, `NamedTemporaryFile()` will delete the temporary file when we close it, which might not be what we want if, for instance, we want to create a temporary file to pass to another program. So how can we securely create a temporary file without having it automatically deleted?...

(Note that if you try the Python commands above, you will almost certainly get a completely different file name for the temporary file.)

(Finally, note that, starting with Python 2.6, `NamedTemporaryFile()` has a named argument, `delete`, that you can set to `False` when calling `NamedTemporaryFile()` if you do not want the temporary file to be deleted when it is closed, like this:

```
tempfile.NamedTemporaryFile(delete=False)
```

Unfortunately, this functionality does not exist in versions of Python prior to Python 2.6, so you can't do this if you are using a version of Python earlier than 2.6.)

# mkstemp()

```
>>> import tempfile
```

**mkstemp( ) function:**  
securely creates a  
temporary file

```
>>> (fhandle, fname) = tempfile.mkstemp()
```

OS file handle to  
the opened file

Name (and full path)  
of the temporary file

38

The `mkstemp()` function (which also lives in the `tempfile` module) will securely create and open for both reading and writing (in binary mode) a temporary file for us, but having created it, it leaves it alone. It is up to us to delete it when we've finished using it. The `mkstemp()` function returns a tuple consisting of a *file handle* to the opened file, and the file's name (and full path), as a string. If you want `mkstemp()` to open the file in text mode, set the named argument `text` to `True` when calling `mkstemp()`, like this:

```
tempfile.mkstemp(text=True)
```

(Note that the `mkstemp()` function was also introduced in Python 2.3, so you can't use it in earlier versions of Python.)

The file handle is **not** a `file` object, and so does not have all the useful `file` object methods. Instead it provides low level operating system (OS) access to the file, which is not something we wish to use if we can help it. Consequently the best thing to do with this file handle is use it to create a Python `file` object, after which we can forget about it and just use the familiar Python `file` object methods. How do we do that...?

## mkstemp()

```
>>> import tempfile  
>>> (fhandle, fname) = tempfile.mkstemp()
```

OS file handle to  
the opened file

```
>>> import os
```

```
>>> data = os.fdopen(fhandle, 'wb')
```

open file  
for writing,  
in **binary**  
mode

**fdopen()** function: creates a **file** object  
from an OS file handle

39

To create a Python **file** object from a file handle we need to use the **fdopen()** function that lives in the **os** module. If we give the **os.fdopen()** function an open file handle, it will create a corresponding Python **file** object for us, created with the specified mode (if we don't specify a mode it behaves as though we specified a mode of '**r**'). (The mode that we give to **os.fdopen()** is the same as we would give to the **open()** command, except that it *must* start with an '**r**', '**w**' or '**a**').

The mode we give **os.fdopen()** **must** be compatible with the mode which was used when creating the file handle. So, if **tempfile.mkstemp()** has opened the file in **binary** mode (its default behaviour), then we should tell **os.fdopen()** to do likewise (i.e. add a '**b**' to the end of the mode we give **os.fdopen()**). Similarly, if **tempfile.mkstemp()** has opened the file in **text** mode, we should tell **os.fdopen()** to do likewise (no '**b**').

Once we've created a **file** object for our newly created temporary file, we can get on with accessing it in the normal Python manner (using the **write()** method, etc). **Remember** to close the file using the **file** object's **close()** method when you've finished using it!

**It is only when a file is closed that the writes to it are committed to the file system.**

# Saving complex objects to a file

## Object serialization: **pickle** and **cPickle** modules

40

Python has two modules which can be used for what is sometimes called “object serialization”, which is also known – in the Python world – as “pickling”. This is essentially a way of taking a Python object and storing it in a compact format (usually on disk). (“Serialization” is also known as “marshalling” or “flattening”, although Python uses the term “marshalling” in a more specialised manner.)

Python can pickle almost all its basic object types – integers, long integers, floating point numbers, complex numbers, Booleans, the `NoneType`, strings, etc – and, more usefully, many of its composite data types – such as lists, tuples and dictionaries – provided all their individual items are also objects it can pickle. Thus, if, for example, your dictionary contains only integers, floating point numbers, etc, or lists or tuples of such objects, then you can pickle it. This provides a very easy way of storing a complex object like a dictionary or a list of lists without you having to individually write each item the object contains out to a file. You can find the complete list of objects that can be pickled in the “What can be pickled and unpickled?” subsection of the `pickle` module’s documentation:

<http://docs.python.org/library/pickle.html>

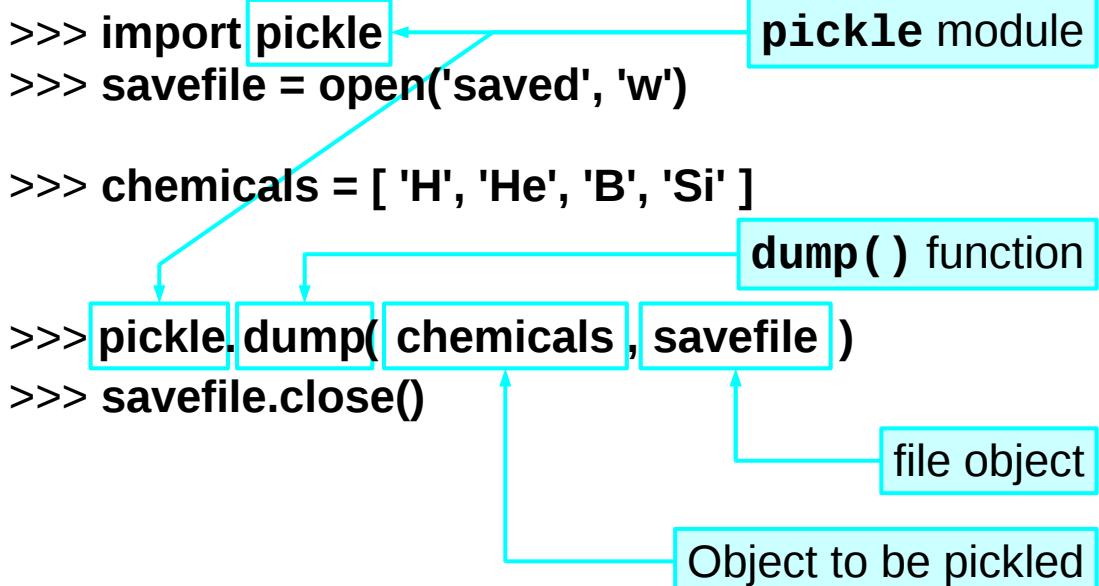
There are two modules which you can use almost interchangeably for pickling – the `pickle` module and the `cPickle` module. Why are there two of them? Well, the `cPickle` module is implemented in C and so is much, much faster than the `pickle` module. However, the `pickle` module can be extended using Python’s object oriented framework (not covered in this course). So if you have some special requirement that can’t be satisfied by the built-in `pickle` module, you might want to extend it – which you can’t do with the `cPickle` module. Most users don’t need to do this though, and so can use the `cPickle` module (and gain the benefit of its speed).

Python guarantees that if you use the `pickle` module to store something, you can load it again using the `cPickle` module, and vice-versa. In addition, if you pickle something on one machine, you can load it again on a different machine, even if that machine is running a different operating system or has a different version of Python (well, provided the versions of Python aren’t *too* different).

The `pickle` and `cPickle` modules are covered in more detail in the “Python: Checkpointing” course:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonchckpt>

## Pickling data to a file



41

As previously mentioned, Python can pickle almost all its basic object types – integers, floating point numbers, strings, etc, and, more usefully, many of its composite data types – such as lists, tuples and dictionaries – provided all their individual items are also objects it can pickle. Thus, if, for example, your dictionary contains only integers, floating point numbers, etc, or lists or tuples of such objects, then you can pickle it.

The basic way of “pickling” data to a file is to use the `dump()` function. The `dump()` function works on `file` objects, so you need to open the file (for writing) before calling the `dump()` function.

As mentioned before, you can use the `dump()` function from either the `pickle` or the `cPickle` module.

If you give the `dump()` function something that it cannot pickle, Python will raise a `PicklingError` exception (as this exception is defined in the `pickle` and `cPickle` modules, if you wish to handle it you would refer to it as `pickle.PicklingError` or `cPickle.PicklingError`). In rare cases, attempting to pickle a very complex data structure may cause a `RuntimeError` exception to be raised.

**Remember** to close the file to which you are pickling using its `close()` method when you've finished using it.

**It is only when a file is closed that the writes to it are committed to the file system.**

You normally only store a single “pickle” of data in a file. If you need to pickle several pieces of data and store them in the same file, just put all the data into a tuple and pickle the tuple. The author knows of no good reason to store multiple “pickles” of data in a single file. If, however, you are absolutely convinced you need to do this, then have a look at the `shelve` module (one of the standard Python modules).

## Restoring pickled data

```
>>> import cPickle  
>>> savefile = open('saved')  
  
>>> new_chemicals = cPickle.load( savefile )  
  
>>> savefile.close()  
>>> print new_chemicals  
[ 'H', 'He', 'B', 'Si' ]
```

The diagram illustrates the flow of variables in the code. It shows the 'cPickle module' being imported, a 'file object' being opened, the 'load()' function being called on that file object to restore the data, and finally a variable holding the restored data.

42

The basic way of restoring pickled data from a file is to use the `load()` function. The `load()` function works on `file` objects, so you need to open the file (for reading) before calling the `load()` function. (Obviously, once you've restored the pickled data, make sure you `close` the file.)

As mentioned before, you can use the `load()` function from either the `pickle` or the `cPickle` module.

If the `load()` function has a problem with unpickling the data, Python will usually raise an `UnpicklingError` exception. (Note that as this exception is defined in the `pickle` and `cPickle` modules, if you wish to handle it you would refer to it as `pickle.UnpicklingError` or `cPickle.UnpicklingError`.) However, there are a number of other exceptions that might be raised instead if there is a problem unpickling the data depending on exactly what the problem was. Some of the other exceptions that Python might raise when there is a problem unpickling data include (but are not limited to) the `AttributeError`, `EOFError`, `ImportError` or `IndexError` exceptions.

# Structuring a program for checkpointing

1. Initialise
2. Check for the existence of a previous checkpoint:
  1. If present, load it
3. Start processing loop:
  1. If there's a checkpoint file, retrieve state from file
  2. Process data
  3. Save state to checkpoint file
4. Final output

43

The most common use of pickling is for *checkpointing*.

Basically, you put all the variables that hold the current state of your program (i.e. all the variables whose values you would need if you wanted to restart the program whatever point it has just reached) into a tuple and then pickle that tuple. The file that contains this pickled data is your *checkpoint file*.

Each time you have done a certain amount of processing you dump the state of your program out to a checkpoint file. Then you restore from the checkpoint, i.e. load the pickled data from the checkpoint file – so you can be sure that the checkpoint actually did correctly store all the data it should have – and continue.

We examine this process in more detail in the “Python: Checkpointing” course:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonchckpt>

# Graphical output

Whilst there are Python modules that provide graphical capabilities, they are far too complex to be covered in this course, and probably overkill for most scientific computing anyway, so...

A more useful approach is to use something like **gnuplot** or **ploticus** to produce the graphical output for you.

44

There are two basic approaches to using such packages for your graphical output.

The first is to create a file containing the commands that tell the package what to do. You then save your data in one or more other files and call the package you want to use giving it the file of commands and the data file(s) as input. (See the next slide for more details.)

The second is to use – if it exists – a Python module that allows you to (more) directly interface with the package in question.

Fortunately, there are Python modules for both gnuplot and ploticus (although the ploticus module is not very sophisticated). These modules have been installed on PWF Linux, but are not part of standard Python and so may not be on other systems you use. They are, however freely available from the following sources:

Gnuplot.py:

<http://gnuplot-py.sourceforge.net/>

Python API for ploticus:

<http://www.srcc.lsu.edu/pyploticus.html>

In case you are unfamiliar with gnuplot and/or ploticus, you may wish to look at their home pages:

gnuplot: <http://www.gnuplot.info/>

ploticus: <http://ploticus.sourceforge.net/>

# DIY graphical output

1. Obtain (create, load, calculate, etc) the data to be graphically displayed
2. Write the data to a temporary file
3. Write the commands for your graphics package to another temporary file
4. Run your graphics package using the temporary files you've created

45

If you don't have the appropriate modules to communicate directly with your graphics package (or you can't install them or get them to work), then you can (probably) still use your graphics package with Python.

First, you get the data you want to graphically display (whether you are calculating that data, loading it from a file, or whatever).

Next, you write that data to a temporary file in whatever format your graphics package understands.

Then you write the commands that will control your graphics package to another temporary file (assuming your graphics package works by reading a file of commands that tell it what to do: gnuplot and ploticus can both work like this). If your graphics package doesn't support working in this way, check its documentation to see how it can be used to automatically read and plot data from a file (this is sometimes called "scripting" the package). If it provides no way of doing this, then you are probably out of luck! – use a more flexible graphics package.

Finally you run your graphics package, telling it to use the temporary files you've created for its input (the commands that tell it what to do, and the data on which those commands will operate). We cover how to call other programs from your Python script (the best way of doing this is to use the subprocess module) in the "Python: Operating System Access" course:

<http://www.training.cam.ac.uk/ucs/course/ucs-pythonopsys>

# Any questions?

46

If there are any questions about anything we've covered on either day of this course then I'll try to answer them now.



## Chapter 5

# Object Oriented Python Concepts

## CHAPTER 7



# More Abstraction

In the previous chapters, you looked at Python’s main built-in object types (numbers, strings, lists, tuples, and dictionaries); you peeked at the wealth of built-in functions and standard libraries; and you even created your own functions. Now, only one thing seems to be missing—making your own objects. And that’s what you do in this chapter.

You may wonder how useful this is. It might be cool to make your own kinds of objects, but what would you use them for? With all the dictionaries and sequences and numbers and strings available, can’t you just use them and make the functions do the job? Certainly, but making your own objects (and especially types or *classes* of objects) is a central concept in Python—so central, in fact, that Python is called an *object-oriented* language (along with Smalltalk, C++, Java, and many others). In this chapter, you learn how to make objects. You learn about polymorphism and encapsulation, methods and attributes, superclasses, and inheritance—you learn a lot. So let’s get started.

---

**Note** If you’re already familiar with the concepts of object-oriented programming, you probably know about *constructors*. Constructors will not be dealt with in this chapter; for a full discussion, see Chapter 9.

---

## The Magic of Objects

In object-oriented programming, the term *object* loosely means a collection of data (attributes) with a set of methods for accessing and manipulating those data. There are several reasons for using objects instead of sticking with global variables and functions. Some of the most important benefits of objects include the following:

- **Polymorphism:** You can use the same operations on objects of different classes, and they will work as if “by magic.”
- **Encapsulation:** You hide unimportant details of how objects work from the outside world.
- **Inheritance:** You can create specialized classes of objects from general ones.

In many presentations of object-oriented programming, the order of these concepts is different. Encapsulation and inheritance are presented first, and then they are used to model real-world objects. That’s all fine and dandy, but in my opinion, the most interesting feature of

object-oriented programming is polymorphism. It is also the feature that confuses most people (in my experience). Therefore I'll start with polymorphism, and try to show that this concept alone should be enough to make you like object-oriented programming.

## Polymorphism

The term *polymorphism* is derived from a Greek word meaning “having multiple forms.” Basically, that means that even if you don't know what kind of object a variable refers to, you may still be able to perform operations on it that will work differently depending on the type (or class) of the object. For example, assume that you are creating an online payment system for a commercial web site that sells food. Your program receives a “shopping cart” of goods from another part of the system (or other similar systems that may be designed in the future)—all you need to worry about is summing up the total and billing some credit card.

Your first thought may be to specify exactly how the goods must be represented when your program receives them. For example, you may want to receive them as tuples, like this:

```
('SPAM', 2.50)
```

If all you need is a descriptive tag and a price, this is fine. But it's not very flexible. Let's say that some clever person starts an auctioning service as part of the web site—where the price of an item is gradually reduced until someone buys it. It would be nice if the user could put the object in her shopping cart, proceed to the checkout (your part of the system), and just wait until the price was right before clicking the Pay button.

But that wouldn't work with the simple tuple scheme. For that to work, the object would need to check its current price (through some network magic) each time your code asked for the price—it couldn't be frozen like in a tuple. You can solve that by making a function:

```
# Don't do it like this...
def getPrice(object):
    if isinstance(object, tuple):
        return object[1]
    else:
        return magic_network_method(object)
```

---

**Note** The type/class checking and use of `isinstance` here is meant to illustrate a point—namely that type checking isn't generally a satisfactory solution. Avoid type checking if you possibly can. The function `isinstance` is described in the section “Investigating Inheritance,” later in this chapter.

---

In the preceding code, I use the function `isinstance` to find out whether the object is a tuple. If it is, its second element is returned; otherwise, some “magic” network method is called.

Assuming that the network stuff already exists, you've solved the problem—for now. But this still isn't very flexible. What if some clever programmer decides that she'll represent the price as a string with a hex value, stored in a dictionary under the key 'price'? No problem—you just update your function:

```
# Don't do it like this...
def getPrice(object):
    if isinstance(object, tuple):
        return object[1]
    elif isinstance(object, dict):
        return int(object['price'])
    else:
        return magic_network_method(object)
```

Now, surely you must have covered every possibility? But let's say someone decides to add a new type of dictionary with the price stored under a different key. What do you do now? You could certainly update `getPrice` again, but for how long could you continue doing that? Every time someone wanted to implement some priced object differently, you would need to reimplement your module. But what if you already sold your module and moved on to other, cooler projects—what would the client do then? Clearly, this is an inflexible and impractical way of coding the different behaviors.

So what do you do instead? You let the objects handle the operation themselves. It sounds really obvious, but think about how much easier things will get. Every new object type can retrieve or calculate its own price and return it to you—all you have to do is ask for it. And this is where polymorphism (and, to some extent, encapsulation) enters the scene.

## Polymorphism and Methods

You receive an object and have no idea of how it is implemented—it may have any one of many “shapes.” All you know is that you can ask for its price, and that's enough for you. The way you do that should be familiar:

```
>>> object.getPrice()
2.5
```

Functions that are bound to object attributes like this are called *methods*. You've already encountered them in the form of string, list, and dictionary methods. There, too, you saw some polymorphism:

```
>>> 'abc'.count('a')
1
>>> [1, 2, 'a'].count('a')
1
```

If you had a variable `x`, you wouldn't need to know whether it was a string or a list to call the `count` method—it would work regardless (as long as you supplied a single character as the argument).

Let's do an experiment. The standard library `random` contains a function called `choice` that selects a random element from a sequence. Let's use that to give your variable a value:

```
>>> from random import choice  
>>> x = choice(['Hello, world!', [1, 2, 'e', 'e', 4]])
```

After performing this, `x` can either contain the string '`Hello, world!`' or the list `[1, 2, 'e', 'e', 4]`—you don't know, and you don't have to worry about it. All you care about is how many times you find '`e`' in `x`, and you can find that out regardless of whether `x` is a list or a string. By calling the `count` method as before, you find out just that:

```
>>> x.count('e')  
2
```

In this case, it seems that the list won out. But the point is that you didn't need to check. Your only requirement was that `x` has a method called `count` that takes a single character as an argument and returned an integer. If someone else had made his own class of objects that had this method, it wouldn't matter to you—you could use his objects just as well as the strings and lists.

## Polymorphism Comes in Many Forms

Polymorphism is at work every time you can “do something” to an object without having to know exactly what kind of object it is. This doesn't apply only to methods—we've already used polymorphism a lot in the form of built-in operators and functions. Consider the following:

```
>>> 1+2  
3  
>>> 'Fish'+'license'  
'Fishlicense'
```

Here, the plus operator (`+`) works fine for both numbers (integers in this case) and strings (as well as other types of sequences). To illustrate the point, let's say you wanted to make a function called `add` that added two things together. You could simply define it like this (equivalent to, but less efficient than, the `add` function from the `operator` module):

```
def add(x, y):  
    return x+y
```

This would also work with many kinds of arguments:

```
>>> add(1, 2)  
3  
>>> add('Fish', 'license')  
'Fishlicense'
```

This might seem silly, but the point is that the arguments can be *anything that supports addition*.<sup>1</sup> If you want to write a function that prints a message about the length of an object, all that's required is that it *has* a length (that the `len` function will work on it):

```
def length_message(x):
    print "The length of", repr(x), "is", len(x)
```

As you can see, the function also uses `repr`, but `repr` is one of the grand masters of polymorphism—it works with anything. Let's see how:

```
>>> length_message('Fnord')
The length of 'Fnord' is 5
>>> length_message([1, 2, 3])
The length of [1, 2, 3] is 3
```

Many functions and operators are polymorphic—probably most of yours will be, too, even if you don't intend them to be. Just by using polymorphic functions and operators, the polymorphism “rubs off.” In fact, virtually the only thing you can do to destroy this polymorphism is to do explicit type checking with functions such as `type`, `isinstance`, and `issubclass`. If you can, you *really* should avoid destroying polymorphism this way. What matters should be that an object acts the way you want, not whether it is of the right type (or class).

---

**Note** The form of polymorphism discussed here, which is so central to the Python way of programming, is sometimes called “duck typing.” The term derives from the phrase, “If it quacks like a duck ...” For more information, see [http://en.wikipedia.org/wiki/Duck\\_typing](http://en.wikipedia.org/wiki/Duck_typing).

---

## Encapsulation

*Encapsulation* is the principle of hiding unnecessary details from the rest of the world. This may sound like polymorphism—there, too, you use an object without knowing its inner details. The two concepts are similar because they are both *principles of abstraction*. They both help you deal with the components of your program without caring about unnecessary detail, just as functions do.

But encapsulation isn't the same as polymorphism. Polymorphism enables you to call the methods of an object without knowing its class (type of object). Encapsulation enables you to use the object without worrying about how it's constructed. Does it still sound similar? Let's construct an example *with* polymorphism, but *without* encapsulation. Assume that you have a class called `OpenObject` (you learn how to create classes later in this chapter):

```
>>> o = OpenObject() # This is how we create objects...
>>> o.setName('Sir Lancelot')
>>> o.getName()
'Sir Lancelot'
```

---

1. Note that these objects need to support addition with each other. So calling `add(1, 'license')` would not work.

You create an object (by calling the class as if it were a function) and bind the variable `o` to it. You can then use the methods `setName` and `getName` (assuming that they are methods that are supported by the class `OpenObject`). Everything seems to be working perfectly. However, let's assume that `o` stores its name in the global variable `globalName`:

```
>>> globalName  
'Sir Lancelot'
```

This means that you need to worry about the contents of `globalName` when you use instances (objects) of the class `OpenObject`. In fact, you must make sure that no one changes it:

```
>>> globalName = 'Sir Gumby'  
>>> o.getName()  
'Sir Gumby'
```

Things get even more problematic if you try to create more than one `OpenObject` because they will all be messing with the same variable:

```
>>> o1 = OpenObject()  
>>> o2 = OpenObject()  
>>> o1.setName('Robin Hood')  
>>> o2.getName()  
'Robin Hood'
```

As you can see, setting the name of one automatically sets the name of the other—not exactly what you want.

Basically, you want to treat objects as abstract. When you call a method, you don't want to worry about anything else, such as not disturbing global variables. So how can you "encapsulate" the name within the object? No problem. You make it an *attribute*.

Attributes are variables that are a part of the object, just like methods; actually, methods are almost like attributes bound to functions. (You'll see an important difference between methods and functions in the section "Attributes, Functions, and Methods," later in this chapter.) If you rewrite the class to use an attribute instead of a global variable, and you rename it `ClosedObject`, it works like this:

```
>>> c = ClosedObject()  
>>> c.setName('Sir Lancelot')  
>>> c.getName()  
'Sir Lancelot'
```

So far, so good. But for all you know, this could still be stored in a global variable. Let's make another object:

```
>>> r = ClosedObject()  
>>> r.setName('Sir Robin')  
r.getName()  
'Sir Robin'
```

Here, you can see that the new object has its name set properly, which is probably what you expected. But what has happened to the first object now?

```
>>> c.getName()  
'Sir Lancelot'
```

The name is still there! This is because the object has its own *state*. The state of an object is described by its attributes (like its name, for example). The methods of an object may change these attributes. So it's like lumping together a bunch of functions (the methods) and giving them access to some variables (the attributes) where they can keep values stored between function calls.

You'll see even more details on Python's encapsulation mechanisms in the section "Privacy Revisited," later in the chapter.

## Inheritance

Inheritance is another way of dealing with laziness (in the positive sense). Programmers want to avoid typing the same code more than once. We avoided that earlier by making functions, but now I will address a more subtle problem. What if you have a class already, and you want to make one that is very similar? Perhaps one that adds only a few methods? When making this new class, you don't want to need to copy all the code from the old one over to the new one.

For example, you may already have a class called `Shape`, which knows how to draw itself on the screen. Now you want to make a class called `Rectangle`, which *also* knows how to draw itself on the screen, but which can, in addition, calculate its own area. You wouldn't want to do all the work of making a new `draw` method when `Shape` has one that works just fine. So what do you do? You let `Rectangle` *inherit* the methods from `Shape`. You can do this in such a way that when `draw` is called on a `Rectangle` object, the method from the `Shape` class is called automatically (see the section "Specifying a Superclass," later in this chapter).

## Classes and Types

By now, you're getting a feeling for what classes are—or you *may* be getting impatient for me to tell you how to make the darn things. Before jumping into the technicalities, let's have a look at what a class is, and how it is different from (or similar to) a type.

### What Is a Class, Exactly?

I've been throwing around the word *class* a lot, using it more or less synonymously with words such as *kind* or *type*. In many ways that's exactly what a class is—a kind of object. All objects *belong* to a class and are said to be *instances* of that class.

So, for example, if you look outside your window and see a bird, that bird is an instance of the class "birds." This is a very general (abstract) class that has several *subclasses*; your bird might belong to the subclass "larks." You can think of the class "birds" as the set of all birds, while the class "larks" is just a subset of that. When the objects belonging to one class form a subset of the objects belonging to another class, the first is called a *subclass* of the second. Thus, "larks" is a subclass of "birds." Conversely, "birds" is a *superclass* of "larks."

**Note** In everyday speech, we denote classes of objects with plural nouns such as “birds” and “larks.” In Python, it is customary to use singular, capitalized nouns such as `Bird` and `Lark`.

---

When stated like this, subclasses and superclasses are easy to understand. But in object-oriented programming, the subclass relation has important implications because a class is defined by what methods it supports. All the instances of a class have these methods, so all the instances of all *subclasses* must *also* have them. Defining subclasses is then only a matter of defining *more* methods (or, perhaps, overriding some of the existing ones).

For example, `Bird` might supply the method `fly`, while `Penguin` (a subclass of `Bird`) might add the method `eatFish`. When making a `Penguin` class, you would probably also want to *override* a method of the superclass, namely the `fly` method. In a `Penguin` instance, this method should either do nothing, or possibly raise an exception (see Chapter 8), given that penguins can’t fly.

---

**Note** In older versions of Python, there was a sharp distinction between types and classes. Built-in objects had types; your custom objects had classes. You could create classes, but not types. In recent versions of Python, things are starting to change. The division between basic types and classes is blurring. You can now make subclasses (or subtypes) of the built-in types, and the types are behaving more like classes. Chances are you won’t notice this change much until you become more familiar with the language. If you’re interested, you can find more information on the topic in Chapter 9.

---

## Making Your Own Classes

Finally, you get to make your own classes! Here is a simple example:

```
__metaclass__ = type # Make sure we get new style classes

class Person:

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

    def greet(self):
        print "Hello, world! I'm %s." % self.name
```

---

**Note** There is a difference between so-called *old-style* and *new-style* classes. There is really no reason to use the old-style classes anymore, except that they're what you get by default in Python versions prior to 3.0. To get new-style classes, you should place the assignment `__metaclass__ = type` at the beginning of your script or module. (I may not explicitly include this statement in every example.) There are also other solutions, such as subclassing a new-style class (for example, `object`). You learn more about subclassing in a minute. In Python 3.0, there is no need to worry about this, as old-style classes don't exist there. You find more information about this in Chapter 9.

---

This example contains three method definitions, which are like function definitions except that they are written inside a `class` statement. `Person` is, of course, the name of the class. The `class` statement creates its own namespace where the functions are defined. (See the section “The Class Namespace” later in this chapter.) All this seems fine, but you may wonder what this `self` parameter is. It refers to the object itself. And what object is that? Let's make a couple of instances and see:

```
>>> foo = Person()
>>> bar = Person()
>>> foo.setName('Luke Skywalker')
>>> bar.setName('Anakin Skywalker')
>>> foo.greet()
Hello, world! I'm Luke Skywalker.
>>> bar.greet()
Hello, world! I'm Anakin Skywalker.
```

Okay, so this example may be a bit obvious, but perhaps it clarifies what `self` is. When I call `setName` and `greet` on `foo`, `foo` itself is automatically passed as the first parameter in each case—the parameter that I have so fittingly called `self`. You may, in fact, call it whatever you like, but because it is always the object itself, it is almost always called `self`, by convention.

It should be obvious why `self` is useful, and even necessary here. Without it, none of the methods would have access to the object itself—the object whose attributes they are supposed to manipulate. As before, the attributes are also accessible from the outside:

```
>>> foo.name
'Luke Skywalker'
>>> bar.name = 'Yoda'
>>> bar.greet()
Hello, world! I'm Yoda.
```

---

**Tip** Another way of viewing this is that `foo.greet()` is simply a convenient way of writing `Person.greet(foo)`, if you know that `foo` is an instance of `Person`.

---

## Attributes, Functions, and Methods

The `self` parameter (mentioned in the previous section) is, in fact, what distinguishes methods from functions. Methods (or, more technically, *bound* methods) have their first parameter bound to the instance they belong to, so you don't have to supply it. While you can certainly bind an attribute to a plain function, it won't have that special `self` parameter:

```
>>> class Class:  
    def method(self):  
        print 'I have a self!'  
  
>>> def function():  
    print "I don't..."  
  
>>> instance = Class()  
>>> instance.method()  
I have a self!  
>>> instance.method = function  
>>> instance.method()  
I don't...
```

Note that the `self` parameter is not dependent on calling the method the way I've done until now, as `instance.method`. You're free to use another variable that refers to the same method:

```
>>> class Bird:  
    song = 'Squaawk!'  
    def sing(self):  
        print self.song  
  
>>> bird = Bird()  
>>> bird.sing()  
Squaawk!  
>>> birdsong = bird.sing  
>>> birdsong()  
Squaawk!
```

Even though the last method call looks exactly like a function call, the variable `birdsong` refers to the bound method `bird.sing`, which means that it still has access to the `self` parameter (that is, it is still bound to the same instance of the class).

## Privacy Revisited

By default, you can access the attributes of an object from the “outside.” Let's revisit the example from the earlier discussion on encapsulation:

```
>>> c.name  
'Sir Lancelot'  
>>> c.name = 'Sir Gumby'
```

```
>>> c.getName()
'Sir Gumby'
```

Some programmers are okay with this, but some (like the creators of Smalltalk, a language where attributes of an object are accessible only to the methods of the same object) feel that it breaks with the principle of encapsulation. They believe that the state of the object should be *completely hidden* (inaccessible) to the outside world. You might wonder why they take such an extreme stand. Isn't it enough that each object manages its own attributes? Why should you hide them from the world? After all, if you just used the `name` attribute directly on `ClosedObject` (the class of `c` in this case), you wouldn't need to make the `setName` and `getName` methods.

The point is that other programmers may not know (and perhaps shouldn't know) what's going on inside your object. For example, `ClosedObject` may send an email message to some administrator every time an object changes its name. This could be part of the `setName` method. But what happens when you set `c.name` directly? Nothing happens—no email message is sent. To avoid this sort of thing, you have *private* attributes. These are attributes that are not accessible outside the object; they are accessible only through *accessor* methods, such as `getName` and `setName`.

---

**Note** In Chapter 9, you learn about *properties*, a powerful alternative to accessors.

---

Python doesn't support privacy directly, but relies on the programmer to know when it is safe to modify an attribute from the outside. After all, you should know how to use an object before using that object. It *is*, however, possible to achieve something like private attributes with a little trickery.

To make a method or attribute private (inaccessible from the outside), simply start its name with two underscores:

```
class Secretive:

    def __inaccessible(self):
        print "Bet you can't see me..."

    def accessible(self):
        print "The secret message is:"
        self.__inaccessible()
```

Now `__inaccessible` is inaccessible to the outside world, while it can still be used inside the class (for example, from `accessible`):

```
>>> s = Secretive()
>>> s.__inaccessible()
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in ?
    s.__inaccessible()
AttributeError: Secretive instance has no attribute '__inaccessible'
```

```
>>> s.accessible()
The secret message is:
Bet you can't see me...
```

Although the double underscores are a bit strange, this seems like a standard private method, as found in other languages. What's not so standard is what actually happens. Inside a class definition, all names beginning with a double underscore are “translated” by adding a single underscore and the class name to the beginning:

```
>>> Secretive._Secretive_inaccessible
<unbound method Secretive._inaccessible>
```

If you know how this works behind the scenes, it is still possible to access private methods outside the class, even though you're not supposed to:

```
>>> s._Secretive_inaccessible()
Bet you can't see me...
```

So, in short, you can't be sure that others won't access the methods and attributes of your objects, but this sort of name-mangling is a pretty strong signal that they *shouldn't*.

If you don't want the name-mangling effect, but you still want to send a signal for other objects to stay away, you can use a *single* initial underscore. This is mostly just a convention, but has some practical effects. For example, names with an initial underscore aren't imported with starred imports (`from module import *`).<sup>2</sup>

## The Class Namespace

The following two statements are (more or less) equivalent:

```
def foo(x): return x*x
foo = lambda x: x*x
```

Both create a function that returns the square of its argument, and both bind the variable `foo` to that function. The name `foo` may be defined in the global (module) scope, or it may be local to some function or method. The same thing happens when you define a class: all the code in the `class` statement is executed in a special namespace—the *class namespace*. This namespace is accessible later by all members of the class. Not all Python programmers know that class definitions are simply code sections that are executed, but it can be useful information. For example, you aren't restricted to `def` statements inside the class definition block:

```
>>> class C:
    print 'Class C being defined...'

Class C being defined...
>>>
```

---

2. Some languages support several *degrees* of privacy for its member variables (attributes). Java, for example, has four different levels. Python doesn't really have equivalent privacy support, although single and double initial underscores do to some extent give you two levels of privacy.

Okay, that was a bit silly. But consider the following:

```
class MemberCounter:  
    members = 0  
    def init(self):  
        MemberCounter.members += 1  
  
>>> m1 = MemberCounter()  
>>> m1.init()  
>>> MemberCounter.members  
1  
>>> m2 = MemberCounter()  
>>> m2.init()  
>>> MemberCounter.members  
2
```

In the preceding code, a variable is defined in the class scope, which can be accessed by all the members (instances), in this case to count the number of class members. Note the use of `init` to initialize all the instances: I'll automate that (that is, turn it into a proper constructor) in Chapter 9.

This class scope variable is accessible from every instance as well, just as methods are:

```
>>> m1.members  
2  
>>> m2.members  
2
```

What happens when you rebind the `members` attribute in an instance?

```
>>> m1.members = 'Two'  
>>> m1.members  
'Two'  
>>> m2.members  
2
```

The new `members` value has been written into an attribute in `m1`, shadowing the class-wide variable. This mirrors the behavior of local and global variables in functions, as discussed in the sidebar “The Problem of Shadowing” in Chapter 6.

## Specifying a Superclass

As I discussed earlier in the chapter, subclasses expand on the definitions in their superclasses. You indicate the superclass in a `class` statement by writing it in parentheses after the class name:

```
class Filter:  
    def init(self):  
        self.blocked = []  
    def filter(self, sequence):  
        return [x for x in sequence if x not in self.blocked]
```

```
class SPAMFilter(Filter): # SPAMFilter is a subclass of Filter
    def init(self): # Overrides init method from Filter superclass
        self.blocked = ['SPAM']
```

Filter is a general class for filtering sequences. Actually it doesn't filter out anything:

```
>>> f = Filter()
>>> f.init()
>>> f.filter([1, 2, 3])
[1, 2, 3]
```

The usefulness of the Filter class is that it can be used as a base class (superclass) for other classes, such as SPAMFilter, which filters out 'SPAM' from sequences:

```
>>> s = SPAMFilter()
>>> s.init()
>>> s.filter(['SPAM', 'SPAM', 'SPAM', 'SPAM', 'eggs', 'bacon', 'SPAM'])
['eggs', 'bacon']
```

Note two important points in the definition of SPAMFilter:

- I override the definition of `init` from Filter by simply providing a new definition.
- The definition of the `filter` method carries over (is inherited) from Filter, so you don't need to write the definition again.

The second point demonstrates why inheritance is useful: I can now make a number of different filter classes, all subclassing Filter, and for each one I can simply use the `filter` method I have already implemented. Talk about useful laziness . . .

## Investigating Inheritance

If you want to find out whether a class is a subclass of another, you can use the built-in method `issubclass`:

```
>>> issubclass(SPAMFilter, Filter)
True
>>> issubclass(Filter, SPAMFilter)
False
```

If you have a class and want to know its base classes, you can access its special attribute `__bases__`:

```
>>> SPAMFilter.__bases__
(<class '__main__.Filter' at 0x171e40>,)
>>> Filter.__bases__
()
```

In a similar manner, you can check whether an object is an instance of a class by using `isinstance`:

```
>>> s = SPAMFilter()
>>> isinstance(s, SPAMFilter)
True
>>> isinstance(s, Filter)
True
>>> isinstance(s, str)
False
```

---

**Note** Using `isinstance` is usually not good practice. Relying on polymorphism is almost always better.

---

As you can see, `s` is a (direct) member of the class `SPAMFilter`, but it is also an indirect member of `Filter` because `SPAMFilter` is a subclass of `Filter`. Another way of putting it is that all `SPAMFilters` are `Filters`. As you can see in the preceding example, `isinstance` also works with types, such as the string type (`str`).

If you just want to find out which class an object belongs to, you can use the `__class__` attribute:

```
>>> s.__class__
<class '__main__.SPAMFilter' at 0x1707c0>
```

---

**Note** If you have a new-style class, either by setting `__metaclass__ = type` or subclassing `object`, you could also use `type(s)` to find the class of your instance.

---

## Multiple Superclasses

I'm sure you noticed a small detail in the previous section that may have seemed odd: the plural form in `__bases__`. I said you could use it to find the base classes of a class, which implies that it may have more than one. This is, in fact, the case. To show how it works, let's create a few classes:

```
class Calculator:
    def calculate(self, expression):
        self.value = eval(expression)

class Talker:
    def talk(self):
        print 'Hi, my value is', self.value

class TalkingCalculator(Calculator, Talker):
    pass
```

The subclass (`TalkingCalculator`) does nothing by itself; it inherits all its behavior from its superclasses. The point is that it inherits both `calculate` from `Calculator` and `talk` from `Talker`, making it a talking calculator:

```
>>> tc = TalkingCalculator()
>>> tc.calculate('1+2*3')
>>> tc.talk()
Hi, my value is 7
```

This is called *multiple inheritance*, and can be a very powerful tool. However, unless you know you need multiple inheritance, you may want to stay away from it, as it can, in some cases, lead to unforeseen complications.

If you are using multiple inheritance, there is one thing you should look out for: if a method is implemented differently by two or more of the superclasses (that is, you have two different methods with the same name), you must be careful about the order of these superclasses (in the class statement). The methods in the earlier classes *override* the methods in the later ones. So if the `Calculator` class in the preceding example had a method called `talk`, it would override (and make inaccessible) the `talk` method of the `Talker`. Reversing their order, like this:

```
class TalkingCalculator(Talker, Calculator): pass
```

would make the `talk` method of the `Talker` accessible. If the superclasses share a common superclass, the order in which the superclasses are visited while looking for a given attribute or method is called the *method resolution order* (MRO), and follows a rather complicated algorithm. Luckily, it works very well, so you probably don't need to worry about it.

## Interfaces and Introspection

The “interface” concept is related to polymorphism. When you handle a polymorphic object, you only care about its interface (or “protocol”—the methods and attributes known to the world). In Python, you don't explicitly specify which methods an object needs to have to be acceptable as a parameter. For example, you don't write interfaces explicitly (as you do in Java); you just assume that an object can do what you ask it to do. If it can't, the program will fail.

Usually, you simply require that objects conform to a certain interface (in other words, implement certain methods), but if you want to, you can be quite flexible in your demands. Instead of just calling the methods and hoping for the best, you can check whether the required methods are present, and if not, perhaps do something else:

```
>>> hasattr(tc, 'talk')
True
>>> hasattr(tc, 'fnord')
False
```

In the preceding code, you find that `tc` (a `TalkingCalculator`, as described earlier in this chapter) has the attribute `talk` (which refers to a method), but not the attribute `fnord`. If you wanted to, you could even check whether the `talk` attribute was callable:

```
>>> callable(getattr(tc, 'talk', None))
True
```

```
>>> callable(getattr(tc, 'fnord', None))
False
```

---

**Note** The function `callable` is no longer available in Python 3.0. Instead of `callable(x)`, you can use `hasattr(x, '__call__')`.

---

Note that instead of using `hasattr` in an `if` statement and accessing the attribute directly, I'm using `getattr`, which allows me to supply a default value (in this case `None`) that will be used if the attribute is not present. I then use `callable` on the returned object.

---

**Note** The inverse of `getattr` is `setattr`, which can be used to set the attributes of an object:

```
>>> setattr(tc, 'name', 'Mr. Gumby')
>>> tc.name
'Mr. Gumby'
```

---

If you want to see all the values stored in an object, you can examine its `__dict__` attribute. And if you *really* want to find out what an object is made of, you should take a look at the `inspect` module. It is meant for fairly advanced users who want to make object browsers (programs that enable you to browse Python objects in a graphical manner) and other similar programs that require such functionality. For more information on exploring objects and modules, see the section “Exploring Modules” in Chapter 10.

## Some Thoughts on Object-Oriented Design

Many books have been written about object-oriented program design, and although that's not the focus of this book, I'll give you some pointers:

- Gather what belongs together. If a function manipulates a global variable, the two of them might be better off in a class, as an attribute and a method.
- Don't let objects become too intimate. Methods should mainly be concerned with the attributes of their own instance. Let other instances manage their own state.
- Go easy on the inheritance, *especially* multiple inheritance. Inheritance is useful at times, but can make things unnecessarily complex in some cases. And multiple inheritance can be very difficult to get right and even harder to debug.
- Keep it simple. Keep your methods small. As a rule of thumb, it should be possible to read (and understand) most of your methods in, say, 30 seconds. For the rest, try to keep them shorter than one page or screen.

When determining which classes you need and which methods they should have, you may try something like this:

1. Write down a description of your problem (what should the program do?). Underline all the nouns, verbs, and adjectives.
2. Go through the nouns, looking for potential classes.
3. Go through the verbs, looking for potential methods.
4. Go through the adjectives, looking for potential attributes.
5. Allocate methods and attributes to your classes.

Now you have a first sketch of an *object-oriented model*. You may also want to think about what responsibilities and relationships (such as inheritance or cooperation) the classes and objects will have. To refine your model, you can do the following:

1. Write down (or dream up) a set of *use cases*—scenarios of how your program may be used. Try to cover all the functionality.
2. Think through every use case step by step, making sure that everything you need is covered by your model. If something is missing, add it. If something isn’t quite right, change it. Continue until you are satisfied.

When you have a model you think will work, you can start hacking away. Chances are you’ll need to revise your model or revise parts of your program. Luckily, that’s easy in Python, so don’t worry about it. Just dive in. (If you would like some more guidance in the ways of object-oriented programming, check out the list of suggested books in Chapter 19.)

## A Quick Summary

This chapter has given you more than just information about the Python language; it has introduced you to several concepts that may have been completely foreign to you. Here’s a summary:

**Objects:** An object consists of attributes and methods. An attribute is merely a variable that is part of an object, and a method is more or less a function that is stored in an attribute. One difference between (bound) methods and other functions is that methods always receive the object they are part of as their first argument, usually called `self`.

**Classes:** A class represents a set (or kind) of objects, and every object (instance) has a class. The class’s main task is to define the methods its instances will have.

**Polymorphism:** Polymorphism is the characteristic of being able to treat objects of different types and classes alike—you don’t need to know which class an object belongs to in order to call one of its methods.

**Encapsulation:** Objects may hide (or encapsulate) their internal state. In some languages, this means that their state (their attributes) is available only through their methods. In

In Python, all attributes are publicly available, but programmers should still be careful about accessing an object’s state directly, since they might unwittingly make the state inconsistent in some way.

**Inheritance:** One class may be the subclass of one or more other classes. The subclass then inherits all the methods of the superclasses. You can use more than one superclass, and this feature can be used to compose orthogonal (independent and unrelated) pieces of functionality. A common way of implementing this is using a core superclass along with one or more *mix-in* superclasses.

**Interfaces and introspection:** In general, you don’t want to prod an object too deeply. You rely on polymorphism, and call the methods you need. However, if you want to find out what methods or attributes an object has, there are functions that will do the job for you.

**Object-oriented design:** There are many opinions about how (or whether!) to do object-oriented design. No matter where you stand on the issue, it’s important to understand your problem thoroughly, and to create a design that is easy to understand.

## New Functions in This Chapter

Function	Description
<code>callable(object)</code>	Determines if the object is callable (such as a function or a method)
<code>getattr(object, name[, default])</code>	Gets the value of an attribute, optionally providing a default
<code>hasattr(object, name)</code>	Determines if the object has the given attribute
<code>isinstance(object, class)</code>	Determines if the object is an instance of the class
<code>issubclass(A, B)</code>	Determines if A is a subclass of B
<code>random.choice(sequence)</code>	Chooses a random element from a nonempty sequence
<code>setattr(object, name, value)</code>	Sets the given attribute of the object to value
<code>type(object)</code>	Returns the type of the object

## What Now?

You’ve learned a lot about creating your own objects and how useful that can be. Before diving headlong into the magic of Python’s special methods (Chapter 9), let’s take a breather with a little chapter about exception handling.

# Chapter 6

## Python Magic

## CHAPTER 9



# Magic Methods, Properties, and Iterators

In Python, some names are spelled in a peculiar manner, with two leading and two trailing underscores. You have already encountered some of these (`__future__`, for example). This spelling signals that the name has a special significance—you should never invent such names for your own programs. One very prominent set of such names in the language consists of the *magic* (or special) method names. If your object implements one of these methods, that method will be called under specific circumstances (exactly which will depend on the name) by Python. There is rarely any need to call these methods directly.

This chapter deals with a few important magic methods (most notably the `__init__` method and some methods dealing with item access, allowing you to create sequences or mappings of your own). It also tackles two related topics: properties (dealt with through magic methods in previous versions of Python, but now handled by the `property` function), and iterators (which use the magic method `__iter__` to enable them to be used in `for` loops). You'll find a meaty example at the end of the chapter, which uses some of the things you have learned so far to solve a fairly difficult problem.

## Before We Begin . . .

A while ago (in version 2.2), the way Python objects work changed quite a bit. This change has several consequences, most of which won't be important to you as a beginning Python programmer.<sup>1</sup> One thing is worth noting, though: even if you're using a recent version of Python, some features (such as properties and the `super` function) won't work on "old-style" classes. To make your classes "new-style," you should either put the assignment `__metaclass__ = type` at the top of your modules (as mentioned in Chapter 7) or (directly or indirectly) subclass the built-in class (or, actually, `type`) `object` (or some other new-style class). Consider the following two classes:

```
class NewStyle(object):
    more_code_here
```

---

1. For a thorough description of the differences between old-style and new-style classes, see Chapter 8 in Alex Martelli's *Python in a Nutshell* (O'Reilly & Associates, 2003).

```
class OldStyle:  
    more_code_here
```

Of these two, `NewStyle` is a new-style class; `OldStyle` is an old-style class. If the file began with `__metaclass__ = type`, though, both classes would be new-style.

---

**Note** You can also assign to the `__metaclass__` variable in the class scope of your class. That would set the metaclass of only that class. Metaclasses are the classes of other classes (or types)—a rather advanced topic. For more information about metaclasses, take a look at the (somewhat technical) article called “Unifying types and classes in Python 2.2” by Guido van Rossum (<http://python.org/2.2/descrintro.html>), or do a web search for the term “python metaclasses.”

---

I do not explicitly set the metaclass (or subclass `object`) in all the examples in this book. However, if you do not specifically need to make your programs compatible with old versions of Python, I advise you to make all your classes new-style, and consistently use features such as the `super` function (described in the section “Using the `super` Function,” later in this chapter).

---

**Note** There are no “old-style” classes in Python 3.0, and no need to explicitly subclass `object` or set the metaclass to `type`. All classes will implicitly be subclasses of `object`—directly, if you don’t specify a super-class, or indirectly otherwise.

---

## Constructors

The first magic method we’ll take a look at is the constructor. In case you have never heard the word *constructor* before, it’s basically a fancy name for the kind of initializing method I have already used in some of the examples, under the name `init`. What separates constructors from ordinary methods, however, is that the constructors are called automatically right after an object has been created. Thus, instead of doing what I’ve been doing up until now:

```
>>> f = FooBar()  
>>> f.init()
```

constructors make it possible to simply do this:

```
>>> f = FooBar()
```

Creating constructors in Python is really easy; simply change the `init` method's name from the plain old `init` to the magic version, `__init__`:

```
class FooBar:  
    def __init__(self):  
        self.somevar = 42  
  
>>> f = FooBar()  
>>> f.somevar  
42
```

Now, that's pretty nice. But you may wonder what happens if you give the constructor some parameters to work with. Consider the following:

```
class FooBar:  
    def __init__(self, value=42):  
        self.somevar = value
```

How do you think you could use this? Because the parameter is optional, you certainly could go on like nothing had happened. But what if you wanted to use it (or you hadn't made it optional)? I'm sure you've guessed it, but let me show you anyway:

```
>>> f = FooBar('This is a constructor argument')  
>>> f.somevar  
'This is a constructor argument'
```

Of all the magic methods in Python, `__init__` is quite certainly the one you'll be using the most.

---

**Note** Python has a magic method called `__del__`, also known as the *destructor*. It is called just before the object is destroyed (garbage-collected), but because you cannot really know when (or if) this happens, I advise you to stay away from `__del__` if at all possible.

---

## Overriding Methods in General, and the Constructor in Particular

In Chapter 7, you learned about inheritance. Each class may have one or more superclasses, from which they inherit behavior. If a method is called (or an attribute is accessed) on an instance of class B and it is not found, its superclass A will be searched. Consider the following two classes:

```
class A:  
    def hello(self):  
        print "Hello, I'm A."
```

```
class B(A):
    pass
```

Class A defines a method called `hello`, which is inherited by class B. Here is an example of how these classes work:

```
>>> a = A()
>>> b = B()
>>> a.hello()
Hello, I'm A.
>>> b.hello()
Hello, I'm A.
```

Because B does not define a `hello` method of its own, the original message is printed when `hello` is called.

One basic way of adding functionality in the subclass is simply to add methods. However, you may want to customize the inherited behavior by overriding some of the superclass's methods. For example, it is possible for B to override the `hello` method. Consider this modified definition of B:

```
class B(A):
    def hello(self):
        print "Hello, I'm B."
```

Using this definition, `b.hello()` will give a different result:

```
>>> b = B()
>>> b.hello()
Hello, I'm B.
```

Overriding is an important aspect of the inheritance mechanism in general, and may be especially important for constructors. Constructors are there to initialize the state of the newly constructed object, and most subclasses will need to have initialization code of their own, in addition to that of the superclass. Even though the mechanism for overriding is the same for all methods, you will most likely encounter one particular problem more often when dealing with constructors than when overriding ordinary methods: if you override the constructor of a class, you need to call the constructor of the superclass (the class you inherit from) or risk having an object that isn't properly initialized.

Consider the following class, `Bird`:

```
class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print 'Aaaah...'
            self.hungry = False
        else:
            print 'No, thanks!'
```

This class defines one of the most basic capabilities of all birds: eating. Here is an example of how you might use it:

```
>>> b = Bird()
>>> b.eat()
Aaaah...
>>> b.eat()
No, thanks!
```

As you can see from this example, once the bird has eaten, it is no longer hungry. Now consider the subclass `SongBird`, which adds singing to the repertoire of behaviors:

```
class SongBird(Bird):
    def __init__(self):
        self.sound = 'Squawk!'
    def sing(self):
        print self.sound
```

The `SongBird` class is just as easy to use as `Bird`:

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
```

Because `SongBird` is a subclass of `Bird`, it inherits the `eat` method, but if you try to call it, you'll discover a problem:

```
>>> sb.eat()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "birds.py", line 6, in eat
    if self.hungry:
AttributeError: SongBird instance has no attribute 'hungry'
```

The exception is quite clear about what's wrong: the `SongBird` has no attribute called `hungry`. Why should it? In `SongBird`, the constructor is overridden, and the new constructor doesn't contain any initialization code dealing with the `hungry` attribute. To rectify the situation, the `SongBird` constructor must call the constructor of its superclass, `Bird`, to make sure that the basic initialization takes place. There are basically two ways of doing this: by calling the unbound version of the superclass's constructor or by using the `super` function. In the next two sections, I explain both techniques.

## Calling the Unbound Superclass Constructor

The approach described in this section is, perhaps, mainly of historical interest. With current versions of Python, using the `super` function (as explained in the following section) is clearly the way to go (and with Python 3.0, it will be even more so). However, much existing code uses the approach described in this section, so you need to know about it. Also, it can be quite instructive—it's a nice example of the difference between bound and unbound methods.

Now, let's get down to business. If you find the title of this section a bit intimidating, relax. Calling the constructor of a superclass is, in fact, very easy (and useful). I'll start by giving you the solution to the problem posed at the end of the previous section:

```
class SongBird(Bird):
    def __init__(self):
        Bird.__init__(self)
        self.sound = 'Squawk!'
    def sing(self):
        print self.sound
```

Only one line has been added to the `SongBird` class, containing the code `Bird.__init__(self)`. Before I explain what this really means, let me just show you that this really works:

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
>>> sb.eat()
No, thanks!
```

But why does this work? When you retrieve a method from an instance, the `self` argument of the method is automatically *bound* to the instance (a so-called bound method). You've seen several examples of that. However, if you retrieve the method directly from the class (such as in `Bird.__init__`), there is no instance to which to bind. Therefore, you are free to supply any `self` you want to. Such a method is called *unbound*, which explains the title of this section.

By supplying the current instance as the `self` argument to the unbound method, the songbird gets the full treatment from its superclass's constructor (which means that it has its hungry attribute set).

## Using the `super` Function

If you're not stuck with an old version of Python, the `super` function is really the way to go. It works only with new-style classes, but you should be using those anyway. It is called with the current class and instance as its arguments, and any method you call on the returned object will be fetched from the superclass rather than the current class. So, instead of using `Bird` in the `SongBird` constructor, you can use `super(SongBird, self)`. Also, the `__init__` method can be called in a normal (bound) fashion.

---

**Note** In Python 3.0, `super` can be called without any arguments, and will do its job as if "by magic."

---

The following is an updated version of the bird example:

```
__metaclass__ = type # super only works with new-style classes

class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print 'Aaaah...'
            self.hungry = False
        else:
            print 'No, thanks!'

class SongBird(Bird):
    def __init__(self):
        super(SongBird, self).__init__()
        self.sound = 'Squawk!'
    def sing(self):
        print self.sound
```

This new-style version works just like the old-style one:

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
>>> sb.eat()
No, thanks!
```

## WHAT'S SO SUPER ABOUT SUPER?

In my opinion, the `super` function is more intuitive than calling unbound methods on the superclass directly, but that is not its only strength. The `super` function is actually quite smart, so even if you have multiple superclasses, you only need to use `super` once (provided that all the superclass constructors also use `super`). Also, some obscure situations that are tricky when using old-style classes (for example, when two of your superclasses share a superclass) are automatically dealt with by new-style classes and `super`. You don't need to understand exactly how it works internally, but you should be aware that, in most cases, it is clearly superior to calling the unbound constructors (or other methods) of your superclasses.

So, what does `super` return, really? Normally, you don't need to worry about it, and you can just pretend it returns the superclass you need. What it actually does is return a *super object*, which will take care of method resolution for you. When you access an attribute on it, it will look through all your superclasses (and super-superclasses, and so forth until it finds the attribute (or raises an `AttributeError`).

## Item Access

Although `__init__` is by far the most important special method you'll encounter, many others are available to enable you to achieve quite a lot of cool things. One useful set of magic methods described in this section allows you to create objects that behave like sequences or mappings.

The basic sequence and mapping protocol is pretty simple. However, to implement all the functionality of sequences and mappings, there are many magic methods to implement. Luckily, there are some shortcuts, but I'll get to that.

---

**Note** The word *protocol* is often used in Python to describe the rules governing some form of behavior. This is somewhat similar to the notion of *interfaces* mentioned in Chapter 7. The protocol says something about which methods you should implement and what those methods should do. Because polymorphism in Python is based on only the object's behavior (and not on its *ancestry*, for example, its class or superclass, and so forth), this is an important concept: where other languages might require an object to belong to a certain class or to implement a certain interface, Python often simply requires it to follow some given protocol. So, to *be* a sequence, all you have to do is follow the sequence protocol.

---

## The Basic Sequence and Mapping Protocol

Sequences and mappings are basically collections of *items*. To implement their basic behavior (protocol), you need two magic methods if your objects are immutable, or four if they are mutable:

`__len__(self)`: This method should return the number of items contained in the collection. For a sequence, this would simply be the number of elements. For a mapping, it would be the number of key-value pairs. If `__len__` returns zero (and you don't implement `__nonzero__`, which overrides this behavior), the object is treated as *false* in a Boolean context (as with empty lists, tuples, strings, and dictionaries).

`__getitem__(self, key)`: This should return the value corresponding to the given key. For a sequence, the key should be an integer from zero to  $n-1$  (or, it could be negative, as noted later), where  $n$  is the length of the sequence. For a mapping, you could really have any kind of keys.

`__setitem__(self, key, value)`: This should store `value` in a manner associated with `key`, so it can later be retrieved with `__getitem__`. Of course, you define this method only for mutable objects.

`__delitem__(self, key)`: This is called when someone uses the `del` statement on a part of the object, and should delete the element associated with `key`. Again, only mutable objects (and not all of them—only those for which you want to let items be removed) should define this method.

Some extra requirements are imposed on these methods:

- For a sequence, if the key is a negative integer, it should be used to count from the end. In other words, treat  $x[-n]$  the same as  $x[len(x)-n]$ .
- If the key is of an inappropriate type (such as a string key used on a sequence), a `TypeError` may be raised.
- If the index of a sequence is of the right type, but outside the allowed range, an `IndexError` should be raised.

Let's have a go at it—let's see if we can create an infinite sequence:

```
def checkIndex(key):  
    """  
    Is the given key an acceptable index?  
  
    To be acceptable, the key should be a non-negative integer. If it  
    is not an integer, a TypeError is raised; if it is negative,  
    an IndexError is raised (since the sequence is of infinite length).  
    """  
    if not isinstance(key, (int, long)): raise TypeError  
    if key<0: raise IndexError  
  
class ArithmeticSequence:  
    def __init__(self, start=0, step=1):  
        """  
        Initialize the arithmetic sequence.  
  
        start - the first value in the sequence  
        step - the difference between two adjacent values  
        changed - a dictionary of values that have been modified by  
                  the user  
        """  
        self.start = start # Store the start value  
        self.step = step # Store the step value  
        self.changed = {} # No items have been modified  
  
    def __getitem__(self, key):  
        """  
        Get an item from the arithmetic sequence.  
        """  
        checkIndex(key)  
  
        try: return self.changed[key] # Modified?  
        except KeyError: # otherwise...  
            return self.start + key*self.step # ...calculate the value
```

```
def __setitem__(self, key, value):
    """
    Change an item in the arithmetic sequence.
    """
    checkIndex(key)

    self.changed[key] = value           # Store the changed value
```

This implements an *arithmetic sequence*—a sequence of numbers in which each is greater than the previous one by a constant amount. The first value is given by the constructor parameter `start` (defaulting to zero), while the step between the values is given by `step` (defaulting to one). You allow the user to change some of the elements by keeping the exceptions to the general rule in a dictionary called `changed`. If the element hasn't been changed, it is calculated as `self.start + key * self.step`.

Here is an example of how you can use this class:

```
>>> s = ArithmeticSequence(1, 2)
>>> s[4]
9
>>> s[4] = 2
>>> s[4]
2
>>> s[5]
11
```

Note that I want it to be illegal to delete items, which is why I haven't implemented `__del__`:

```
>>> del s[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: ArithmeticSequence instance has no attribute '__delitem__'
```

Also, the class has no `__len__` method because it is of infinite length.

If an illegal type of index is used, a `TypeError` is raised, and if the index is the correct type but out of range (that is, negative in this case), an `IndexError` is raised:

```
>>> s["four"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "arithseq.py", line 31, in __getitem__
    checkIndex(key)
  File "arithseq.py", line 10, in checkIndex
    if not isinstance(key, int): raise TypeError
```

```
TypeError
>>> s[-42]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "arithseq.py", line 31, in __getitem__
    checkIndex(key)
  File "arithseq.py", line 11, in checkIndex
    if key<0: raise IndexError
IndexError
```

The index checking is taken care of by a utility function I've written for the purpose, `checkIndex`.

One thing that might surprise you about the `checkIndex` function is the use of `isinstance` (which you should rarely use because type or class checking goes against the grain of Python's polymorphism). I've used it because the language reference explicitly states that the index should be an integer (this includes long integers). And complying with standards is one of the (very few) valid reasons for using type checking.

---

**Note** You can simulate slicing, too, if you like. When slicing an instance that supports `__getitem__`, a slice object is supplied as the key. Slice objects are described in the Python Library Reference (<http://python.org/doc/lib>) in Section 2.1, “Built-in Functions,” under the `slice` function. Python 2.5 also has the more specialized method called `__index__`, which allows you to use noninteger limits in your slices. This is mainly useful only if you wish to go beyond the basic sequence protocol, though.

---

## Subclassing `list`, `dict`, and `str`

While the four methods of the basic sequence/mapping protocol will get you far, the official language reference also recommends that several other magic and ordinary methods be implemented (see the section “Emulating container types” in the Python Reference Manual, <http://www.python.org/doc/ref/sequence-types.html>), including the `__iter__` method, which I describe in the section “Iterators,” later in this chapter. Implementing all these methods (to make your objects fully polymorphically equivalent to lists or dictionaries) is a lot of work and hard to get right. If you want custom behavior in only *one* of the operations, it makes no sense that you should need to reimplement all of the others. It's just programmer laziness (also called common sense).

So what should you do? The magic word is *inheritance*. Why reimplement all of these things when you can inherit them? The standard library comes with three ready-to-use implementations of the sequence and mapping protocols (`UserList`, `UserString`, and `UserDict`), and in current versions of Python, you can subclass the built-in types themselves. (Note that this is mainly useful if your class's behavior is close to the default. If you need to reimplement most of the methods, it might be just as easy to write a new class.)

So, if you want to implement a sequence type that behaves similarly to the built-in lists, you can simply subclass `list`.

---

**Note** When you subclass a built-in type such as `list`, you are indirectly subclassing `object`. Therefore your class is automatically new-style, which means that features such as the `super` function are available.

---

Let's just do a quick example—a list with an access counter:

```
class CounterList(list):
    def __init__(self, *args):
        super(CounterList, self).__init__(*args)
        self.counter = 0
    def __getitem__(self, index):
        self.counter += 1
        return super(CounterList, self).__getitem__(index)
```

The `CounterList` class relies heavily on the behavior of its subclass superclass (`list`). Any methods not overridden by `CounterList` (such as `append`, `extend`, `index`, and so on) may be used directly. In the two methods that *are* overridden, `super` is used to call the superclass version of the method, adding only the necessary behavior of initializing the `counter` attribute (in `__init__`) and updating the `counter` attribute (in `__getitem__`).

---

**Note** Overriding `__getitem__` is not a bulletproof way of trapping user access because there are other ways of accessing the list contents, such as through the `pop` method.

---

Here is an example of how `CounterList` may be used:

```
>>> cl = CounterList(range(10))
>>> cl
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> cl.reverse()
>>> cl
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> del cl[3:6]
>>> cl
[9, 8, 7, 3, 2, 1, 0]
>>> cl.counter
0
>>> cl[4] + cl[2]
9
>>> cl.counter
2
```

As you can see, CounterList works just like list in most respects. However, it has a counter attribute (initially zero), which is incremented each time you access a list element. After performing the addition c1[4] + c1[2], the counter has been incremented twice, to the value 2.

## More Magic

Special (magic) names exist for many purposes—what I've shown you so far is just a small taste of what is possible. Most of the magic methods available are meant for fairly advanced use, so I won't go into detail here. However, if you are interested, it is possible to emulate numbers, make objects that can be called as if they were functions, influence how objects are compared, and much more. For more information about which magic methods are available, see section "Special method names" in the Python Reference Manual (<http://www.python.org/doc/ref/specnames.html>).

## Properties

In Chapter 7, I mentioned *accessor methods*. Accessors are simply methods with names such as getHeight and setHeight, and are used to retrieve or rebind some attribute (which may be private to the class—see the section "Privacy Revisited" in Chapter 7). Encapsulating state variables (attributes) like this can be important if certain actions must be taken when accessing the given attribute. For example, consider the following Rectangle class:

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def setSize(self, size):
        self.width, self.height = size
    def getSize(self):
        return self.width, self.height
```

Here is an example of how you can use the class:

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.getSize()
(10, 5)
>>> r.setSize((150, 100))
>>> r.width
150
```

The getSize and setSize methods are accessors for a fictitious attribute called size—which is simply the tuple consisting of width and height. (Feel free to replace this with something more exciting, such as the area of the rectangle or the length of its diagonal.) This code isn't directly wrong, but it is flawed. The programmer using this class shouldn't need to worry about how it is implemented (encapsulation). If you someday wanted to change the implementation so that size was a real attribute and width and height were calculated on the fly, you

would need to wrap *them* in accessors, and any programs using the class would also have to be rewritten. The client code (the code using your code) should be able to treat all your attributes in the same manner.

So what is the solution? Should you wrap all your attributes in accessors? That is a possibility, of course. However, it would be impractical (and kind of silly) if you had a lot of simple attributes, because you would need to write many accessors that did nothing but retrieve or set these attributes, with no useful action taken. This smells of *copy-paste* programming, or *cookie-cutter code*, which is clearly a bad thing (although quite common for this specific problem in certain languages). Luckily, Python can hide your accessors for you, making all of your attributes look alike. Those attributes that are defined through their accessors are often called *properties*.

Python actually has two mechanisms for creating properties in Python. I'll focus on the most recent one, the `property` function, which works only on new-style classes. Then I'll give you a short description of how to implement properties with magic methods.

## The `property` Function

Using the `property` function is delightfully simple. If you have already written a class such as `Rectangle` from the previous section, you need to add only a single line of code (in addition to subclassing `object`, or using `__metaclass__ = type`):

```
__metaclass__ = type

class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def setSize(self, size):
        self.width, self.height = size
    def getSize(self):
        return self.width, self.height
    size = property(getSize, setSize)
```

In this new version of `Rectangle`, a property is created with the `property` function with the accessor functions as arguments (the *getter* first, then the *setter*), and the name `size` is then bound to this property. After this, you no longer need to worry about how things are implemented, but can treat `width`, `height`, and `size` the same way:

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.size
(10, 5)
>>> r.size = 150, 100
>>> r.width
150
```

As you can see, the `size` attribute is still subject to the calculations in `getSize` and `setSize`, but it looks just like a normal attribute.

---

**Note** If your properties are behaving oddly, make sure you’re using a new-style class (by subclassing `object` either directly or indirectly—or by setting the metaclass directly). If you aren’t, the *getter* part of the property will still work, but the *setter* may not (depending on your Python version). This can be a bit confusing.

---

In fact, the `property` function may be called with zero, one, three, or four arguments as well. If called without any arguments, the resulting property is neither readable nor writable. If called with only one argument (a getter method), the property is readable only. The third (optional) argument is a method used to *delete* the attribute (it takes no arguments). The fourth (optional) argument is a docstring. The parameters are called `fget`, `fset`, `fdel`, and `doc`—you can use them as keyword arguments if you want a property that, say, is only writable and has a docstring.

Although this section has been short (a testament to the simplicity of the `property` function), it is very important. The moral is this: with new-style classes, you should use `property` rather than accessors.

### BUT HOW DOES IT WORK?

In case you’re curious about how `property` does its magic, I’ll give you an explanation here. If you don’t care, just skip ahead.

The fact is that `property` isn’t really a function—it’s a class whose instances have some magic methods that do all the work. The methods in question are `__get__`, `__set__`, and `__delete__`. Together, these three methods define the so-called *descriptor protocol*. An object implementing any of these methods is a descriptor. The special thing about descriptors is how they are accessed. For example, when reading an attribute (specifically, when accessing it in an instance, but when the attribute is defined in the class), if the attribute is bound to an object that implements `__get__`, the object won’t simply be returned; instead, the `__get__` method will be called and the resulting value will be returned. This is, in fact, the mechanism underlying properties, bound methods, static and class methods (see the following section for more information), and `super`. A brief description of the descriptor protocol may be found in the Python Reference Manual (<http://python.org/doc/ref/descriptors.html>). A more thorough source of information is Raymond Hettinger’s How-To Guide for Descriptors (<http://users.rcn.com/python/download/Descriptor.htm>).

## Static Methods and Class Methods

Before discussing the old way of implementing properties, let’s take a slight detour, and look at another couple of features that are implemented in a similar manner to the new-style properties. Static methods and class methods are created by wrapping methods in objects of the `staticmethod` and `classmethod` types, respectively. Static methods are defined without `self` arguments, and they can be called directly on the class itself. Class methods are defined with a

self-like parameter normally called `cls`. You can call class methods directly on the class object too, but the `cls` parameter then automatically is bound to the class. Here is a simple example:

```
__metaclass__ = type

class MyClass:

    def smeth():
        print 'This is a static method'
    smeth = staticmethod(smeth)

    def cmeth(cls):
        print 'This is a class method of', cls
    cmeth = classmethod(cmeth)
```

The technique of wrapping and replacing the methods manually like this is a bit tedious. In Python 2.4, a new syntax was introduced for wrapping methods like this, called *decorators*. (They actually work with any callable objects as wrappers, and can be used on both methods and functions.) You specify one or more decorators (which are applied in reverse order) by listing them above the method (or function), using the @ operator:

```
__metaclass__ = type

class MyClass:

    @staticmethod
    def smeth():
        print 'This is a static method'

    @classmethod
    def cmeth(cls):
        print 'This is a class method of', cls
```

Once you've defined these methods, they can be used like this (that is, without instantiating the class):

```
>>> MyClass.smeth()
This is a static method
>>> MyClass.cmeth()
This is a class method of <class '__main__.MyClass'>
```

Static methods and class methods haven't historically been important in Python, mainly because you could always use functions or bound methods instead, in some way, but also because the support hasn't really been there in earlier versions. So even though you may not see them used

much in current code, they do have their uses (such as factory functions, if you've heard of those), and you may well think of some new ones.

## **`__getattr__`, `__setattr__`, and Friends**

It's possible to intercept every attribute access on an object. Among other things, you could use this to implement properties with old-style classes (where `property` won't necessarily work as it should). To have code executed when an attribute is accessed, you must use a couple of magic methods. The following four provide all the functionality you need (in old-style classes, you only use the last three):

`__getattribute__(self, name)`: Automatically called when the attribute `name` is accessed. (This works correctly on new-style classes only.)

`__getattr__(self, name)`: Automatically called when the attribute `name` is accessed and the object has no such attribute.

`__setattr__(self, name, value)`: Automatically called when an attempt is made to bind the attribute `name` to `value`.

`__delattr__(self, name)`: Automatically called when an attempt is made to delete the attribute `name`.

Although a bit trickier to use (and in some ways less efficient) than `property`, these magic methods are quite powerful, because you can write code in one of these methods that deals with several properties. (If you have a choice, though, stick with `property`.)

Here is the `Rectangle` example again, this time with magic methods:

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def __setattr__(self, name, value):
        if name == 'size':
            self.width, self.height = value
        else:
            self.__dict__[name] = value
    def __getattribute__(self, name):
        if name == 'size':
            return self.width, self.height
        else:
            raise AttributeError
```

As you can see, this version of the class needs to take care of additional administrative details. When considering this code example, it's important to note the following:

- The `__setattr__` method is called even if the attribute in question is not `size`. Therefore, the method must take both cases into consideration: if the attribute is `size`, the same operation is performed as before; otherwise, the magic attribute `__dict__` is used. It contains a dictionary with all the instance attributes. It is used instead of ordinary attribute assignment to avoid having `__setattr__` called again (which would cause the program to loop endlessly).
- The `__getattr__` method is called only if a normal attribute is not found, which means that if the given name is not `size`, the attribute does not exist, and the method raises an `AttributeError`. This is important if you want the class to work correctly with built-in functions such as `hasattr` and `getattr`. If the name is `size`, the expression found in the previous implementation is used.

---

**Note** Just as there is an “endless loop” trap associated with `__setattr__`, there is a trap associated with `__getattribute__`. Because it intercepts *all* attribute accesses (in new-style classes), it will intercept accesses to `__dict__` as well! The only safe way to access attributes on `self` inside `__getattribute__` is to use the `__getattribute__` method of the superclass (using `super`).

---

## Iterators

I've mentioned iterators (and iterables) briefly in preceding chapters. In this section, I go into some more detail. I cover only one magic method, `__iter__`, which is the basis of the iterator protocol.

### The Iterator Protocol

To *iterate* means to repeat something several times—what you do with loops. Until now I have iterated over only sequences and dictionaries in `for` loops, but the truth is that you can iterate over other objects, too: objects that implement the `__iter__` method.

The `__iter__` method returns an iterator, which is any object with a method called `next`, which is callable without any arguments. When you call the `next` method, the iterator should return its “next value.” If the method is called, and the iterator has no more values to return, it should raise a `StopIteration` exception.

---

**Note** The iterator protocol is changed a bit in Python 3.0. In the new protocol, iterator objects should have a method called `__next__` rather than `next`, and a new built-in function called `next` may be used to access this method. In other words, `next(it)` is the equivalent of the pre-3.0 `it.next()`.

---

What's the point? Why not just use a list? Because it may often be overkill. If you have a function that can compute values one by one, you may need them only one by one—not all at once, in a list, for example. If the number of values is large, the list may take up too much memory. But there are other reasons: using iterators is more general, simpler, and more elegant. Let's take a look at an example you couldn't do with a list, simply because the list would need to be of infinite length!

Our “list” is the sequence of Fibonacci numbers. An iterator for these could be the following:

```
class Fibs:  
    def __init__(self):  
        self.a = 0  
        self.b = 1  
    def next(self):  
        self.a, self.b = self.b, self.a+self.b  
        return self.a  
    def __iter__(self):  
        return self
```

Note that the iterator implements the `__iter__` method, which will, in fact, return the iterator itself. In many cases, you would put the `__iter__` method in *another* object, which you would use in the `for` loop. That would then return your iterator. It is recommended that iterators implement an `__iter__` method of their own in addition (returning `self`, just as I did here), so they themselves can be used directly in `for` loops.

---

**Note** In formal terms, an object that implements the `__iter__` method is *iterable*, and the object implementing `next` is the *iterator*.

---

First, make a `Fibs` object:

```
>>> fibs = Fibs()
```

You can then use it in a `for` loop—for example, to find the smallest Fibonacci number that is greater than 1,000:

```
>>> for f in fibs:  
    if f > 1000:  
        print f  
        break  
...  
1597
```

Here, the loop stops because I issue a `break` inside it; if I didn't, the `for` loop would never end.

**Tip** The built-in function `iter` can be used to get an iterator from an iterable object:

```
>>> it = iter([1, 2, 3])
>>> it.next()
1
>>> it.next()
2
```

It can also be used to create an iterable from a function or other callable object (see the Python Library Reference, <http://docs.python.org/lib/>, for details).

---

## Making Sequences from Iterators

In addition to *iterating* over the iterators and iterables (which is what you normally do), you can convert them to sequences. In most contexts in which you can use a sequence (except in operations such as indexing or slicing), you can use an iterator (or an iterable object) instead. One useful example of this is explicitly converting an iterator to a list using the `list` constructor:

```
>>> class TestIterator:
...     value = 0
...     def next(self):
...         self.value += 1
...         if self.value > 10: raise StopIteration
...         return self.value
...     def __iter__(self):
...         return self
...
>>> ti = TestIterator()
>>> list(ti)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Generators

Generators (also called *simple generators* for historical reasons) are relatively new to Python, and are (along with iterators) perhaps one of the most powerful features to come along for years. However, the generator concept is rather advanced, and it may take a while before it “clicks” and you see how it works or how it would be useful for you. Rest assured that while generators can help you write really elegant code, you can certainly write any program you wish without a trace of generators.

A generator is a kind of iterator that is defined with normal function syntax. Exactly how generators work is best shown through example. Let's first have a look at how you make them and use them, and then take a peek under the hood.

## Making a Generator

Making a generator is simple; it's just like making a function. I'm sure you are starting to tire of the good old Fibonacci sequence by now, so let me do something else. I'll make a function that flattens nested lists. The argument is a list that may look something like this:

```
nested = [[1, 2], [3, 4], [5]]
```

In other words, it's a list of lists. My function should then give me the numbers in order. Here's a solution:

```
def flatten(nested):
    for sublist in nested:
        for element in sublist:
            yield element
```

Most of this function is pretty simple. First, it iterates over all the sublists of the supplied nested list; then it iterates over the elements of each sublist in order. If the last line had been `print element`, for example, the function would have been easy to understand, right?

So what's new here is the `yield` statement. Any function that contains a `yield` statement is called a *generator*. And it's not just a matter of naming; it will behave quite differently from ordinary functions. The difference is that instead of returning *one* value, as you do with `return`, you can yield *several* values, one at a time. Each time a value is yielded (with `yield`), the function *freezes*; that is, it stops its execution at exactly that point and waits to be reawakened. When it is, it resumes its execution at the point where it stopped.

I can make use of all the values by iterating over the generator:

```
>>> nested = [[1, 2], [3, 4], [5]]
>>> for num in flatten(nested):
...     print num
...
1
2
3
4
5
```

or

```
>>> list(flatten(nested))
[1, 2, 3, 4, 5]
```

## LOOPY GENERATORS

In Python 2.4, a relative of list comprehension (see Chapter 5) was introduced: *generator comprehension* (or *generator expressions*). It works in the same way as list comprehension, except that a list isn't constructed (and the “body” isn't looped over immediately). Instead, a generator is returned, allowing you to perform the computation step by step:

```
>>> g = ((i+2)**2 for i in range(2,27))
>>> g.next()
16
```

As you can see, this differs from list comprehension in the use of plain parentheses. In a simple case such as this, I might as well have used a list comprehension. However, if you wish to “wrap” an iterable object (possibly yielding a huge number of values), a list comprehension would void the advantages of iteration by immediately instantiating a list.

A neat bonus is that when using generator comprehension directly inside a pair of existing parentheses, such as in a function call, you don't need to add another pair. In other words, you can write pretty code like this:

```
sum(i**2 for i in range(10))
```

## A Recursive Generator

The generator I designed in the previous section could deal only with lists nested two levels deep, and to do that it used two `for` loops. What if you have a set of lists nested arbitrarily deeply? Perhaps you use them to represent some tree structure, for example. (You can also do that with specific tree classes, but the strategy is the same.) You need a `for` loop for each level of nesting, but because you don't know how many levels there are, you must change your solution to be more flexible. It's time to turn to the magic of recursion:

```
def flatten(nested):
    try:
        for sublist in nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

When `flatten` is called, you have two possibilities (as is always the case when dealing with recursion): the *base case* and the *recursive case*. In the base case, the function is told to flatten a single element (for example, a number), in which case the `for` loop raises a `TypeError` (because you're trying to iterate over a number), and the generator simply yields the element.

If you are told to flatten a list (or any iterable), however, you need to do some work. You go through all the sublists (some of which may not really be lists) and call `flatten` on them. Then you yield all the elements of the flattened sublists by using another `for` loop. It may seem slightly magical, but it works:

```
>>> list(flatten([[1],2],3,4,[5,[6,7]],8))
[1, 2, 3, 4, 5, 6, 7, 8]
```

There is one problem with this, however. If `nested` is a string-like object (string, Unicode, `UserString`, and so on), it is a sequence and will not raise `TypeError`, yet you do *not* want to iterate over it.

---

**Note** There are two main reasons why you shouldn't iterate over string-like objects in the `flatten` function. First, you want to treat string-like objects as atomic values, not as sequences that should be flattened. Second, iterating over them would actually lead to infinite recursion because the first element of a string is another string of length one, and the first element of *that* string is the string itself!

---

To deal with this, you must add a test at the beginning of the generator. Trying to concatenate the object with a string and seeing if a `TypeError` results is the simplest and fastest way to check whether an object is string-like.<sup>2</sup> Here is the generator with the added test:

```
def flatten(nested):
    try:
        # Don't iterate over string-like objects:
        try: nested + ''
        except TypeError: pass
        else: raise TypeError
        for sublist in nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

As you can see, if the expression `nested + ''` raises a `TypeError`, it is ignored; however, if the expression does *not* raise a `TypeError`, the `else` clause of the inner `try` statement raises a `TypeError` of its own. This causes the string-like object to be yielded as is (in the outer `except` clause). Got it?

Here is an example to demonstrate that this version works with strings as well:

```
>>> list(flatten(['foo', ['bar', ['baz']])))
['foo', 'bar', 'baz']
```

Note that there is no type checking going on here. I don't test whether `nested` is a string (which I could do by using `isinstance`), only whether it *behaves* like one (that is, it can be concatenated with a string).

## Generators in General

If you followed the examples so far, you know how to use generators, more or less. You've seen that a generator is a function that contains the keyword `yield`. When it is called, the code in the function body is not executed. Instead, an iterator is returned. Each time a value is requested,

---

2. Thanks to Alex Martelli for pointing out this idiom and the importance of using it here.

the code in the generator is executed until a `yield` or a `return` is encountered. A `yield` means that a value should be yielded. A `return` means that the generator should stop executing (without yielding anything more; `return` can be called without arguments only when used inside a generator).

In other words, generators consist of two separate components: the *generator-function* and the *generator-iterator*. The generator-function is what is defined by the `def` statement containing a `yield`. The generator-iterator is what this function returns. In less precise terms, these two entities are often treated as one and collectively called *a generator*.

```
>>> def simple_generator():
...     yield 1
...
>>> simple_generator
<function simple_generator at 153b44>
>>> simple_generator()
<generator object at 1510b0>
```

The iterator returned by the generator-function can be used just like any other iterator.

## Generator Methods

A relatively new feature of generators (added in Python 2.5) is the ability to supply generators with values after they have started running. This takes the form of a communications channel between the generator and the “outside world,” with the following two end points:

- The outside world has access to a method on the generator called `send`, which works just like `next`, except that it takes a single argument (the “message” to send—an arbitrary object).
- Inside the suspended generator, `yield` may now be used as an *expression*, rather than a *statement*. In other words, when the generator is resumed, `yield` returns a value—the value sent from the outside through `send`. If `next` was used, `yield` returns `None`.

Note that using `send` (rather than `next`) makes sense only after the generator has been suspended (that is, after it has hit the first `yield`). If you need to give some information to the generator before that, you can simply use the parameters of the generator-function.

---

**Tip** If you *really* want to use `send` on a newly started generator, you can use it with `None` as its parameter.

---

Here’s a rather silly example that illustrates the mechanism:

```
def repeater(value):
    while True:
        new = (yield value)
        if new is not None: value = new
```

Here's an example of its use:

```
r = repeater(42)
r.next()
42
r.send("Hello, world!")
"Hello, world!"
```

Note the use of parentheses around the `yield` expression. While not strictly necessary in some cases, it is probably better to be safe than sorry, and simply always enclose `yield` expressions in parentheses if you are using the return value in some way.

Generators also have two other methods (in Python 2.5 and later):

- The `throw` method (called with an exception type, an optional value and traceback object) is used to raise an exception inside the generator (at the `yield` expression).
- The `close` method (called with no arguments) is used to stop the generator.

The `close` method (which is also called by the Python garbage collector, when needed) is also based on exceptions. It raises the `GeneratorExit` exception at the `yield` point, so if you want to have some cleanup code in your generator, you can wrap your `yield` in a `try/finally` statement. If you wish, you can also catch the `GeneratorExit` exception, but then you must reraise it (possibly after cleaning up a bit), raise another exception, or simply return. Trying to yield a value from a generator after `close` has been called on it will result in a `RuntimeError`.

---

**Tip** For more information about generator methods, and how these actually turn generators into simple *coroutines*, see PEP 342 (<http://www.python.org/dev/peps/pep-0342/>).

---

## Simulating Generators

If you need to use an older version of Python, generators aren't available. What follows is a simple recipe for simulating them with normal functions.

Starting with the code for the generator, begin by inserting the following line at the beginning of the function body:

```
result = []
```

If the code already uses the name `result`, you should come up with another. (Using a more descriptive name may be a good idea anyway.) Then replace all lines of this form:

```
yield some_expression
```

with this:

```
result.append(some_expression)
```

Finally, at the end of the function, add this line:

```
return result
```

Although this may not work with all generators, it works with most. (For example, it fails with infinite generators, which of course can't stuff their values into a list.)

Here is the `flatten` generator rewritten as a plain function:

```
def flatten(nested):
    result = []
    try:
        # Don't iterate over string-like objects:
        try: nested + ''
        except TypeError: pass
        else: raise TypeError
        for sublist in nested:
            for element in flatten(sublist):
                result.append(element)
    except TypeError:
        result.append(nested)
    return result
```

## The Eight Queens

Now that you've learned about all this magic, it's time to put it to work. In this section, you see how to use generators to solve a classic programming problem.

### Generators and Backtracking

Generators are ideal for complex recursive algorithms that gradually build a result. Without generators, these algorithms usually require you to pass a half-built solution around as an extra parameter so that the recursive calls can build on it. With generators, all the recursive calls need to do is `yield` their part. That is what I did with the preceding recursive version of `flatten`, and you can use the exact same strategy to traverse graphs and tree structures.

In some applications, however, you don't get the answer right away; you need to try several alternatives, and you need to do that on *every* level in your recursion. To draw a parallel from real life, imagine that you have an important meeting to attend. You're not sure where it is, but you have two doors in front of you, and the meeting room has to be behind one of them. You choose the left and step through. There, you face another two doors. You choose the left, but it turns out to be wrong. So you *backtrack*, and choose the right door, which also turns out to be wrong (excuse the pun). So, you backtrack again, to the point where you started, ready to try the right door there.

## GRAPHS AND TREES

If you have never heard of graphs and trees before, you should learn about them as soon as possible, because they are very important concepts in programming and computer science. To find out more, you should probably get a book about computer science, discrete mathematics, data structures, or algorithms. For some concise definitions, you can check out the following web pages:

- <http://mathworld.wolfram.com/Graph.html>
- <http://mathworld.wolfram.com/Tree.html>
- <http://www.nist.gov/dads/HTML/tree.html>
- <http://www.nist.gov/dads/HTML/graph.html>

A quick web search or some browsing in Wikipedia (<http://wikipedia.org>) will turn up a lot of material.

This strategy of backtracking is useful for solving problems that require you to try every combination until you find a solution. Such problems are solved like this:

```
# Pseudocode
for each possibility at level 1:
    for each possibility at level 2:
        ...
            for each possibility at level n:
                is it viable?
```

To implement this directly with `for` loops, you need to know how many levels you'll encounter. If that is not possible, you use recursion.

## The Problem

This is a much loved computer science puzzle: you have a chessboard and eight queen pieces to place on it. The only requirement is that none of the queens threatens any of the others; that is, you must place them so that no two queens can capture each other. How do you do this? Where should the queens be placed?

This is a typical backtracking problem: you try one position for the first queen (in the first row), advance to the second, and so on. If you find that you are unable to place a queen, you backtrack to the previous one and try another position. Finally, you either exhaust all possibilities or find a solution.

In the problem as stated, you are provided with information that there will be only eight queens, but let's assume that there can be any number of queens. (This is more similar to real-world backtracking problems.) How do you solve that? If you want to try to solve it yourself, you should stop reading now, because I'm about to give you the solution.

---

**Note** You can find much more efficient solutions for this problem. If you want more details, a web search should turn up a wealth of information. A brief history of various solutions may be found at <http://www.cit.gu.edu.au/~sosic/nqueens.html>.

---

## State Representation

To represent a possible solution (or part of it), you can simply use a tuple (or a list, for that matter). Each element of the tuple indicates the position (that is, column) of the queen of the corresponding row. So if `state[0] == 3`, you know that the queen in row one is positioned in column four (we are counting from zero, remember?). When working at one level of recursion (one specific row), you know only which positions the queens above have, so you may have a state tuple whose length is less than eight (or whatever the number of queens is).

---

**Note** I could well have used a list instead of a tuple to represent the state. It's mostly a matter of taste in this case. In general, if the sequence is small and static, tuples may be a good choice.

---

## Finding Conflicts

Let's start by doing some simple abstraction. To find a configuration in which there are no conflicts (where no queen may capture another), you first must define what a conflict is. And why not define it as a function while you're at it?

The `conflict` function is given the positions of the queens *so far* (in the form of a state tuple) and determines if a position for the *next* queen generates any new conflicts:

```
def conflict(state, nextX):
    nextY = len(state)
    for i in range(nextY):
        if abs(state[i]-nextX) in (0, nextY-i):
            return True
    return False
```

The `nextX` parameter is the suggested horizontal position (x coordinate, or column) of the next queen, and `nextY` is the vertical position (y coordinate, or row) of the next queen. This function does a simple check for each of the previous queens. If the next queen has the same x coordinate, or is on the same diagonal as (`nextX`, `nextY`), a conflict has occurred, and `True` is returned. If no such conflicts arise, `False` is returned. The tricky part is the following expression:

```
abs(state[i]-nextX) in (0, nextY-i)
```

It is true if the horizontal distance between the next queen and the previous one under consideration is either zero (same column) or equal to the vertical distance (on a diagonal). Otherwise, it is false.

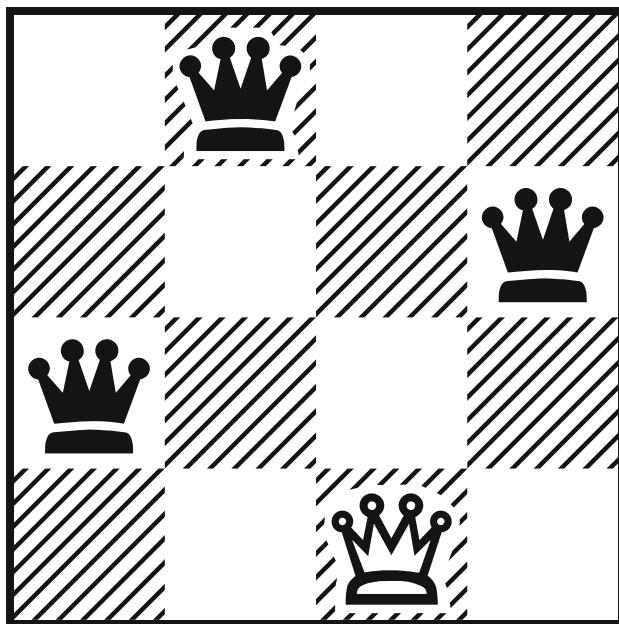
## The Base Case

The Eight Queens problem can be a bit tricky to implement, but with generators it isn't so bad. If you aren't used to recursion, I wouldn't expect you to come up with this solution by yourself, though. Note also that this solution isn't particularly efficient, so with a very large number of queens, it might be a bit slow.

Let's begin with the base case: the last queen. What would you want her to do? Let's say you want to find all possible solutions. In that case, you would expect her to produce (generate) all the positions she could occupy (possibly none) given the positions of the others. You can sketch this out directly:

```
def queens(num, state):
    if len(state) == num-1:
        for pos in range(num):
            if not conflict(state, pos):
                yield pos
```

In human-speak, this means, "If all queens but one have been placed, go through all possible positions for the last one, and return the positions that don't give rise to any conflicts." The `num` parameter is the number of queens in total, and the `state` parameter is the tuple of positions for the previous queens. For example, let's say you have four queens, and that the first three have been given the positions 1, 3, and 0, respectively, as shown in Figure 9-1. (Pay no attention to the white queen at this point.)



**Figure 9-1.** Placing four queens on a  $4 \times 4$  board

As you can see in the figure, each queen gets a (horizontal) row, and the queens' positions are numbered across the top (beginning with zero, as is normal in Python):

```
>>> list(queens(4, (1,3,0)))
[2]
```

It works like a charm. Using `list` simply forces the generator to yield all of its values. In this case, only one position qualifies. The white queen has been put in this position in Figure 9-1. (Note that color has no special significance and is not part of the program.)

## The Recursive Case

Now let's turn to the recursive part of the solution. When you have your base case covered, the recursive case may correctly assume (by induction) that all results from lower levels (the queens with higher numbers) are correct. So what you need to do is add an `else` clause to the `if` statement in the previous implementation of the `queens` function.

What results do you expect from the recursive call? You want the positions of all the lower queens, right? Let's say they are returned as a tuple. In that case, you probably need to change your base case to return a tuple as well (of length one)—but I get to that later.

So, you're supplied with one tuple of positions from “above,” and for each legal position of the current queen, you are supplied with a tuple of positions from “below.” All you need to do to keep things flowing is to yield the result from below with your own position added to the front:

```
...
else:
    for pos in range(num):
        if not conflict(state, pos):
            for result in queens(num, state + (pos,)):
                yield (pos,) + result
```

The `for pos` and `if not conflict` parts of this are identical to what you had before, so you can rewrite this a bit to simplify the code. Let's add some default arguments as well:

```
def queens(num=8, state=()):
    for pos in range(num):
        if not conflict(state, pos):
            if len(state) == num-1:
                yield (pos,)
            else:
                for result in queens(num, state + (pos,)):
                    yield (pos,) + result
```

If you find the code hard to understand, you might find it helpful to formulate what it does in your own words. (And you do remember that the comma in `(pos,)` is necessary to make it a tuple, and not simply a parenthesized value, right?)

The queens generator gives you all the solutions (that is, all the legal ways of placing the queens):

```
>>> list(queens(3))
[]
>>> list(queens(4))
[(1, 3, 0, 2), (2, 0, 3, 1)]
>>> for solution in queens(8):
...     print solution
...
(0, 4, 7, 5, 2, 6, 1, 3)
(0, 5, 7, 2, 6, 3, 1, 4)
...
(7, 2, 0, 5, 1, 4, 6, 3)
(7, 3, 0, 2, 5, 1, 6, 4)
>>>
```

If you run queens with eight queens, you see a lot of solutions flashing by. Let's find out how many:

```
>>> len(list(queens(8)))
92
```

## Wrapping It Up

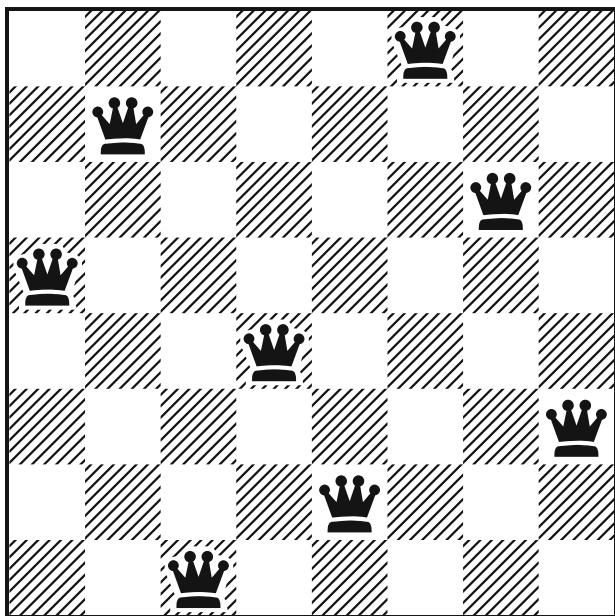
Before leaving the queens, let's make the output a bit more understandable. Clear output is always a good thing because it makes it easier to spot bugs, among other things.

```
def prettyprint(solution):
    def line(pos, length=len(solution)):
        return '.' * (pos) + 'X' + '.' * (length-pos-1)
    for pos in solution:
        print line(pos)
```

Note that I've made a little helper function inside prettyprint. I put it there because I assumed I wouldn't need it anywhere outside. In the following, I print out a random solution to satisfy myself that it is correct:

```
>>> import random
>>> prettyprint(random.choice(list(queens(8))))
. . . . X . .
. X . . . . .
. . . . . X .
X . . . . .
. . . X . . .
. . . . . X
. . . X . . .
. . X . . . .
```

This “drawing” corresponds to the diagram in Figure 9-2. Fun to play with Python, isn’t it?



**Figure 9-2.** One of many possible solutions to the Eight Queens problem

## A Quick Summary

You’ve seen a lot of magic here. Let’s take stock:

**New-style vs. old-style classes:** The way classes work in Python is changing. Recent (pre-3.0) versions of Python have two sorts of classes, with the old-style ones quickly going out of fashion. The new-style classes were introduced in version 2.2, and they provide several extra features (for example, they work with `super` and `property`, while old-style classes do not). To create a new-style class, you must subclass `object`, either directly or indirectly, or set the `__metaclass__` property.

**Magic methods:** Several special methods (with names beginning and ending with double underscores) exist in Python. These methods differ quite a bit in function, but most of them are called automatically by Python under certain circumstances. (For example, `__init__` is called after object creation.)

**Constructors:** These are common to many object-oriented languages, and you’ll probably implement one for almost every class you write. Constructors are named `__init__` and are automatically called immediately after an object is created.

**Overriding:** A class can override methods (or any other attributes) defined in its superclasses simply by implementing the methods. If the new method needs to call the overridden version, it can either call the unbound version from the superclass directly (old-style classes) or use the `super` function (new-style classes).

**Sequences and mappings:** Creating a sequence or mapping of your own requires implementing all the methods of the sequence and mapping protocols, including such magic

methods as `__getitem__` and `__setitem__`. By subclassing `list` (or `UserList`) and `dict` (or `UserDict`), you can save a lot of work.

**Iterators:** An *iterator* is simply an object that has a `next` method. Iterators can be used to iterate over a set of values. When there are no more values, the `next` method should raise a `StopIteration` exception. *Iterable* objects have an `__iter__` method, which returns an iterator, and can be used in `for` loops, just like sequences. Often, an iterator is also iterable; that is, it has an `__iter__` method that returns the iterator itself.

**Generators:** A *generator-function* (or method) is a function (or method) that contains the keyword `yield`. When called, the generator-function returns a *generator*, which is a special type of iterator. You can interact with an active generator from the outside by using the methods `send`, `throw`, and `close`.

**Eight Queens:** The Eight Queens problem is well known in computer science and lends itself easily to implementation with generators. The goal is to position eight queens on a chessboard so that none of the queens is in a position from which she can attack any of the others.

## New Functions in This Chapter

Function	Description
<code>iter(obj)</code>	Extracts an iterator from an iterable object
<code>property(fget, fset, fdel, doc)</code>	Returns a property; all arguments are optional
<code>super(class, obj)</code>	Returns a bound instance of <code>class</code> 's superclass

Note that `iter` and `super` may be called with other parameters than those described here. For more information, see the standard Python documentation (<http://python.org/doc>).

## What Now?

Now you know most of the Python language. So why are there still so many chapters left? Well, there is still a *lot* to learn, much of it about how Python can connect to the external world in various ways. And then we have testing, extending, packaging, and the projects, so we're not done yet—not by far.

# Chapter 7

## Python GUI

## CHAPTER 12



# Graphical User Interfaces

In this chapter, you learn how to make graphical user interfaces (GUIs) for your Python programs—you know, windows with buttons and text fields and stuff like that. Pretty cool, huh?

Plenty of so-called “GUI toolkits” are available for Python, but none of them is recognized as *the* standard GUI toolkit. This has its advantages (greater freedom of choice) and drawbacks (others can’t use your programs unless they have the same GUI toolkit installed). Fortunately, there is no conflict between the various GUI toolkits available for Python, so you can install as many different GUI toolkits as you want.

This chapter gives a brief introduction to one of the most mature cross-platform GUI toolkits for Python, called wxPython. For a more thorough introduction to wxPython programming, consult the official documentation (<http://wxpython.org>). For some more information about GUI programming, see Chapter 28.

## A Plethora of Platforms

Before writing a GUI program in Python, you need to decide which GUI platform you want to use. Simply put, a platform is one specific set of graphical components, accessible through a given Python module, called a GUI toolkit. As noted earlier, many such toolkits are available for Python. Some of the most popular ones are listed in Table 12-1. For an even more detailed list, you could search the Vaults of Parnassus (<http://py.vaults.ca/>) for the keyword “GUI.” An extensive list of toolkits can also be found in the Python Wiki (<http://wiki.python.org/moin/GuiProgramming>), and Guilherme Polo has written a paper comparing four major platforms.<sup>1</sup>

**Table 12-1.** Some Popular GUI Toolkits Available for Python

Package	Description	Web Site
Tkinter	Uses the Tk platform. Readily available. Semistandard.	<a href="http://wiki.python.org/moin/TkInter">http://wiki.python.org/moin/TkInter</a>
wxPython	Based on wxWindows. Cross-platform. Increasingly popular.	<a href="http://wxpython.org">http://wxpython.org</a>

*Continued*

1. “PyGTK, PyQt, Tkinter and wxPython comparison,” *The Python Papers*, Volume 3, Issue 1, pages 26–37. Available from <http://pythonpapers.org>.

**Table 12-1.** *Continued*

Package	Description	Web Site
PythonWin	Windows only. Uses native Windows GUI capabilities.	<a href="http://starship.python.net/crew/mhammond">http://starship.python.net/crew/mhammond</a>
Java Swing	Jython only. Uses native Java GUI capabilities.	<a href="http://java.sun.com/docs/books/tutorial/uiswing">http://java.sun.com/docs/books/tutorial/uiswing</a>
PyGTK	Uses the GTK platform. Especially popular on Linux.	<a href="http://pygtk.org">http://pygtk.org</a>
PyQt	Uses the Qt platform. Cross-platform.	<a href="http://wiki.python.org/moin/PyQt">http://wiki.python.org/moin/PyQt</a>

So which GUI toolkit should you use? It is largely a matter of taste, although each toolkit has its advantages and drawbacks. Tkinter is sort of a *de facto* standard because it has been used in most “official” Python GUI programs, and it is included as a part of the Windows binary distribution. On UNIX, however, you need to compile and install it yourself. I’ll cover Tkinter, as well as Java Swing, in the section “But I’d Rather Use . . .” later in this chapter.

Another toolkit that is gaining in popularity is wxPython. This is a mature and feature-rich toolkit, which also happens to be the favorite of Python’s creator, Guido van Rossum. We’ll use wxPython for this chapter’s example.

For information about PythonWin, PyGTK, and PyQt, check out the project home pages (see Table 12-1).

## Downloading and Installing wxPython

To download wxPython, simply visit the download page, <http://wxpython.org/download.php>. This page gives you detailed instructions about which version to download, as well as the prerequisites for the various versions.

If you’re running Windows, you probably want a prebuilt binary. You can choose between one version with Unicode support and one without; unless you know you need Unicode, it probably won’t make much of a difference which one you choose. Make sure you choose the binary that corresponds to your version of Python. A version of wxPython compiled for Python 2.3 won’t work with Python 2.4, for example.

For Mac OS X, you should again choose the wxPython version that agrees with your Python version. You might also need to take the OS version into consideration. Again, you may need to choose between a version with Unicode support and one without; just take your pick. The download links and associated explanations should make it perfectly clear which version you need.

If you're using Linux, you could check to see if your package manager has wxPython. It should be present in most mainstream distributions. There are also RPM packages for various flavors of Linux. If you're running a Linux distribution with RPM, you should at least download the wxPython common and runtime packages; you probably won't need the devel package. Again, choose the version corresponding to your Python version and Linux distribution.

If none of the binaries fit your hardware or operating system (or Python version, for that matter), you can always download the source distribution. Getting this to compile might require downloading other source packages for various prerequisites. You'll find fairly detailed explanations on the wxPython download page.

Once you have wxPython itself, I strongly suggest that you download the demo distribution, which contains documentation, sample programs, and one very thorough (and instructive) demo program. This demo program exercises most of the wxPython features, and lets you see the source code for each portion in a very user-friendly manner—definitely worth a look if you want to keep learning about wxPython on your own.

Installation should be fairly automatic and painless. To install Windows binaries, simply run the downloaded executables (.exe files). In OS X, the downloaded file should appear as if it were a CD-ROM that you can open, with a .pkg you can double-click. To install using RPM, consult your RPM documentation. Both the Windows and Mac OS X versions will start an installation wizard, which should be simple to follow. Simply accept all default settings, keep clicking Continue, and, finally, click Finish.

To see whether your installation works, you could try out the wxPython demo (which must be installed separately). In Windows, it should be available in your Start menu. When installing it in OS X, you could simply drag the wxPython Demo file to Applications, and then run it from there later. Once you've finished playing with the demo (for now, anyway), you can get started writing your own program, which is, of course, much more fun.

## Building a Sample GUI Application

To demonstrate using wxPython, I will show you how to build a simple GUI application. Your task is to write a basic program that enables you to edit text files. We aren't going to write a full-fledged text editor, but instead stick to the essentials. After all, the goal is to demonstrate the basic mechanisms of GUI programming in Python.

The requirements for this minimal text editor are as follows:

- It must allow you to open text files, given their file names.
- It must allow you to edit the text files.
- It must allow you to save the text files.
- It must allow you to quit.

When writing a GUI program, it's often useful to draw a sketch of how you want it to look. Figure 12-1 shows a simple layout that satisfies the requirements for our text editor.



**Figure 12-1.** A sketch of the text editor

The elements of the interface can be used as follows:

- Type a file name in the text field to the left of the buttons and click Open to open a file. The text contained in the file is put in the text field at the bottom.
- You can edit the text to your heart's content in the large text field.
- If and when you want to save your changes, click the Save button, which again uses the text field containing the file name, and writes the contents of the large text field to the file.
- There is no Quit button. If you close the window, the program quits.

In some languages, writing a program like this is a daunting task, but with Python and the right GUI toolkit, it's really a piece of cake. (You may not agree with me right now, but by the end of this chapter, I hope you will.)

## Getting Started

To get started, import the `wx` module:

```
import wx
```

There are several ways of writing wxPython programs, but one thing you can't escape is creating an application object. The basic application class is called `wx.App`, and it takes care of all kinds of initialization behind the scenes. The simplest wxPython program would be something like this:

```
import wx
app = wx.App()
app.MainLoop()
```

---

**Note** If you’re having trouble getting `wx.App` to work, you may want to try to replace it with `wx.PySimpleApp`.

---

Because there are no windows the user can interact with, the program exits immediately.

As you can see from this example, the methods in the `wx` package are written with an initial uppercase character, contrary to common practice in Python. The reason for this is that the method names mirror method names from the underlying C++ package, `wxWidgets`. Even though there is no formal rule against initial cap method or function names, the norm is to reserve such names for classes.

## Windows and Components

Windows, also known as *frames*, are simply instances of the `wx.Frame` class. Widgets in the `wx` framework are created with their *parent* as the first argument to their constructor. If you’re creating an individual window, there will be no parent to consider, so simply use `None`, as you see in Listing 12-1. Also, make sure you call the window’s `Show` method before you call `app.MainLoop`; otherwise, it will remain hidden. (You could also call `win.Show` in an event handler, as discussed a bit later.)

**Listing 12-1.** *Creating and Showing a Frame*

```
import wx
app = wx.App()
win = wx.Frame(None)
win.Show()
app.MainLoop()
```

If you run this program, you should see a single window appear, similar to that in Figure 12-2.



**Figure 12-2.** A GUI program with only one window

Adding a button to this frame is about as simple as it can be—simply instantiate `wx.Button`, using `win` as the parent argument, as shown in Listing 12-2.

**Listing 12-2.** Adding a Button to a Frame

```
import wx
app = wx.App()
win = wx.Frame(None)
btn = wx.Button(win)
win.Show()
app.MainLoop()
```

This will give you a window with a single button, as shown in Figure 12-3.



**Figure 12-3.** The program after adding a button

This certainly is quite rough. The window has no title, the button has no label, and you probably don't want the button to cover the entire window in this way.

## Labels, Titles, and Positions

You can set the labels of widgets when you create them, by using the `label` argument of the constructor. Similarly, you can set the titles of frames by using the `title` argument. I find it most practical to use keyword arguments with the `wx` constructors, so I don't need to remember their order. You can see an example of this in Listing 12-3.

**Listing 12-3.** Adding Labels and Titles with Keyword Arguments

```
import wx

app = wx.App()
win = wx.Frame(None, title="Simple Editor")

loadButton = wx.Button(win, label='Open')

saveButton = wx.Button(win, label='Save')

win.Show()

app.MainLoop()
```

The result of running this program should be something like what you see in Figure 12-4.



**Figure 12-4.** A window with layout problems

Something isn't quite right about this version of the program: one button seems to be missing! Actually, it's not missing—it's just hiding. By placing the buttons more carefully, you should be able to uncover the hidden button. A very basic (and not very practical) method is to simply set positions and size by using the `pos` and `size` arguments to the constructors, as in the code presented in Listing 12-4.

#### **Listing 12-4.** Setting Button Positions

```
import wx

app = wx.App()
win = wx.Frame(None, title="Simple Editor", size=(410, 335))
win.Show()

loadButton = wx.Button(win, label='Open',
                      pos=(225, 5), size=(80, 25))

saveButton = wx.Button(win, label='Save',
                      pos=(315, 5), size=(80, 25))

filename = wx.TextCtrl(win, pos=(5, 5), size=(210, 25))

contents = wx.TextCtrl(win, pos=(5, 35), size=(390, 260),
                      style=wx.TE_MULTILINE | wx.HSCROLL)

app.MainLoop()
```

As you can see, both position and size are pairs of numbers. The position is a pair of x and y coordinates, while the size consists of width and height.

This piece of code has a couple other new things: I've created a couple of *text controls* (`wx.TextCtrl` objects) and given one of them a custom *style*. The default text control is a *text field*, with a single line of editable text, and no scroll bar. In order to create a *text area*, you can simply tweak the style with the `style` parameter. The style is actually a single integer, but

you don't need to specify it directly. Instead, you use bitwise OR (the pipe) to combine various style facets that are available under special names from the `wx` module. In this case, I've combined `wx.TE_MULTILINE`, to get a multiline text area (which, by default, has a vertical scroll bar), and `wx.HSCROLL`, to get a horizontal scroll bar. The result of running this program is shown in Figure 12-5.



**Figure 12-5.** Properly positioned components

## More Intelligent Layout

Although specifying the geometry of each component is easy to understand, it can be a bit tedious. Doodling a bit on graph paper may help in getting the coordinates right, but there are more serious drawbacks to this approach than having to play around with numbers. If you run the program and try to resize the window, you'll notice that the geometries of the components don't change. This is no disaster, but it does look a bit odd. When you resize a window, you assume that its contents will be resized and relocated as well.

If you consider how I did the layout, this behavior shouldn't really come as a surprise. I explicitly set the position and size of each component, but didn't say anything about how they should behave when the window was resized. There are many ways of specifying this. One of the easiest ways of doing layout in `wx` is using *sizers*, and the easiest one to use is `wx.BoxSizer`.

A sizer manages the size of contents. You simply add widgets to a sizer, together with a few layout parameters, and then give this sizer the job of managing the layout of the parent component. In our case, we'll add a background component (a `wx.Panel`), create some nested `wx.BoxSizers`, and then set the sizer of the panel with its `SetSizer` method, as shown in Listing 12-5.

**Listing 12-5.** Using a Sizer

```
import wx

app = wx.App()
win = wx.Frame(None, title="Simple Editor", size=(410, 335))
```

```
bkg = wx.Panel(win)

loadButton = wx.Button(bkg, label='Open')
saveButton = wx.Button(bkg, label='Save')
filename = wx.TextCtrl(bkg)
contents = wx.TextCtrl(bkg, style=wx.TE_MULTILINE | wx.HSCROLL)

hbox = wx.BoxSizer()
hbox.Add(filename, proportion=1, flag=wx.EXPAND)
hbox.Add(loadButton, proportion=0, flag=wx.LEFT, border=5)
hbox.Add(saveButton, proportion=0, flag=wx.LEFT, border=5)

vbox = wx.BoxSizer(wx.VERTICAL)
vbox.Add(hbox, proportion=0, flag=wx.EXPAND | wx.ALL, border=5)
vbox.Add(contents, proportion=1,
         flag=wx.EXPAND | wx.LEFT | wx.BOTTOM | wx.RIGHT, border=5)

bkg.SetSizer(vbox)
win.Show()

app.MainLoop()
```

This code gives the same result as the previous program, but instead of using lots of absolute coordinates, I am now placing things in relation to one another.

The constructor of the `wx.BoxSizer` takes an argument determining whether it's horizontal or vertical (`wx.HORIZONTAL` or `wx.VERTICAL`), with horizontal being the default. The `Add` method takes several arguments. The `proportion` argument sets the proportions according to which space is allocated when the window is resized. For example, in the horizontal box sizer (the first one), the `filename` widget gets all of the extra space when resizing. If each of the three had its `proportion` set to 1, each would get an equal share. You can set the `proportion` to any number.

The `flag` argument is similar to the `style` argument of the constructor. You construct it by using bitwise OR between symbolic constants (integers that have special names). The `wx.EXPAND` flag makes sure the component will expand into the allotted space. The `wx.LEFT`, `wx.RIGHT`, `wx.TOP`, `wx.BOTTOM`, and `wx.ALL` flags determine on which sides the `border` argument applies, and the `border` arguments gives the width of the border (spacing).

And that's it. I've got the layout I wanted. One crucial thing is lacking, however. If you click the buttons, nothing happens.

---

**Tip** For more information about sizers, or anything else related to wxPython, check out the wxPython demo. It has sample code for anything you might want to know about, and then some. If that seems daunting, check out the wxPython web site, <http://wxpython.org>.

---

## Event Handling

In GUI lingo, the actions performed by the user (such as clicking a button) are called *events*. You need to make your program notice these events somehow, and then react to them. You accomplish this by binding a function to the widget where the event in question might occur. When the event does occur (if ever), that function will then be called. You link the event handler to a given event with a widget method called `Bind`.

Let's assume that you have written a function responsible for opening a file, and you've called it `load`. Then you can use that as an event handler for `loadButton` as follows:

```
loadButton.Bind(wx.EVT_BUTTON, load)
```

This is pretty intuitive, isn't it? I've linked a function to the button—when the button is clicked, the function is called. The symbolic constant `wx.EVT_BUTTON` signifies a *button event*. The `wx` framework has such event constants for all kinds of events, from mouse motion to keyboard presses and more.

---

**Note** There is nothing magical about my choice to use `loadButton` and `load` as the button and handler names, even though the button text says "Open." It's just that if I had called the button `openButton`, `open` would have been the natural name for the handler, and that would have made the built-in file-opening function `open` unavailable. While there are ways of dealing with this, I found it easier to use a different name.

---

## The Finished Program

Let's fill in the remaining blanks. All you need now are the two event handlers, `load` and `save`. When an event handler is called, it receives a single event object as its only parameter, which holds information about what happened. But let's ignore that here, because you're only interested in the fact that a click occurred.

Even though the event handlers are the meat of the program, they are surprisingly simple. Let's take a look at the `load` function first. It looks like this:

```
def load(event):
    file = open(filename.GetValue())
    contents.SetValue(file.read())
    file.close()
```

The file opening/reading part should be familiar from Chapter 11. As you can see, the file name is found by using `filename`'s `GetValue` method (where `filename` is the small text field, remember?). Similarly, to put the text into the text area, you simply use `contents.SetValue`.

The `save` function is just as simple. It's the exact same as `load`, except that it has a '`w`' and a `write` for the file-handling part, and `GetValue` for the text area:

```
def save(event):
    file = open(filename.GetValue(), 'w')
    file.write(contents.GetValue())
    file.close()
```

And that's it. Now I simply bind these to their respective buttons, and the program is ready to run. See Listing 12-6 for the final program.

**Listing 12-6.** *The Final GUI Program*

```
import wx
def load(event):
    file = open(filename.GetValue())
    contents.SetValue(file.read())
    file.close()

def save(event):
    file = open(filename.GetValue(), 'w')
    file.write(contents.GetValue())
    file.close()

app = wx.App()
win = wx.Frame(None, title="Simple Editor", size=(410, 335))

bkg = wx.Panel(win)

loadButton = wx.Button(bkg, label='Open')
loadButton.Bind(wx.EVT_BUTTON, load)

saveButton = wx.Button(bkg, label='Save')
saveButton.Bind(wx.EVT_BUTTON, save)

filename = wx.TextCtrl(bkg)
contents = wx.TextCtrl(bkg, style=wx.TE_MULTILINE | wx.HSCROLL)

hbox = wx.BoxSizer()
hbox.Add(filename, proportion=1, flag=wx.EXPAND)
hbox.Add(loadButton, proportion=0, flag=wx.LEFT, border=5)
hbox.Add(saveButton, proportion=0, flag=wx.LEFT, border=5)

vbox = wx.BoxSizer(wx.VERTICAL)
vbox.Add(hbox, proportion=0, flag=wx.EXPAND | wx.ALL, border=5)
vbox.Add(contents, proportion=1,
         flag=wx.EXPAND | wx.LEFT | wx.BOTTOM | wx.RIGHT, border=5)

bkg.SetSizer(vbox)
win.Show()

app.MainLoop()
```

You can try out the editor using the following steps:

1. Run the program. You should get a window like the one in the previous runs.
2. Type something in the large text area (for example, “Hello, world!”).
3. Type a file name in the small text field (for example, `hello.txt`). Make sure that this file does not already exist or it will be overwritten.
4. Click the Save button.
5. Close the editor window (just for fun).
6. Restart the program.
7. Type the same file name in the little text field.
8. Click the Open button. The text of the file should reappear in the large text area.
9. Edit the file to your heart’s content, and save it again.

Now you can keep opening, editing, and saving until you grow tired of that. Then you can start thinking of improvements. How about allowing your program to download files with the `urllib` module, for example?

You might also consider using more object-oriented design in your programs, of course. For example, you may want to manage the main application as an instance of a custom application class (a subclass of `wx.App`, perhaps?), and instead of setting up your layout at the top level of your program, you could make a separate window class (a subclass of `wx.Frame`?). See Chapter 28 for some examples.

### HEY! WHAT ABOUT PYW?

In Windows, you could save your GUI programs with a `.pyw` ending. In Chapter 1, I asked you to give your file this ending and double-click it (in Windows). Nothing happened then, and I promised to explain it later. In Chapter 10, I mentioned it again, and said I would explain it in this chapter. So I will.

It’s no big deal, really. It’s just that when you double-click an ordinary Python script in Windows, a DOS window appears with a Python prompt in it. That’s fine if you use `print` and `raw_input` as the basis of your interface, but now that you know how to make GUIs, this DOS window will only be in your way. The truth behind the `.pyw` window is that it will run Python without the DOS window, which is just perfect for GUI programs.

## But I’d Rather Use . . .

As you’ve learned, you can choose from many GUI toolkits for Python. Here, I will give you some examples from a couple of the more popular ones: Tkinter and Jython/Swing.

To illustrate these toolkits, I’ve created a simple example—simpler, even, than the editor example you just completed. It’s just a single window containing a single button with the label

“Hello” filling the window. When you click the button, it prints out the words “Hello, world!” In the interest of simplicity, I’m not using any fancy layout features here. Here is a simple wxPython version:

```
import wx

def hello(event):
    print "Hello, world!"

app = wx.App()

win = wx.Frame(None, title="Hello, wxPython!", size=(200, 100))
button = wx.Button(win, label="Hello")
button.Bind(wx.EVT_BUTTON, hello)

win.Show()
app.MainLoop()
```

The resulting window is shown in Figure 12-6.



**Figure 12-6.** A simple GUI example

## Using Tkinter

Tkinter is an old-timer in the Python GUI business. It is a wrapper around the Tk GUI toolkit (associated with the programming language Tcl). It is included by default in the Windows and Mac OS distributions. The following URLs may be useful:

- <http://www.ibm.com/developerworks/linux/library/l-tkprg>
- <http://www.nmt.edu/tcc/help/lang/python/tkinter.pdf>

Here is the GUI example implemented with Tkinter:

```
from Tkinter import *
def hello(): print 'Hello, world'
win = Tk() # Tkinter's 'main window'
win.title('Hello, Tkinter! ')
win.geometry('200x100') # Size 200, 100

btn = Button(win, text='Hello ', command=hello)
btn.pack(expand=YES, fill=BOTH)

mainloop()
```

## Using Jython and Swing

If you're using Jython (the Java implementation of Python), packages such as wxPython and Tkinter aren't available. The only GUI toolkits that are readily available are the Java standard library packages Abstract Window Toolkit (AWT) and Swing (Swing is the most recent and considered the standard Java GUI toolkit). The good news is that both of these are automatically available so you don't need to install them separately. For more information, visit the Jython web site and look into the Swing documentation written for Java:

- <http://www.jython.org>
- <http://java.sun.com/docs/books/tutorial/uiswing>

Here is the GUI example implemented with Jython and Swing:

```
from javax.swing import *
import sys

def hello(event): print 'Hello, world! '
btn = JButton('Hello')
btn.actionPerformed = hello

win = JFrame('Hello, Swing!')
win.contentPane.add(btn)

def closeHandler(event): sys.exit()
win.windowClosing = closeHandler

btn.size = win.size = 200, 100
win.show()
```

Note that one additional event handler has been added here (`closeHandler`) because the Close button doesn't have any useful default behavior in Java Swing. Also note that you don't need to explicitly enter the main event loop because it's running in parallel with the program (in a separate thread).

## Using Something Else

The basics of most GUI toolkits are the same. Unfortunately, however, when learning how to use a new package, it takes time to find your way through all the details that enable you to do exactly what you want. So you should take your time before deciding which package you want to work with (the section "A Plethora of Platforms" earlier in this chapter should give you some idea of where to start), and then immerse yourself in its documentation and start writing code. I hope this chapter has provided the basic concepts you need to make sense of that documentation.

## A Quick Summary

Once again, let's review what we've covered in this chapter:

**Graphical user interfaces (GUIs):** GUIs are useful in making your programs more user friendly. Not all programs need them, but whenever your program interacts with a user, a GUI is probably helpful.

**GUI platforms for Python:** Many GUI platforms are available to the Python programmer. Although this richness is definitely a boon, choosing between them can sometimes be difficult.

**wxPython:** wxPython is a mature and feature-rich cross-platform GUI toolkit for Python.

**Layout:** You can position components quite simply by specifying their geometry directly. However, to make them behave properly when their containing window is resized, you will need to use some sort of layout manager. One common layout mechanism in wxPython is *sizers*.

**Event handling:** Actions performed by the user trigger *events* in the GUI toolkit. To be of any use, your program will probably be set up to react to some of these events; otherwise, the user won't be able to interact with it. In wxPython, event handlers are added to components with the `Bind` method.

## What Now?

That's it. You now know how to write programs that can interact with the outside world through files and GUIs. In the next chapter, you learn about another important component of many program systems: databases.

# Chapter 8

## Network Programming



# Network Programming

In this chapter, I give you a sample of the various ways in which Python can help you write programs that use a network, such as the Internet, as an important component. Python is a very powerful tool for network programming. Many libraries for common network protocols and for various layers of abstractions on top of them are available, so you can concentrate on the logic of your program, rather than on shuffling bits across wires. Also, it's easy to write code for handling various protocol formats that may *not* have existing code, because Python's really good at tackling patterns in byte streams (you've already seen this in dealing with text files in various ways).

Because Python has such an abundance of network tools available for you to use, I can only give you a brief peek at its networking capabilities here. You can find some other examples elsewhere in this book. Chapter 15 includes a discussion of web-oriented network programming, and several of the projects in later chapters use networking modules to get the job done. If you want to know even *more* about network programming in Python, I can heartily recommend John Goerzen's *Foundations of Python Network Programming* (Apress, 2004), which deals with the subject very thoroughly.

In this chapter, I give you an overview of some of the networking modules available in the Python standard library. Then comes a discussion of the `SocketServer` class and its friends, followed by a brief look at the various ways in which you can handle several connections at once. Finally, I give you a look at the Twisted framework, a rich and mature framework for writing networked applications in Python.

---

**Note** If you have a strict firewall in place, it will probably warn you once you start running your own network programs and stop them from connecting to the network. You should either configure your firewall to let your Python do its work, or, if the firewall has an interactive interface (such as the Windows XP firewall), simply allow the connections when asked. Note, though, that any software connected to a network is a potential security risk, even if (or especially if) you wrote the software yourself.

---

## A Handful of Networking Modules

You can find plenty of networking modules in the standard library, and many more elsewhere. In addition to those that clearly deal mainly with networking, several modules (such as those

that deal with various forms of data encoding for network transport) may be seen as network related. I've been fairly restrictive in my selection of modules here.

## The socket Module

A basic component in network programming is the *socket*. A socket is basically an “information channel” with a program on both ends. The programs may be on different computers (connected through a network) and may send information to each other through the socket. Most network programming in Python hides the basic workings of the socket module and doesn’t interact with the sockets directly.

Sockets come in two varieties: server sockets and client sockets. After you create a server socket, you tell it to wait for connections. It will then listen at a certain network address (a combination of an IP address and a port number) until a client socket connects. The two can then communicate.

Dealing with client sockets is usually quite a bit easier than dealing with the server side, because the server must be ready to deal with clients whenever they connect, and it must deal with multiple connections, while the client simply connects, does its thing, and disconnects. Later in this chapter, I discuss server programming through the `SocketServer` class family and the Twisted framework.

A socket is an instance of the `socket` class from the `socket` module. It is instantiated with up to three parameters: an address family (defaulting to `socket.AF_INET`), whether it’s a stream (`socket.SOCK_STREAM`, the default) or a datagram (`socket.SOCK_DGRAM`) socket, and a protocol (defaulting to 0, which should be okay). For a plain-vanilla socket, you don’t really need to supply any arguments.

A server socket uses its `bind` method followed by a call to `listen` to listen to a given address. A client socket can then connect to the server by using its `connect` method with the same address as used in `bind`. (On the server side, you can, for example, get the name of the current machine using the function `socket.gethostname()`.) In this case, an address is just a tuple of the form `(host, port)`, where `host` is a host name (such as `www.example.com`) and `port` is a port number (an integer). The `listen` method takes a single argument, which is the length of its backlog (the number of connections allowed to queue up, waiting for acceptance, before connections start being disallowed).

Once a server socket is listening, it can start accepting clients. This is done using the `accept` method. This method will block (wait) until a client connects, and then it will return a tuple of the form `(client, address)`, where `client` is a client socket and `address` is an address, as explained earlier. The server can deal with the client as it sees fit, and then start waiting for new connections, with another call to `accept`. This is usually done in an infinite loop.

---

**Note** The form of server programming discussed here is called *blocking* or *synchronous* network programming. In the section “Multiple Connections” later in this chapter, you’ll see examples of nonblocking or asynchronous network programming, as well as using threads to be able to deal with several clients at once.

---

For transmitting data, sockets have two methods: `send` and `recv` (for “receive”). You can call `send` with a string argument to send data, and `recv` with a desired (maximum) number of bytes to receive data. If you’re not sure which number to use, 1024 is as good a choice as any.

Listings 14-1 and 14-2 show an example client/server pair that is about as simple as it gets. If you run them on the same machine (starting the server first), the server should print out a message about getting a connection, and the client should then print out a message it has received from the server. You can run several clients while the server is still running. By replacing the call to `gethostname` in the client with the actual host name of the machine where the server is running, you can have the two programs connect across a network from one machine to another.

---

**Note** The port numbers you use are normally restricted. In a Linux or UNIX system, you need administrator privileges to use a port below 1024. These low-numbered ports are used for standard services, such as port 80 for your web server (if you have one). Also, if you stop a server with Ctrl+C, for example, you might need to wait for a bit before using the same port number again (you may get an “Address already in use” error).

---

**Listing 14-1. A Minimal Server**

```
import socket

s = socket.socket()

host = socket.gethostname()
port = 1234
s.bind((host, port))

s.listen(5)
while True:
    c, addr = s.accept()
    print 'Got connection from', addr
    c.send('Thank you for connecting')
    c.close()
```

**Listing 14-2. A Minimal Client**

```
import socket

s = socket.socket()

host = socket.gethostname()
port = 1234

s.connect((host, port))
print s.recv(1024)
```

You can find more information about the `socket` module in the Python Library Reference (<http://python.org/doc/lib/module-socket.html>) and in Gordon McMillan's *Socket Programming HOWTO* (<http://docs.python.org/dev/howto/sockets.html>).

## The `urllib` and `urllib2` Modules

Of the networking libraries available, the ones that probably give you the most bang for the buck are `urllib` and `urllib2`. They enable you to access files across a network, just as if they were located on your computer. Through a simple function call, virtually anything you can refer to with a Uniform Resource Locator (URL) can be used as input to your program. Just imagine the possibilities you get if you combine this with the `re` module: you can download web pages, extract information, and create automatic reports of your findings.

The two modules do more or less the same job, with `urllib2` being a bit more "fancy." For simple downloads, `urllib` is quite all right. If you need HTTP authentication or cookies, or you want to write extensions to handle your own protocols, then `urllib2` might be the right choice for you.

### Opening Remote Files

You can open remote files almost exactly as you do local files; the difference is that you can use only read mode, and instead of `open` (or `file`), you use `urlopen` from the `urllib` module:

```
>>> from urllib import urlopen  
>>> webpage = urlopen('http://www.python.org')
```

If you are online, the variable `webpage` should now contain a file-like object linked to the Python web page at <http://www.python.org>.

---

**Note** If you want to experiment with `urllib` but aren't currently online, you can access local files with URLs that start with `file:`, such as `file:c:\text\somefile.txt`. (Remember to escape your backslashes.)

---

The file-like object that is returned from `urlopen` supports (among others) the `close`, `read`, `readline`, and `readlines` methods, as well as iteration.

Let's say you want to extract the (relative) URL of the "About" link on the Python page you just opened. You could do that with regular expressions (for more information about regular expressions, see the section about the `re` module in Chapter 10):

```
>>> import re  
>>> text = webpage.read()  
>>> m = re.search('<a href="([^\"]+)" .*?>about</a>', text, re.IGNORECASE)  
>>> m.group(1)  
'/about/'
```

---

**Note** You may need to modify the regular expression if the web page has changed since the time of writing, of course.

---

## Retrieving Remote Files

The `urlopen` function gives you a file-like object you can read from. If you would rather have `urllib` take care of downloading the file for you, storing a copy in a local file, you can use `urlretrieve` instead. Rather than returning a file-like object, it returns a tuple (`filename`, `headers`), where `filename` is the name of the local file (this name is created automatically by `urllib`), and `headers` contains some information about the remote file. (I'll ignore `headers` here; look up `urlretrieve` in the standard library documentation of `urllib` if you want to know more about it.) If you want to specify a file name for the downloaded copy, you can supply that as a second parameter:

```
urlretrieve('http://www.python.org', 'C:\\python_webpage.html')
```

This retrieves the Python home page and stores it in the file `C:\python_webpage.html`. If you don't specify a file name, the file is put in some temporary location, available for you to open (with the `open` function), but when you're finished with it, you may want to have it removed so that it doesn't take up space on your hard drive. To clean up such temporary files, you can call the function `urlcleanup` without any arguments, and it takes care of things for you.

## SOME UTILITIES

In addition to reading and downloading files through URLs, `urllib` also offers some functions for manipulating the URLs themselves. (The following assumes some knowledge of URLs and CGI.) The following functions are available:

- `quote(string[, safe])`: Returns a string in which all special characters (characters that have special significance in URLs) have been replaced by URL-friendly versions (such as `%7E` instead of `~`). This can be useful if you have a string that might contain such special characters and you want to use it as a URL. The `safe` string includes characters that should not be coded like this. The default is `'/'`.
- `quote_plus(string[, safe])`: Works like `quote`, but also replaces spaces with plus signs.
- `unquote(string)`: The reverse of `quote`.
- `unquote_plus(string)`: The reverse of `quote_plus`.
- `urlencode(query[, doseq])`: Converts a mapping (such as a dictionary) or a sequence of two-element tuples—of the form `(key, value)`—into a “URL-encoded” string, which can be used in CGI queries. (Check the Python documentation for more information.)

## Other Modules

As mentioned, beyond the modules explicitly discussed in this chapter, there are hordes of network-related modules in the Python library and elsewhere. Table 14-1 lists some network-related modules from the Python standard library. As noted in the table, some of these modules are discussed elsewhere in the book.

**Table 14-1. Some Network-Related Modules in the Standard Library**

Module	Description
asynchat	Additional functionality for asyncore (see Chapter 24)
asyncore	Asynchronous socket handler (see Chapter 24)
cgi	Basic CGI support (see Chapter 15)
Cookie	Cookie object manipulation, mainly for servers
cookielib	Client-side cookie support
email	Support for e-mail messages (including MIME)
ftplib	FTP client module
gopherlib	Gopher client module
httplib	HTTP client module
imaplib	IMAP4 client module
mailbox	Reads several mailbox formats
mailcap	Access to MIME configuration through mailcap files
mhlib	Access to MH mailboxes
nntplib	NNTP client module (see Chapter 23)
poplib	POP client module
robotparser	Support for parsing web server robot files
SimpleXMLRPCServer	A simple XML-RPC server (see Chapter 27)
smtpd	SMTP server module
smtplib	SMTP client module
telnetlib	Telnet client module
urlparse	Support for interpreting URLs
xmlrpclib	Client support for XML-RPC (see Chapter 27)

## SocketServer and Friends

As you saw in the section about the `socket` module earlier, writing a simple socket server isn't really hard. If you want to go beyond the basics, however, getting some help can be nice. The `SocketServer` module is the basis for a framework of several servers in the standard library,

including BaseHTTPServer, SimpleHTTPServer, CGIHTTPServer, SimpleXMLRPCServer, and DocXMLRPCServer, all of which add various specific functionality to the basic server.

SocketServer contains four basic classes: TCPServer, for TCP socket streams; UDPServer, for UDP datagram sockets; and the more obscure UnixStreamServer and UnixDatagramServer. You probably won't need the last three.

To write a server using the SocketServer framework, you put most of your code in a request handler. Each time the server gets a request (a connection from a client), a request handler is instantiated, and various handler methods are called on it to deal with the request. Exactly which methods are called depends on the specific server and handler class used, and you can subclass them to make the server call a custom set of handlers. The basic BaseRequestHandler class places all of the action in a single method on the handler, called handle, which is called by the server. This method then has access to the client socket in the attribute self.request. If you're working with a stream (which you probably are, if you use TCPServer), you can use the class StreamRequestHandler, which sets up two other attributes, self.rfile (for reading) and self.wfile (for writing). You can then use these file-like objects to communicate with the client.

The various other classes in the SocketServer framework implement basic support for HTTP servers, including running CGI scripts, as well as support for XML-RPC (discussed in Chapter 27).

Listing 14-3 gives you a SocketServer version of the minimal server from Listing 14-1. It can be used with the client in Listing 14-2. Note that the StreamRequestHandler takes care of closing the connection when it has been handled. Also note that giving '' as the host name means that you're referring to the machine where the server is running.

#### **Listing 14-3. A SocketServer-Based Minimal Server**

```
from SocketServer import TCPServer, StreamRequestHandler

class Handler(StreamRequestHandler):

    def handle(self):
        addr = self.request.getpeername()
        print 'Got connection from', addr
        self.wfile.write('Thank you for connecting')

server = TCPServer(('', 1234), Handler)
server.serve_forever()
```

You can find more information about the SocketServer framework in the Python Library Reference (<http://python.org/doc/lib/module-SocketServer.html>) and in John Goerzen's *Foundations of Python Network Programming* (Apress, 2004).

## **Multiple Connections**

The server solutions discussed so far have been *synchronous*: only one client can connect and get its request handled at a time. If one request takes a bit of time, such as, for example, a complete chat session, it's important that more than one connection can be dealt with simultaneously.

You can deal with multiple connections in three main ways: forking, threading, and asynchronous I/O. Forking and threading can be dealt with very simply, by using mix-in classes with any of the `SocketServer` servers (see Listings 14-4 and 14-5). Even if you want to implement them yourself, these methods are quite easy to work with. They do have their drawbacks, however. Forking takes up resources, and may not scale well if you have many clients (although, for a reasonable number of clients, on modern UNIX or Linux systems, forking is quite efficient, and can be even more so if you have a multi-CPU system). Threading can lead to synchronization problems. I won't go into these problems in any detail here (nor will I discuss multithreading in depth), but I'll show you how to use the techniques in the following sections.

### FORKS? THREADS? WHAT'S ALL THIS, THEN?

Just in case you don't know about forking or threads, here is a little clarification. *Forking* is a UNIX term. When you fork a process (a running program), you basically duplicate it, and both resulting processes keep running from the current point of execution, each with its own copy of the memory (variables and such). One process (the original one) will be the *parent* process, while the other (the copy) will be the *child*. If you're a science fiction fan, you might think of parallel universes; the forking operation creates a fork in the timeline, and you end up with two universes (the two processes) existing independently. Luckily, the processes are able to determine whether they are the original or the child (by looking at the return value of the `fork` function), so they can act differently. (If they couldn't, what would be the point, really? Both processes would do the same job, and you would just bog down your computer.)

In a forking server, a child is forked off for every client connection. The parent process keeps listening for new connections, while the child deals with the client. When the client is satisfied, the child process simply exits. Because the forked processes run in parallel, the clients don't need to wait for each other.

Because forking can be a bit resource intensive (each forked process needs its own memory), an alternative exists: threading. *Threads* are lightweight processes, or subprocesses, all of them existing within the same (real) process, sharing the same memory. This reduction in resource consumption comes with a downside, though. Because threads share memory, you must make sure they don't interfere with the variables for each other, or try to modify the same things at the same time, creating a mess. These issues fall under the heading of "synchronization." With modern operating systems (except Microsoft Windows, which doesn't support forking), forking is actually quite fast, and modern hardware can deal with the resource consumption much better than before. If you don't want to bother with synchronization issues, then forking may be a good alternative.

The best thing may, however, be to avoid this sort of parallelism altogether. In this chapter, you find other solutions, based on the `select` function. Another way to avoid threads and forks is to switch to Stackless Python (<http://stackless.com>), a version of Python designed to be able to switch between different contexts quickly and painlessly. It supports a form of thread-like parallelism called *microthreads*, which scale much better than real threads. For example, Stackless Python microthreads have been used in EVE Online (<http://www.eve-online.com>) to serve thousands of users.

Asynchronous I/O is a bit more difficult to implement at a low level. The basic mechanism is the `select` function of the `select` module (described in the section "Asynchronous I/O with `select` and `poll`"), which is quite hard to deal with. Luckily, frameworks exist that work with asynchronous I/O on a higher level, giving you a simple, abstract interface to a very powerful

and scalable mechanism. A basic framework of this kind, which is included in the standard library, consists of the `asyncore` and `asynchat` modules, discussed in Chapter 24. Twisted (which is discussed last in this chapter) is a very powerful asynchronous network programming framework.

## Forking and Threading with SocketServer

Creating a forking or threading server with the `SocketServer` framework is so simple it hardly needs any explanation. Listings 14-4 and 14-5 show you how to make the server from Listing 14-3 forking and threading, respectively. The forking or threading behavior is useful only if the `handle` method takes a long time to finish. Note that forking doesn't work in Windows.

### **Listing 14-4.** A Forking Server

```
from SocketServer import TCPServer, ForkingMixIn, StreamRequestHandler

class Server(ForkingMixIn, TCPServer): pass

class Handler(StreamRequestHandler):

    def handle(self):
        addr = self.request.getpeername()
        print 'Got connection from', addr
        self.wfile.write('Thank you for connecting')

server = Server(('', 1234), Handler)
server.serve_forever()
```

### **Listing 14-5.** A Threading Server

```
from SocketServer import TCPServer, ThreadingMixIn, StreamRequestHandler

class Server(ThreadingMixIn, TCPServer): pass

class Handler(StreamRequestHandler):

    def handle(self):
        addr = self.request.getpeername()
        print 'Got connection from', addr
        self.wfile.write('Thank you for connecting')

server = Server(('', 1234), Handler)
server.serve_forever()
```

## Asynchronous I/O with select and poll

When a server communicates with a client, the data it receives from the client may come in fits and spurts. If you're using forking and threading, that's not a problem. While one parallel waits

for data, other parallels may continue dealing with their own clients. Another way to go, however, is to deal only with the clients that actually have something to say at a given moment. You don't even have to hear them out—you just hear (or, rather, *read*) a little, and then put it back in line with the others.

This is the approach taken by the frameworks `asyncore/asynchat` (see Chapter 24) and Twisted (see the following section). The basis for this kind of functionality is the `select` function, or, where available, the `poll` function, both from the `select` module. Of the two, `poll` is more scalable, but it is available only in UNIX systems (that is, not in Windows).

The `select` function takes three sequences as its mandatory arguments, with an optional timeout in seconds as its fourth argument. The sequences are file descriptor integers (or objects with a `fileno` method that return such an integer). These are the connections that we're waiting for. The three sequences are for input, output, and exceptional conditions (errors and the like). If no timeout is given, `select` blocks (that is, waits) until one of the file descriptors is ready for action. If a timeout is given, `select` blocks for at most that many seconds, with zero giving a straight poll (that is, no blocking). `select` returns three sequences (a triple—that is, a tuple of length three), each representing an active subset of the corresponding parameter. For example, the first sequence returned will be a sequence of input file descriptors where there is something to read.

The sequences can, for example, contain file objects (not in Windows) or sockets.

Listing 14-6 shows a server using `select` to serve several connections. (Note that the server socket itself is supplied to `select`, so that it can signal when there are new connections ready to be accepted.) The server is a simple logger that prints out (locally) all data received from its clients. You can test it by connecting to it using telnet (or by writing a simple socket-based client that feeds it some data). Try connecting with multiple telnet connections to see that it can serve more than one client at once (although its log will then be a mixture of the input from the two).

**Listing 14-6. A Simple Server Using `select`**

```
import socket, select

s = socket.socket()

host = socket.gethostname()
port = 1234
s.bind((host, port))

s.listen(5)
inputs = [s]
while True:
    rs, ws, es = select.select(inputs, [], [])
    for r in rs:
        if r is s:
            c, addr = s.accept()
            print 'Got connection from', addr
            inputs.append(c)
```

```

else:
    try:
        data = r.recv(1024)
        disconnected = not data
    except socket.error:
        disconnected = True

    if disconnected:
        print r.getpeername(), 'disconnected'
        inputs.remove(r)
    else:
        print data

```

The `poll` method is easier to use than `select`. When you call `poll`, you get a `poll` object. You can then register file descriptors (or objects with a `fileno` method) with the `poll` object, using its `register` method. You can later remove such objects again, using the `unregister` method. Once you've registered some objects (for example, sockets), you can call the `poll` method (with an optional timeout argument) and get a list (possibly empty) of pairs of the form `(fd, event)`, where `fd` is the file descriptor and `event` tells you what happened. It's a bitmask, meaning that it's an integer where the individual bits correspond to various events. The various events are constants of the `select` module, and are explained in Table 14-2. To check whether a given bit is set (that is, if a given event occurred), you use the bitwise and operator (`&`), like this:

```
if event & select.POLLIN: ...
```

**Table 14-2.** Polling Event Constants in the `select` Module

Event Name	Description
POLLIN	There is data to read available from the file descriptor.
POLLPRI	There is urgent data to read from the file descriptor.
POLLOUT	The file descriptor is ready for data, and will not block if written to.
POLLERR	Some error condition is associated with the file descriptor.
POLLHUP	Hung up. The connection has been lost.
POLLNVAL	Invalid request. The connection is not open.

The program in Listing 14-7 is a rewrite of the server from Listing 14-6, now using `poll` instead of `select`. Note that I've added a map (`fdmap`) from file descriptors (ints) to socket objects.

**Listing 14-7.** A Simple Server Using `poll`

```

import socket, select

s = socket.socket()

```

```
host = socket.gethostname()
port = 1234
s.bind((host, port))

fdmap = {s.fileno(): s}

s.listen(5)
p = select.poll()
p.register(s)
while True:
    events = p.poll()
    for fd, event in events:
        if fd in fdmap:
            c, addr = s.accept()
            print 'Got connection from', addr
            p.register(c)
            fdmap[c.fileno()] = c
        elif event & select.POLLIN:
            data = fdmap[fd].recv(1024)
            if not data: # No data -- connection closed
                print fdmap[fd].getpeername(), 'disconnected'
                p.unregister(fd)
                del fdmap[fd]
        else:
            print data
```

You can find more information about `select` and `poll` in the Python Library Reference (<http://python.org/doc/lib/module-select.html>). Also, reading the source code of the standard library modules `asyncore` and `asynchat` (found in the `asyncore.py` and `asynchat.py` files in your Python installation) can be enlightening.

## Twisted

Twisted, from Twisted Matrix Laboratories (<http://twistedmatrix.com>), is an *event-driven* networking framework for Python, originally developed for network games but now used by all kinds of network software. In Twisted, you implement event handlers, much like you would in a GUI toolkit (see Chapter 12). In fact, Twisted works quite nicely together with several common GUI toolkits (Tk, GTK, Qt, and wxWidgets). In this section, I'll cover some of the basic concepts and show you how to do some relatively simple network programming using Twisted. Once you grasp the basic concepts, you can check out the Twisted documentation (available on the Twisted web site, along with quite a bit of other information) to do some more serious network programming. Twisted is a *very* rich framework and supports, among other things, web servers and clients, SSH2, SMTP, POP3, IMAP4, AIM, ICQ, IRC, MSN, Jabber, NNTP, DNS, and more!

## Downloading and Installing Twisted

Installing Twisted is quite easy. First, go to the Twisted Matrix web site (<http://twistedmatrix.com>) and, from there, follow one of the download links. If you’re using Windows, download the Windows installer for your version of Python. If you’re using some other system, download a source archive. (If you’re using a package manager such as Portage, RPM, APT, Fink, or MacPorts, you can probably get it to download and install Twisted directly.) The Windows installer is a self-explanatory step-by-step wizard. It may take some time compiling and unpacking things, but all you have to do is wait. To install the source archive, you first unpack it (using tar and then either gunzip or bunzip2, depending on which type of archive you downloaded), and then run the Distutils script:

```
python setup.py install
```

You should then be able to use Twisted.

## Writing a Twisted Server

The basic socket servers written earlier in this chapter are very explicit. Some of them have an explicit event loop, looking for new connections and new data. `SocketServer`-based servers have an implicit loop where the server looks for connections and creates a handler for each connection, but the handlers still must be explicit about trying to read data. Twisted (like the `asyncore/asynchat` framework, discussed in Chapter 24) uses an even more event-based approach. To write a basic server, you implement event handlers that deal with situations such as a new client connecting, new data arriving, and a client disconnecting (as well as many other events). Specialized classes can build more refined events from the basic ones, such as wrapping “data arrived” events, collecting the data until a newline is found, and then dispatching a “line of data arrived” event.

---

**Note** One thing I have not dealt with in this section, but which is somewhat characteristic of Twisted, is the concept of *deferreds* and deferred execution. See the Twisted documentation for more information (see, for example, the tutorial called “Deferreds are beautiful,” available from the HOWTO page of the Twisted documentation).

---

Your event handlers are defined in a protocol. You also need a factory that can construct such protocol objects when a new connection arrives. If you just want to create instances of a custom protocol class, you can use the factory that comes with Twisted, the `Factory` class in the module `twisted.internet.protocol`. When you write your protocol, use the `Protocol` from the same module as your superclass. When you get a connection, the event handler `connectionMade` is called. When you lose a connection, `connectionLost` is called. Data is received from the client through the handler `dataReceived`. Of course, you can’t use the event-handling strategy to send data back to the client—for that you use the object `self.transport`, which has a `write` method. It also has a `client` attribute, which contains the client address (host name and port).

Listing 14-8 contains a Twisted version of the server from Listings 14-6 and 14-7. I hope you agree that the Twisted version is quite a bit simpler and more readable. There is a little bit of setup involved; you need to instantiate `Factory` and set its `protocol` attribute so it knows

which protocol to use when communicating with clients (that is, your custom protocol). Then you start listening at a given port with that factory standing by to handle connections by instantiating protocol objects. You do this using the `listenTCP` function from the reactor module. Finally, you start the server by calling the `run` function from the same module.

**Listing 14-8.** A Simple Server Using Twisted

```
from twisted.internet import reactor
from twisted.internet.protocol import Protocol, Factory

class SimpleLogger(Protocol):

    def connectionMade(self):
        print 'Got connection from', self.transport.client

    def connectionLost(self, reason):
        print self.transport.client, 'disconnected'

    def dataReceived(self, data):
        print data

factory = Factory()
factory.protocol = SimpleLogger

reactor.listenTCP(1234, factory)
reactor.run()
```

If you connected to this server using telnet to test it, you may have gotten a single character on each line of output, depending on buffering and the like. You could simply use `sys.stdout.write` instead of `print`, but in many cases, you might like to get a single line at a time, rather than just arbitrary data. Writing a custom protocol that handles this for you would be quite easy, but there is, in fact, such a class available already. The module `twisted.protocols.basic` contains a couple of useful predefined protocols, among them `LineReceiver`. It implements `dataReceived` and calls the event handler `lineReceived` whenever a full line is received.

---

**Tip** If you need to do something when you receive data in *addition* to using `lineReceived`, which depends on the `LineReceiver` implementation of `dataReceived`, you can use the new event handler defined by `LineReceiver` called `rawDataReceived`.

---

Switching the protocol requires only a minimum of work. Listing 14-9 shows the result. If you look at the resulting output when running this server, you'll see that the newlines are stripped; in other words, using `print` won't give you double newlines anymore.

**Listing 14-9.** An Improved Logging Server, Using the LineReceiver Protocol

```
from twisted.internet import reactor
from twisted.internet.protocol import Factory
from twisted.protocols.basic import LineReceiver

class SimpleLogger(LineReceiver):

    def connectionMade(self):
        print 'Got connection from', self.transport.client

    def connectionLost(self, reason):
        print self.transport.client, 'disconnected'

    def lineReceived(self, line):
        print line

factory = Factory()
factory.protocol = SimpleLogger

reactor.listenTCP(1234, factory)
reactor.run()
```

As noted earlier, there is a lot more to the Twisted framework than what I've shown you here. If you're interested in learning more, you should check out the online documentation, available at the Twisted web site (<http://twistedmatrix.com>).

## A Quick Summary

This chapter has given you a taste of several approaches to network programming in Python. Which approach you choose will depend on your specific needs and preferences. Once you've chosen, you will, most likely, need to learn more about the specific method. Here are some of the topics this chapter touched upon:

**Sockets and the socket module:** Sockets are information channels that let programs (processes) communicate, possibly across a network. The `socket` module gives you low-level access to both client and server sockets. Server sockets listen at a given address for client connections, while clients simply connect directly.

**urllib and urllib2:** These modules let you read and download data from various servers, given a URL to the data source. The `urllib` module is a simpler implementation, while `urllib2` is very extensible and quite powerful. Both work through straightforward functions such as `urlopen`.

**The SocketServer framework:** This is a network of synchronous server base classes, found in the standard library, which lets you write servers quite easily. There is even support for simple web (HTTP) servers with CGI. If you want to handle several connections simultaneously, you need to use a *forking* or *threading* mix-in class.

**select** and **poll**: These two functions let you consider a set of connections and find out which ones are ready for reading and writing. This means that you can serve several connections piecemeal, in a round-robin fashion. This gives the illusion of handling several connections at the same time, and, although superficially a bit more complicated to code, is a much more scalable and efficient solution than threading or forking.

**Twisted**: This framework, from Twisted Matrix Laboratories, is very rich and complex, with support for most major network protocols. Even though it is large, and some of the idioms used may seem a bit foreign, basic usage is very simple and intuitive. The Twisted framework is also asynchronous, so it's very efficient and scalable. If you have Twisted available, it may very well be the best choice for many custom network applications.

## New Functions in This Chapter

Function	Description
<code>urllib.urlopen(url[, data[, proxies]])</code>	Opens a file-like object from a URL
<code>urllib.urlretrieve(url[, fname[, hook[, data]]])</code>	Downloads a file from a URL
<code>urllib.quote(string[, safe])</code>	Quotes special URL characters
<code>urllib.quote_plus(string[, safe])</code>	The same as quote, but quotes spaces as +
<code>urllib.unquote(string)</code>	The reverse of quote
<code>urllib.unquote_plus(string)</code>	The reverse of quote_plus
<code>urllib.urlencode(query[, doseq])</code>	Encodes mapping for use in CGI queries
<code>select.select(iseq, oseq, eseq[, timeout])</code>	Finds sockets ready for reading/writing
<code>select.poll()</code>	Creates a poll object, for polling sockets
<code>reactor.listenTCP(port, factory)</code>	Twisted function; listens for connections
<code>reactor.run()</code>	Twisted function; main server loop

## What Now?

You thought we were finished with network stuff now, huh? Not a chance. The next chapter deals with a quite specialized and much publicized entity in the world of networking: the Web.



# Chapter 9

## Python Facts

### 9.1 Zip

Combining lists is really easy using zip.

```
names = ['john', 'peter', 'robert']
ages = [24, 26, 35]
cars = ['BMW', 'Audi', 'Ford']

print zip(names, ages, cars)
>>[('john', 24, 'BMW'), ('peter', 26, 'Audi'), ('robert', 35, 'Ford')]
```

### 9.2 List Comprehensions Support Conditions

You can add conditions to your list comprehensions

```
things = [1,2,'3','dog']
numbers = [t for t in things if t.isdigit()]

>>Traceback (most recent call last):
  File "/base/data/home/apps/s~learnpythonjail/2.365841894475711898/main.py", ←
    line 75, in execute_python
      exec(code, {})
  File "<string>", line 2, in <module>
AttributeError: 'int' object has no attribute 'isdigit'
```

### 9.3 Expanding arrays to function arguments

To pass a list of arguments to a function using an array, use the following notation:

```
def add(n1, n2):
    return n1 + n2

numbers = [3, 4]
print add(*numbers)

>>7
```

## 9.4 Ordered and Named Function Arguments

You can accept variable amounts of function parameters by name and by order

```
def f(*args, **kwargs):
    print(args)
    print(kwargs)

f(1, 2, 3, named1="keyword arg1", another="another kwarg")

>>
(1, 2, 3)
{'named1': 'keyword arg1', 'another': 'another kwarg'}
```

## 9.5 Functions as objects

Everything is an object. Functions are true objects, so they can be passed as arguments

```
def talk(text):
    print text

def think(idea):
    print "thinking about %s" % idea

def do_something(what, with_what):
    what(with_what)

do_something(talk, 'hello!')
do_something(think, 'food...')

>>
hello!
thinking about food...
```

## 9.6 Mutability

Identifiers point to objects, so a change to a mutable object (like a list) will change it no matter how it's identified. To make a copy of a list, slice it, or use copy (or deepcopy).

```
original_list = ['a', 'b', 'c']
copy_of_original_list = original_list
copy_of_original_list.pop()
print original_list

original_list = ['a', 'b', 'c']
copy_of_original_list = original_list[:] # added 3 characters in this statement
copy_of_original_list.pop()
print original_list

>>
['a', 'b']
['a', 'b', 'c']
```

## 9.7 Change mutable lists using another name

Changing one list can change another list, which can hide bugs or obfuscate code.

```
congratulatory_message = list("Lucky You!")

temp = congratulatory_message
for x in range(0,8,4):
    temp[x] = chr((8-x)*10)

# Looks like I never changed the congratulatory message...
print ''.join(congratulatory_message)

>>
Puck( You!
```

## 9.8 Factorial Function

There are probably as many different non-native implementations of the factorial function in Python as there are stars in the sky, but this one shines particularly bright. This lambda function uses reduce to iterate through numbers from 2 to n, multiplying them as it goes along. Note that the third argument to reduce is the default value, which is returned if the sequence given by xrange is empty (e.g. if n is zero or one).

```
from operator import mul
factorial = lambda n: reduce(mul, xrange(2, n+1), 1)
print factorial(21)
>>
51090942171709440000
```

## 9.9 String formatting using dictionaries

Python has really powerful string formatting. If you want to use dictionary based (template-like) formatting, then you can:

```
data = {"name" : "Alfred"}
print "Hello %(name)s! How are you?" % data

>>Output:
Hello Alfred! How are you?
```

## 9.10 Never Define One Line Functions

Lambdas are Python's way of telling you to avoid overusing def.

```
from math import sqrt
def quad(a, b, c):
    return -b/a + sqrt(b**2-4*a*c), -b/a - sqrt(b**2-4*a*c)
```

```
roots = lambda a, b, c: (-b/a + sqrt(b**2-4*a*c), -b/a - sqrt(b**2-4*a*c))
#Use with caution.

>>Output
```

## 9.11 Number from Digits

Turn a list of digits to a number.

```
digit_list = [1,2,3,4,5]
print reduce(lambda x,y: 10*x+y, digit_list)
>>Output:
12345
```

## 9.12 Generator Functions

Python supports generator functions, which create iterable sequences on the fly:

```
def lottery_numbers():
    my_numbers = [1,3,2,5,4,6]

    for n in my_numbers:
        if n > 3:
            yield n+10
        else:
            yield n

for n in lottery_numbers():
    print n

>>Output:
1
3
2
15
14
16
```

## 9.13 Parsing REST API

Python is really awesome when it comes to API integration. Check out how easy it is to search Twitter:

```
import urllib, simplejson, pprint
data = simplejson.load(urllib.urlopen("http://search.twitter.com/search.json?q=learnpython&count=5"))
pprint pprint([x["text"] for x in data["results"]])
```

## 9.14 Sets for finding unions and intersections

Using set operations is powerful. Find the elements common to all lists in a list of lists.

```
def get_common_elements(list_of_lists):
    return list(set.intersection(*map(set, list_of_lists)))

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
print get_common_elements([a, b])
print list(set.union( set(a), set(b) ))
```

>>Output:  
[8, 2, 4, 10, 6]  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20]

## 9.15 Locals and strings

Use locals for easy string interpolation.

```
world = "earth"
print "hello %(world)s" % locals()

>>Output:
hello earth
```

## 9.16 Encoding & Decoding

Python supports a wide variety of encoding and decoding methods.

```
print "ABCDEFGH".encode("hex")
print 'VGhpvyBpcyBhIHNlY3JldCBNZXNzYWdl\n'.decode("base64")

>>Output:
4142434445464748
This is a secret Message
```

## 9.17 Set Comprehensions

Python 2.7 supports set literals and comprehensions.

```
print {1, 2, 2, 3, 1}

names = ["Foo", "bar", "foo", "Baz", "CATS", "dogs"]
print {name.upper() for name in names if name[0] in 'ffB'}
```

>>Output:  
set([1, 2, 3])  
set(['BAZ', 'FOO', 'BAR'])

## 9.18 dict() and zip() make for a powerful combination

Create dicts by ziping keys and data.

```
voters = ['friends', 'neighbors', 'countrymen']
counts = [10, 5, 800]
print dict(zip(voters, counts))

votes = ['yes', 66, 'no', 77, 'abstain', 9]
print dict(zip(votes[:-1:2], votes[1::2])))

>>Output:
{'neighbors': 5, 'countrymen': 800, 'friends': 10}
{'yes': 66, 'abstain': 9, 'no': 77}
```

## 9.19 Bools inherit from int

Check help(True) for more details.

```
print True + True == 2
>>Output:
True
```

## 9.20 The Zen of Python

Long time Pythoner Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

```
import this
```

# Chapter 10

## Python Glossary

This is a glossary of terms and topics in Python as covered in Codecademy courses. It does not attempt to completely define and explain terms, but rather provides a general overview that is appropriate for someone new to the language. For a more comprehensive treatment of these topics, we recommend the Mozilla Developer Network Python documentation.

### 10.1 List

A Python data type that holds an ordered collection of values, which can be of any type. This is equivalent to an "array" in many other languages. Python lists are "mutable," implying that they can be changed once created.

```
>> x = [1, 2, 3, 4]
>> y = ['spam', 'eggs']
>> x
[1, 2, 3, 4]
>> y
['spam', 'eggs']

>> y.append('mash')
>> y
['spam', 'eggs', 'mash']

>> y += ['beans']
>> y
['spam', 'eggs', 'mash', 'beans']
```

### 10.2 Slice

A pythonic way of extracting "slices" of a list using a special bracket notation that specifies the start and end of the section of the list you wish to extract. Leaving the beginning value blank indicates you wish to start at the beginning of the list, leaving the ending value blank indicates you wish to go to the end of the list. Using a negative value references the end of the list (so that in a list of 4 elements, -1 means the 4th element). Slicing always yields another list, even when extracting a single value.

```
>> # Specifying a beginning and end:
>> x = [1, 2, 3, 4]
>> x[2:3]
[3]

>> # Specifying start at the beginning and end at the second element
>> x[:2]
[1, 2]

>> # Specifying start at the next to last element and go to the end
>> x[-2:]
[3, 4]

>> # Specifying start at the beginning and go to the next to last element
>> x[:-1]
[1, 2, 3]

>> # Specifying a step argument returns every n-th item
>> y = [1, 2, 3, 4, 5, 6, 7, 8]
>> y[::2]
[1, 3, 5, 7]

>> # Return a reversed version of the list ( or string )
>> x[::-1]
[4, 3, 2, 1]

>> # String reverse
>> my_string = "Aloha"
>> my_string[::-1]
"aholA"
```

## 10.3 Variables

Variables are assigned values using the '=' operator, which is not to be confused with the '==' sign used for testing equality. A variable can hold almost any type of value such as lists, dictionaries, functions.

```
>> x = 12
>> x
12
```

## 10.4 Functions

Python builds functions using the syntax: *def function\_name(variable)* : Functions can be stand-alone or can return values. Functions can also contain other functions.

```
def add_two(a, b):
    c = a + b
    return c

# or without the interim assignment to c
def add_two(a, b):
    return a + b
```

## 10.5 Tuples

A Python data type that holds an ordered collection of values, which can be of any type. Python tuples are "immutable," meaning that they cannot be changed once created.

```
>> x = (1, 2, 3, 4)
>> y = ('spam', 'eggs')

>> my_list = [1,2,3,4]
>> my_tuple = tuple(my_list)
>> my_tuple
(1, 2, 3, 4)
```

## 10.6 List Comprehensions

Convenient ways to generate or extract information from lists. List Comprehensions will take a general form such as: [item for item in List if Condition]

```
>> x_list = [1,2,3,4,5,6,7]
>> even_list = [num for num in x_list if (num % 2 == 0)]
>> even_list
[2,4,6]

>> m_list = ['AB', 'AC', 'DA', 'FG', 'LB']
>> A_list = [duo for duo in m_list if ('A' in duo)]
>> A_list
['AB', 'AC', 'DA']
```

## 10.7 Sets

Sets are collections of unique but unordered items. It is possible to convert certain iterables to a set. "this", "is", "a", "set"

```
>> new_set = {1, 2, 3, 4, 4, 'A', 'B', 'B', 'C'}
>> new_set
{'A', 1, 'C', 3, 4, 2, 'B'}

>> dup_list = [1,1,2,2,2,3,4,55,5,5,6,7,8,8]
>> set_from_list = set(dup_list)
>> set_from_list
{1, 2, 3, 4, 5, 6, 7, 8, 55}
```

## 10.8 Dictionaries

Dictionaries, like sets, contain unique but unordered items. The big difference is the concept of "keys" to retrieve "values"; "keys" can be strings, integers or tuples (or anything else hashable), but the "values" that they map to can be any data type.

```
>> my_dict = {}
>> content_of_value1 = "abcd"
>> content_of_value2 = "wxyz"
>> my_dict.update({ "key_name1":content_of_value1})
>> my_dict.update({ "key_name2":content_of_value2})
>> my_dict
{ 'key_name1':"abcd", 'key_name2': "wxyz" }
>> my_dict.get("key_name2")
"wxyz"
```

## 10.9 Strings

Strings store characters and have many built-in convenience methods that let you modify their content.

```
>> my_string1 = "this is a valid string"
>> my_string2 = 'this is also a valid string'
>> my_string3 = 'this is ' + ' ' + 'also' + ' ' + 'a string'
>> my_string3
"this is also a string"
```

## 10.10 The len() Function

Using `len(some_object)` returns the number of `_toplevel_` items contained in the object being queried.

```
>> my_list = [0,4,5,2,3,4,5]
>> len(my_list)
7
>> my_string = 'abcdef'
>> len(my_string)
6
```

## 10.11 Single Line Comments

Augmenting code with human readable descriptions can help document design decisions.

```
# this is a single line comment.
```

## 10.12 Multi-line Comments

Some comments need to span several lines, use this if you have more than 4 single line comments in a row.

```
'''  
this is  
a multi-line  
comment, i am handy for commenting out whole  
chunks of code very fast  
'''
```

## 10.13 Print

A function to display the output of a program. Using the parenthesized version is arguably more consistent.

```
>> # this will work in all modern versions of Python  
>> print("some text here")  
"some text here"  
  
>> # but this only works in Python versions lower than 3.x  
>> print "some text here too"  
"some text here too"
```

## 10.14 The range() Function

The range() function returns a list of integers, the sequence of which is defined by the arguments passed to it.

argument variations:

- range(terminal)
- range(start, terminal)
- range(start, terminal, step\_size)

```
>> [i for i in range(4)]  
[0, 1, 2, 3]  
  
>> [i for i in range(2, 8)]  
[2, 3, 4, 5, 6, 7]  
  
>> [i for i in range(2, 13, 3)]  
[2, 5, 8, 11]
```

## 10.15 For Loops

Python provides a clean iteration syntax. Note the colon and indentation.

```
>> for i in range(0, 3):  
>>     print(i*2)  
0
```

```

2
4

>> m_list = [ "Sir" , "Lancelot" , "Coconuts" ]
>> for item in m_list:
>>     print(item)
Sir
Lancelot
Coconuts

>> w_string = "Swift"
>> for letter in w_string:
>>     print(letter)
S
w
i
f
t

```

## 10.16 While Loops

A While loop permits code to execute repeatedly until a certain condition is met. This is useful if the number of iterations required to complete a task is unknown prior to flow entering the loop.

```

>> looping_needed = True
>>
>> while looping_needed:
>>     # some operation on data
>>     if condition:
>>         looping_needed = False

```

## 10.17 The str() Function

Using the str() function allows you to represent the content of a variable as a string, provided that the data type of the variable provides a neat way to do so. str() does not change the variable in place, it returns a 'stringified' version of it.

```

>> # such features can be useful for concatenating strings
>> my_var = 123
>> my_var
123

>> str(my_var)
'123'

>> my_booking = "DB Airlines Flight " + str(my_var)
>> my_booking
'DB Airlines Flight 123'

```

## 10.18 »>

The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

## 10.19 ...

The default Python prompt of the interactive shell when entering code for an indented code block or within a pair of matching left and right delimiters (parentheses, square brackets or curly braces).

## 10.20 2to3

A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as lib2to3; a standalone entry point is provided as Tools/scripts/2to3. See 2to3 - Automated Python 2 to 3 code translation.

## 10.21 abstract base class

Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like hasattr() would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by isinstance() and issubclass(); see the abc module documentation. Python comes with many built-in ABCs for data structures (in the collections module), numbers (in the numbers module), and streams (in the io module). You can create your own ABCs with the abc module.

## 10.22 argument

A value passed to a function (or method) when calling the function. There are two types of arguments:

- keyword argument: an argument preceded by an identifier (e.g. name=) in a function call or passed as a value in a dictionary preceded by \*\*. For example, 3 and 5 are both keyword arguments in the following calls to complex():

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- positional argument: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by \*. For example, 3 and 5 are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the Calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

## 10.23 attribute

A value associated with an object which is referenced by name using dotted expressions. For example, if an object `o` has an attribute `a` it would be referenced as `o.a`.

## 10.24 BDFL

Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

## 10.25 bytecode

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a virtual machine that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the `dis` module.

## 10.26 class

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

## 10.27 classic class

Any class which does not inherit from `object`. See new-style class. Classic classes have been removed in Python 3.

## 10.28 coercion

The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before

they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

## 10.29 complex number

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

## 10.30 context manager

An object which controls the environment seen in a `with` statement. See PEP 343.

## 10.31 CPython

The canonical implementation of the Python programming language, as distributed on [python.org](http://python.org). The term “CPython” is used when necessary to distinguish this implementation from others such as Jython or IronPython.

## 10.32 decorator

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

## 10.33 descriptor

Any new-style object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see [Implementing Descriptors](#).

## 10.34 dictionary

An associative array, where arbitrary keys are mapped to values.

## 10.35 docstring

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

## 10.36 duck-typing

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs `hasattr()` tests or EAFP programming.

## 10.37 EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the LBYL style common to many other languages such as C.

## 10.38 expression

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access,

operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as print or if. Assignments are also statements, not expressions.

## 10.39 extension module

A module written in C or C++, using Python's C API to interact with the core and with user code.

## 10.40 file object

An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called file-like objects or streams.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

## 10.41 file-like object

A synonym for file object.

## 10.42 finder

An object that tries to find the loader for a module. It must implement a method named `find_module()`. See PEP 302 for details.

## 10.43 floor division

Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to 2 in contrast to the 2.75 returned by float true division. Note that `(-11) // 4` is -3 because that is -2.75 rounded downward. See PEP 238.

## 10.44 function

A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also parameter, method, and the Function definitions section.

## 10.45 \_\_future\_\_

A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression  $11/4$  currently evaluates to 2. If the module in which it is executed had enabled true division by executing:

```
from __future__ import division
```

the expression  $11/4$  would evaluate to 2.75. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>>
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

## 10.46 garbage collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

## 10.47 generator

A function which returns an iterator. It looks like a normal function except that it contains yield statements for producing a series a values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

## 10.48 generator expression

An expression that returns an iterator. It looks like a normal expression followed by a for expression defining a loop variable, range, and an optional if expression. The combined expression generates values for an enclosing function:

```
>>>
>>> sum(i*i for i in range(10))          # sum of squares 0, 1, 4, ... 81
285
```

## 10.49 GIL

See global interpreter lock.

## 10.50 global interpreter lock

The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

## 10.51 hashable

An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python’s immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is their `id()`.

## 10.52 IDLE

An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

## 10.53 immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value

has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

## 10.54 integer division

Mathematical division discarding any remainder. For example, the expression  $11/4$  currently evaluates to 2 in contrast to the 2.75 returned by float division. Also called floor division. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a float), the result will be coerced (see coercion) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also [\\_\\_future\\_\\_](#).

## 10.55 importer

An object that both finds and loads a module; both a finder and loader object.

## 10.56 interactive

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

## 10.57 interpreted

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also [interactive](#).

## 10.58 iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the

object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also `iterator`, `sequence`, and `generator`.

## 10.59 iterator

An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a list) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in [Iterator Types](#).

## 10.60 key function

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example, the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a lambda expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the [Sorting HOW TO](#) for examples of how to create and use key functions.

## 10.61 keyword argument

See `argument`.

## 10.62 lambda

An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is `lambda`

[arguments]: expression

## 10.63 LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

## 10.64 list

A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked list since access to elements are  $O(1)$ .

## 10.65 list comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x"%x for x in range(256) if x%2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The if clause is optional. If omitted, all elements in range(256) are processed.

## 10.66 loader

An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a finder. See PEP 302 for details.

## 10.67 mapping

A container object that supports arbitrary key lookups and implements the methods specified in the Mapping or MutableMapping abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

## 10.68 metaclass

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses

can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in Customizing class creation.

## 10.69 method

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`). See function and nested scope.

## 10.70 method resolution order

Method Resolution Order is the order in which base classes are searched for a member during lookup. See The Python 2.3 Method Resolution Order.

## 10.71 MRO

See method resolution order.

## 10.72 mutable

Mutable objects can change their value but keep their `id()`. See also immutable.

## 10.73 named tuple

Any tuplelike class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuplelike object where the year is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

## 10.74 namespace

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability

by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

## 10.75 nested scope

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

## 10.76 new-style class

Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

More information can be found in [New-style and classic classes](#).

## 10.77 object

Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any new-style class.

## 10.78 parameter

A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are four types of parameters:

- positional-or-keyword: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example `foo` and `bar` in the following:

```
def func(foo, bar=None): ...
```

- positional-only: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. `abs()`).
- var-positional: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example `args` in the following:

```
def func(*args, **kwargs): ...
```

- var-keyword: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example `kwargs` in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the argument glossary entry, the FAQ question on the difference between arguments and parameters, and the Function definitions section.

## 10.79 positional argument

See argument.

## 10.80 Python 3000

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated “Py3k”.

## 10.81 Pythonic

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don’t have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print food[i]
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print piece
reference count
```

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

## 10.82 `__slots__`

A declaration inside a new-style class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

## 10.83 `sequence`

An iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

## 10.84 `slice`

An object usually containing a portion of a sequence. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses slice objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

## 10.85 `special method`

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in `Special method names`.

## 10.86 `statement`

A statement is part of a suite (a “block” of code). A statement is either an expression or a one of several constructs with a keyword, such as `if`, `while` or `for`.

## 10.87 `struct sequence`

A tuple with named elements. Struct sequences expose an interface similiar to `namedtuple` in that elements can either be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

## 10.88 type

The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

## 10.89 view

The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They are lazy sequences that will see changes in the underlying dictionary. To force the dictionary view to become a full list use `list(dictview)`. See Dictionary view objects.

## 10.90 virtual machine

A computer defined entirely in software. Python's virtual machine executes the bytecode emitted by the bytecode compiler.

## 10.91 Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing "import this" at the interactive prompt.



# **Chapter 11**

## **Project**



# Project 5: A Virtual Tea Party

In this project, you do some serious network programming. You'll write a chat server—a program that lets several people connect via the Internet and chat with each other in real time. There are many ways to create such a beast in Python. A simple and natural approach might be to use the Twisted framework (discussed in Chapter 14), for example, with the `LineReceiver` class taking center stage. In this chapter, I stick to the standard libraries, basing the program on the modules `asyncore` and `asynchat`. If you like, you could try out some of the alternative methods (such as forking or threading) discussed in Chapter 14.

## What's the Problem?

Online chatting is quite common. Many chat services of various kinds (IRC, instant messaging services, and so forth) are available all over the Internet. Some of these are even full-fledged text-based virtual worlds (see <http://www.mudconnect.com> for a long list). If you want to set up a chat server, you can just download and install one of the many free server programs. However, writing a chat server yourself is useful for two reasons:

- You learn about network programming.
- You can customize it as much as you want.

The second point suggests that you can start with a simple chat server and develop it into basically any kind of server (including a virtual world), with all the power of Python at your fingertips. Pretty awesome, isn't it?

For now, the chat server project has the following requirements:

- The server should be able to receive multiple connections from different users.
- It should let the users act *in parallel*.
- It should be able to interpret commands such as `say` or `logout`.
- The server should be easily extensible.

The two things that will require special tools are the network connections and the asynchronous nature of the program.

## Useful Tools

The only new tools you need in this project are the `asyncore` module from the standard library and its relative `asynchat`. I'll describe the basics of how these work. You can find more details about them in the Python Library Reference (<http://python.org/doc/lib/module-asyncore.html> and <http://python.org/doc/lib/module-asynchat.html>).

As discussed in Chapter 14, the basic component in a network program is the `socket`. Sockets can be created directly by importing the `socket` module and using the functions there. So what do you need `asyncore` for?

The `asyncore` framework enables you to juggle several users who are connected simultaneously. Imagine a scenario in which you have no special tools for handling this. When you start up the server, it waits for users to connect. When one user is connected, it starts reading data from that user and supplying results through a socket. But what happens if another user is already connected? The second user to connect must wait until the first one has finished. In some cases, that will work just fine, but when you're writing a chat server, the whole point is that more than one user can be connected—how else could users chat with one another?

The `asyncore` framework is based on an underlying mechanism (the `select` function from the `select` module, as discussed in Chapter 14) that allows the server to serve all the connected users in a piecemeal fashion. Instead of reading *all* the available data from one user before going on to the next, only *some* data is read. Also, the server reads only from the sockets where there *is* data to be read. This is done again and again, in a loop. Writing is handled similarly. You could implement this yourself using just the modules `socket` and `select`, but `asyncore` and `asynchat` provide a very useful framework that takes care of the details for you. (For alternative ways of implementing parallel user connections, see the section “Multiple Connections” in Chapter 14.)

## Preparations

The first thing you need is a computer that's connected to a network (such as the Internet); otherwise, others won't be able to connect to your chat server. (It is possible to connect to the chat server from your own machine, but that's not much fun in the long run, is it?) To be able to connect, the user must know the address of your machine (a machine name such as `foo.bar.baz.com` or an IP address). In addition, the user must know the *port number* used by your server. You can set this in your program; in the code in this chapter, I use the (rather arbitrary) port number 5005.

---

**Note** As mentioned in Chapter 14, certain port numbers are restricted and require administrator privileges. In general, numbers greater than 1023 are okay.

---

To test your server, you need a *client*—the program on the user side of the interaction. A simple program for this sort of thing is `telnet` (which basically lets you connect to any socket server). In UNIX, you probably have this program available on the command line:

```
$ telnet some.host.name 5005
```

The preceding command connects to the machine `some.host.name` on port 5005. To connect to the same machine on which you’re running the `telnet` command, simply use the machine name `localhost`. (You might want to supply an escape character through the `-e` switch to make sure you can quit `telnet` easily. See the `man` page for more details.)

In Windows, you can use either the standard `telnet` command (in a command-prompt window) or a terminal emulator with `telnet` functionality, such as PuTTY (software and more information available at <http://www.chiark.greenend.org.uk/~sgtatham/putty>). However, if you are installing new software, you might as well get a client program tailored to chatting. MUD (or MUSH or MOO or some other related acronym) clients<sup>1</sup> are quite suitable for this sort of thing. My client of choice is TinyFugue (software and more information available at <http://tinyfugue.sf.net>). It is mainly designed for use in UNIX. (Several clients are available for Windows as well; just do a web search for “mud client” or something similar.)

## First Implementation

Let’s break things down a bit. We need to create two main classes: one representing the chat server and one representing each of the chat sessions (the connected users).

### The ChatServer Class

To create the basic `ChatServer`, you subclass the `dispatcher` class from `asyncore`. The `dispatcher` is basically just a socket object, but with some extra event-handling features, which you’ll be using in a minute.

See Listing 24-1 for a basic chat server program (that does very little).

#### **Listing 24-1. A Minimal Server Program**

```
from asyncore import dispatcher
import asyncore

class ChatServer(dispatcher): pass

s = ChatServer()
asyncore.loop()
```

If you run this program, nothing happens. To make the server do anything interesting, you should call its `create_socket` method to create a socket, and its `bind` and `listen` methods to bind the socket to a specific port number and to tell it to listen for incoming connections. (That is what servers do, after all.) In addition, you’ll override the `handle_accept` event-handling method to actually do something when the server accepts a client connection. The resulting program is shown in Listing 24-2.

---

1. MUD stands for Multi-User Dungeon/Domain/Dimension. MUSH stands for Multi-User Shared Hallucination. MOO means MUD, object-oriented. See, for example, Wikipedia (<http://en.wikipedia.org/wiki/MUD>) for more information.

**Listing 24-2.** *A Server That Accepts Connections*

```
from asyncore import dispatcher
import socket, asyncore

class ChatServer(dispatcher):

    def handle_accept(self):
        conn, addr = self.accept()
        print 'Connection attempt from', addr[0]

s = ChatServer()
s.create_socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 5005))
s.listen(5)
asyncore.loop()
```

The `handle_accept` method calls `self.accept`, which lets the client connect. This returns a connection (a socket that is specific for this client) and an address (information about which machine is connecting). Instead of doing anything useful with this connection, the `handle_accept` method simply prints that a connection attempt was made. `addr[0]` is the IP address of the client.

The server initialization calls `create_socket` with two arguments that specify the type of socket you want. You could use different types, but those shown here are what you usually want. The call to the `bind` method simply binds the server to a specific address (host name and port). The host name is empty (an empty string, essentially meaning localhost, or, more technically, “all interfaces on this machine”) and the port number is 5005. The call to `listen` tells the server to listen for connections; it also specifies a backlog of five connections. The final call to `asyncore.loop` starts the server’s listening loop as before.

This server actually works. Try to run it and then connect to it with your client. The client should immediately be disconnected, and the server should print out the following:

```
Connection attempt from 127.0.0.1
```

The IP address will be different if you don’t connect from the same machine as your server.

To stop the server, simply use a keyboard interrupt: `Ctrl+C` in UNIX or `Ctrl+Break` in Windows.

Shutting down the server with a keyboard interrupt results in a stack trace. To avoid that, you can wrap the loop in a `try/except` statement. With some other cleanups, the basic server ends up as shown in Listing 24-3.

**Listing 24-3.** *The Basic Server with Some Cleanups*

```
from asyncore import dispatcher
import socket, asyncore

PORT = 5005
```

```
class ChatServer(dispatcher):  
  
    def __init__(self, port):  
        dispatcher.__init__(self)  
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)  
        self.set_reuse_addr()  
        self.bind(('', port))  
        self.listen(5)  
  
    def handle_accept(self):  
        conn, addr = self.accept()  
        print 'Connection attempt from', addr[0]  
  
if __name__ == '__main__':  
    s = ChatServer(PORT)  
    try: asyncore.loop()  
    except KeyboardInterrupt: pass
```

The added call to `set_reuse_addr` lets you reuse the same address (specifically, the port number) even if the server isn't shut down properly. (Without this call, you may need to wait for a while before the server can be started again, or change the port number each time the server crashes, because your program may not be able to properly notify your operating system that it's finished with the port.)

## The ChatSession Class

The basic `ChatServer` isn't very useful. Instead of ignoring the connection attempts, a new `dispatcher` object should be created for each connection. However, these objects will behave differently from the one used as the main server. They won't be listening on a port for incoming connections; they already *are* connected to a client. Their main task is collecting data (text) coming from the client and responding to it. You could implement this functionality yourself by subclassing `dispatcher` and overriding various methods, but, luckily, there is a module that already does most of the work: `asynchat`.

Despite the name, `asynchat` isn't specifically designed for the type of streaming (continuous) chat application that we're working on. (The `chat` in the name refers to "chat-style" or command-response protocols.) The good thing about the `async_chat` class (found in the `asynchat` module) is that it hides the most basic socket reading and writing operations, which can be a bit difficult to get right. All that's needed to make it work is to override two methods: `collect_incoming_data` and `found_terminator`. The former is called each time a bit of text has been read from the socket, and the latter is called when a *terminator* is read. The terminator (in this case) is just a line break. (You'll need to tell the `async_chat` object about that by calling `set_terminator` as part of the initialization.)

An updated program, now with a `ChatSession` class, is shown in Listing 24-4.

**Listing 24-4.** Server Program with ChatSession Class

```
from asyncore import dispatcher
from asynchat import async_chat
import socket, asyncore

PORT = 5005

class ChatSession(async_chat):

    def __init__(self, sock):
        async_chat.__init__(self, sock)
        self.set_terminator("\r\n")
        self.data = []

    def collect_incoming_data(self, data):
        self.data.append(data)

    def found_terminator(self):
        line = ''.join(self.data)
        self.data = []
        # Do something with the line...
        print line

class ChatServer(dispatcher):

    def __init__(self, port):
        dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(('', port))
        self.listen(5)
        self.sessions = []

    def handle_accept(self):
        conn, addr = self.accept()
        self.sessions.append(ChatSession(conn))

if __name__ == '__main__':
    s = ChatServer(PORT)
    try: asyncore.loop()
    except KeyboardInterrupt: print
```

Several things are worth noting in this new version:

- The `set_terminator` method is used to set the line terminator to "`\r\n`", which is the commonly used line terminator in network protocols.
- The `ChatSession` object keeps the data it has read so far as a list of strings called `data`. When more data is read, `collect_incoming_data` is called automatically, and it simply appends the data to the list. Using a list of strings and later joining them (with the `join` string method) is a common idiom (and historically more efficient than incrementally adding strings). Feel free to use `+=` with strings instead.
- The `found_terminator` method is called when a terminator is found. The current implementation creates a line by joining the current data items, and resets `self.data` to an empty list. However, because you don't have anything useful to do with the line yet, it is simply printed.
- The `ChatServer` keeps a list of sessions.
- The `handle_accept` method of the `ChatServer` now creates a new `ChatSession` object and appends it to the list of sessions.

Try running the server and connecting with two (or more) clients simultaneously. Every line you type in a client should be printed in the terminal where your server is running. That means the server is now capable of handling several simultaneous connections. Now all that's missing is the capability for the clients to see what the others are saying!

## Putting It Together

Before the prototype can be considered a fully functional (albeit simple) chat server, one main piece of functionality is lacking: what the users say (each line they type) should be broadcast to the others. That functionality can be implemented by a simple `for` loop in the server, which loops over the list of sessions and writes the line to each of them. To write data to an `async_chat` object, you use the `push` method.

This broadcasting behavior also adds another problem: you must make sure that connections are removed from the list when the clients disconnect. You can do that by overriding the event-handling method `handle_close`. The final version of the first prototype can be seen in Listing 24-5.

**Listing 24-5.** A Simple Chat Server (`simple_chat.py`)

```
from asyncore import dispatcher
from asynchat import async_chat
import socket, asyncore

PORT = 5005
NAME = 'TestChat'
```

```
class ChatSession(async_chat):
    """
    A class that takes care of a connection between the server
    and a single user.
    """
    def __init__(self, server, sock):
        # Standard setup tasks:
        async_chat.__init__(self, sock)
        self.server = server
        self.set_terminator("\r\n")
        self.data = []
        # Greet the user:
        self.push('Welcome to %s\r\n' % self.server.name)

    def collect_incoming_data(self, data):
        self.data.append(data)

    def found_terminator(self):
        """
        If a terminator is found, that means that a full
        line has been read. Broadcast it to everyone.
        """
        line = ''.join(self.data)
        self.data = []
        self.server.broadcast(line)

    def handle_close(self):
        async_chat.handle_close(self)
        self.server.disconnect(self)

class ChatServer(dispatcher):
    """
    A class that receives connections and spawns individual
    sessions. It also handles broadcasts to these sessions.
    """
    def __init__(self, port, name):
        # Standard setup tasks
        dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(('', port))
        self.listen(5)
        self.name = name
        self.sessions = []
```

```

def disconnect(self, session):
    self.sessions.remove(session)

def broadcast(self, line):
    for session in self.sessions:
        session.push(line + '\r\n')

def handle_accept(self):
    conn, addr = self.accept()
    self.sessions.append(ChatSession(self, conn))

if __name__ == '__main__':
    s = ChatServer(PORT, NAME)
    try: asyncore.loop()
    except KeyboardInterrupt: print

```

## Second Implementation

The first prototype may be a fully functioning chat server, but its functionality is quite limited. The most obvious limitation is that you can't discern who is saying what. Also, it does not interpret commands (such as say or logout), which the original specification requires. So, you need to add support for identity (one unique name per user) and command interpretation, and you must make the behavior of each session depend on the state it's in (just connected, logged in, and so on)—all of this in a manner that lends itself easily to extension.

### Basic Command Interpretation

I'll show you how to model the command interpretation on the `Cmd` class of the `cmd` module in the standard library. (Unfortunately, you can't use this class directly because it can be used only with `sys.stdin` and `sys.stdout`, and you're working with several streams.) What you need is a function or method that can handle a single line of text (as typed by the user). It should split off the first word (the command) and call an appropriate method based on it. For example, this line:

`say Hello, world!`

might result in the following call:

`do_say('Hello, world!')`

possibly with the session itself as an added argument (so `do_say` would know who did the talking).

Here is a simple implementation, with an added method to express that a command is unknown:

```

class CommandHandler:
    """
    Simple command handler similar to cmd.Cmd from the standard
    library.
    """

```

```

def unknown(self, session, cmd):
    session.push('Unknown command: %s\r\n' % cmd)

def handle(self, session, line):
    if not line.strip(): return
    parts = line.split(' ', 1)
    cmd = parts[0]
    try: line = parts[1].strip()
    except IndexError: line = ''
    meth = getattr(self, 'do_'+cmd, None)
    try:
        meth(session, line)
    except TypeError:
        self.unknown(session, cmd)

```

The use of `getattr` in this class is similar to that in the markup project in Chapter 20.

With the basic command handling out of the way, you need to define some actual commands. And which commands are available (and what they do) should depend on the current state of the session. How do you represent that state?

## Rooms

Each state can be represented by a custom command handler. This is easily combined with the standard notion of chat rooms (or locations in a MUD). Each room is a `CommandHandler` with its own specialized commands. In addition, it should keep track of which users (sessions) are currently inside it. Here is a generic superclass for all your rooms:

```

class EndSession(Exception): pass

class Room(CommandHandler):
    """
    A generic environment which may contain one or more users
    (sessions). It takes care of basic command handling and
    broadcasting.
    """

    def __init__(self, server):
        self.server = server
        self.sessions = []

    def add(self, session):
        self.sessions.append(session)

```

```
def remove(self, session):
    self.sessions.remove(session)

def broadcast(self, line):
    for session in self.sessions:
        session.push(line)

def do_logout(self, session, line):
    raise EndSession
```

In addition to the basic add and remove methods, a broadcast method simply calls push on all of the users (sessions) in the room. There is also a single command defined—logout (in the form of the do\_logout method). It raises an exception (EndSession), which is dealt with at a higher level of the processing (in found\_terminator).

## Login and Logout Rooms

In addition to representing normal chat rooms (this project includes only one such chat room), the Room subclasses can represent other states, which was indeed the intention. For example, when a user connects to the server, he is put in a dedicated LoginRoom (with no other users in it). The LoginRoom prints a welcome message when the user enters (in the add method). It also overrides the unknown method to tell the user to log in; the only command it responds to is the login command, which checks whether the name is acceptable (not an empty string, and not already used by another user).

The LogoutRoom is much simpler. Its only job is to delete the user's name from the server (which has a dictionary called users where the sessions are stored). If the name isn't there (because the user never logged in), the resulting KeyError is ignored.

For the source code of these two classes, see Listing 24-6 later in this chapter.

---

**Note** Even though the server's users dictionary keeps references to all the sessions, no session is ever retrieved from it. The users dictionary is used only to keep track of which names are in use. However, instead of using some arbitrary value (such as True), I decided to let each user name refer to the corresponding session. Even though there is no immediate use for it, it may be useful in some later version of the program (for example, if one user wants to send a message privately to another). An alternative would have been to simply keep a set or list of sessions.

---

## The Main Chat Room

The main chat room also overrides the add and remove methods. In add, it broadcasts a message about the user who is entering, and it adds the user's name to the users dictionary in the server. The remove method broadcasts a message about the user who is leaving.

In addition to these methods, the `ChatRoom` class implements three commands:

- The `say` command (implemented by `do_say`) broadcasts a single line, prefixed with the name of the user who spoke.
- The `look` command (implemented by `do_look`) tells the user which users are currently in the room.
- The `who` command (implemented by `do_who`) tells the user which users are currently logged in. In this simple server, `look` and `who` are equivalent, but if you extend it to contain more than one room, their functionality will differ.

For the source code, see Listing 24-6 later in this chapter.

## The New Server

I've now described most of the functionality. The main additions to `ChatSession` and `ChatServer` are as follows:

- `ChatSession` has a method called `enter`, which is used to enter a new room.
- The `ChatSession` constructor uses `LoginRoom`.
- The `handle_close` method uses `LogoutRoom`.
- The `ChatServer` constructor adds the dictionary `users` and the `ChatRoom` called `main_room` to its attributes.

Notice also how `handle_accept` no longer adds the new `ChatSession` to a list of sessions because the sessions are now managed by the rooms.

---

**Note** In general, if you simply instantiate an object, like the `ChatSession` in `handle_accept`, without binding a name to it or adding it to a container, it will be lost, and may be garbage-collected (which means that it will disappear completely). Because all dispatchers are handled (referenced) by `asyncore` (and `async_chat` is a subclass of `dispatcher`), this is not a problem here.

---

The final version of the chat server is shown in Listing 24-6. For your convenience, I've listed the available commands in Table 24-1.

**Listing 24-6.** A Slightly More Complicated Chat Server (`chatserver.py`)

```
from asyncore import dispatcher
from asynchat import async_chat
import socket, asyncore

PORT = 5005
NAME = 'TestChat'
```

```
class EndSession(Exception): pass

class CommandHandler:
    """
    Simple command handler similar to cmd.Cmd from the standard
    library.
    """

    def unknown(self, session, cmd):
        'Respond to an unknown command'
        session.push('Unknown command: %s\r\n' % cmd)

    def handle(self, session, line):
        'Handle a received line from a given session'
        if not line.strip(): return
        # Split off the command:
        parts = line.split(' ', 1)
        cmd = parts[0]
        try: line = parts[1].strip()
        except IndexError: line = ''
        # Try to find a handler:
        meth = getattr(self, 'do_'+cmd, None)
        try:
            # Assume it's callable:
            meth(session, line)
        except TypeError:
            # If it isn't, respond to the unknown command:
            self.unknown(session, cmd)

class Room(CommandHandler):
    """
    A generic environment that may contain one or more users
    (sessions). It takes care of basic command handling and
    broadcasting.
    """

    def __init__(self, server):
        self.server = server
        self.sessions = []

    def add(self, session):
        'A session (user) has entered the room'
        self.sessions.append(session)
```

```

def remove(self, session):
    'A session (user) has left the room'
    self.sessions.remove(session)

def broadcast(self, line):
    'Send a line to all sessions in the room'
    for session in self.sessions:
        session.push(line)

def do_logout(self, session, line):
    'Respond to the logout command'
    raise EndSession

class LoginRoom(Room):
    """
    A room meant for a single person who has just connected.
    """

    def add(self, session):
        Room.add(self, session)
        # When a user enters, greet him/her:
        self.broadcast('Welcome to %s\r\n' % self.server.name)

    def unknown(self, session, cmd):
        # All unknown commands (anything except login or logout)
        # results in a prodding:
        session.push('Please log in\nUse "login <nick>"\r\n')

    def do_login(self, session, line):
        name = line.strip()
        # Make sure the user has entered a name:
        if not name:
            session.push('Please enter a name\r\n')
        # Make sure that the name isn't in use:
        elif name in self.server.users:
            session.push('The name "%s" is taken.\r\n' % name)
            session.push('Please try again.\r\n')
        else:
            # The name is OK, so it is stored in the session, and
            # the user is moved into the main room.
            session.name = name
            session.enter(self.server.main_room)

```

```
class ChatRoom(Room):
    """
    A room meant for multiple users who can chat with the others in
    the room.
    """

    def add(self, session):
        # Notify everyone that a new user has entered:
        self.broadcast(session.name + ' has entered the room.\r\n')
        self.server.users[session.name] = session
        Room.add(self, session)

    def remove(self, session):
        Room.remove(self, session)
        # Notify everyone that a user has left:
        self.broadcast(session.name + ' has left the room.\r\n')

    def do_say(self, session, line):
        self.broadcast(session.name+': '+line+'\r\n')

    def do_look(self, session, line):
        'Handles the look command, used to see who is in a room'
        session.push('The following are in this room:\r\n')
        for other in self.sessions:
            session.push(other.name + '\r\n')

    def do_who(self, session, line):
        'Handles the who command, used to see who is logged in'
        session.push('The following are logged in:\r\n')
        for name in self.server.users:
            session.push(name + '\r\n')

class LogoutRoom(Room):
    """
    A simple room for a single user. Its sole purpose is to remove
    the user's name from the server.
    """

    def add(self, session):
        # When a session (user) enters the LogoutRoom it is deleted
        try: del self.server.users[session.name]
        except KeyError: pass
```

```
class ChatSession(async_chat):
    """
    A single session, which takes care of the communication with a
    single user.
    """

    def __init__(self, server, sock):
        async_chat.__init__(self, sock)
        self.server = server
        self.set_terminator("\r\n")
        self.data = []
        self.name = None
        # All sessions begin in a separate LoginRoom:
        self.enter(LoginRoom(server))

    def enter(self, room):
        # Remove self from current room and add self to
        # next room...
        try: cur = self.room
        except AttributeError: pass
        else: cur.remove(self)
        self.room = room
        room.add(self)

    def collect_incoming_data(self, data):
        self.data.append(data)

    def found_terminator(self):
        line = ''.join(self.data)
        self.data = []
        try: self.room.handle(self, line)
        except EndSession:
            self.handle_close()

    def handle_close(self):
        async_chat.handle_close(self)
        self.enter(LogoutRoom(self.server))

class ChatServer(dispatcher):
    """
    A chat server with a single room.
    """
```

```

def __init__(self, port, name):
    dispatcher.__init__(self)
    self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
    self.set_reuse_addr()
    self.bind(('', port))
    self.listen(5)
    self.name = name
    self.users = {}
    self.main_room = ChatRoom(self)

def handle_accept(self):
    conn, addr = self.accept()
    ChatSession(self, conn)

if __name__ == '__main__':
    s = ChatServer(PORT, NAME)
    try: asyncio.loop()
    except KeyboardInterrupt: print

```

**Table 24-1.** Commands Available in the Chat Server

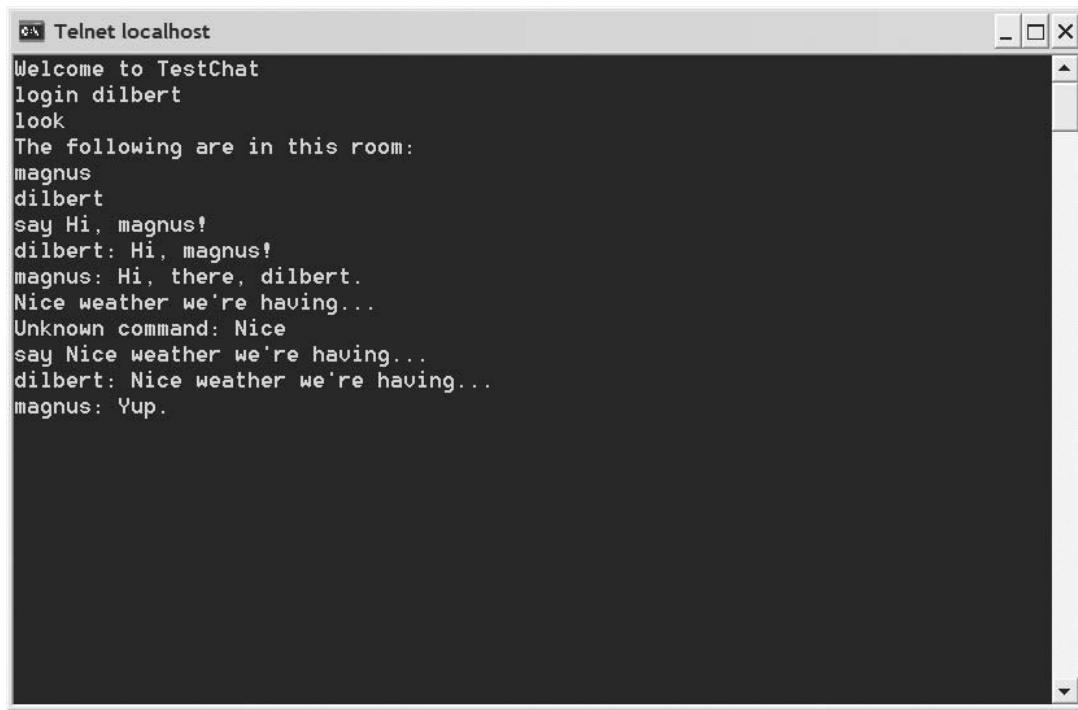
Command	Available In	Description
login name	Login room	Used to log into the server
logout	All rooms	Used to log out of the server
say statement	Chat room(s)	Used to say something
look	Chat room(s)	Used to find out who is in the same room
who	Chat room(s)	Used to find out who is logged on to the server

An example of a chat session is shown in Figure 24-1. The server in that example was started with the this command:

python chatserver.py

and the user dilbert connected to the server using this command:

telnet localhost 5005



```
telnet localhost
Welcome to TestChat
login dilbert
look
The following are in this room:
magnus
dilbert
say Hi, magnus!
dilbert: Hi, magnus!
magnus: Hi, there, dilbert.
Nice weather we're having...
Unknown command: Nice
say Nice weather we're having...
dilbert: Nice weather we're having...
magnus: Yup.
```

Figure 24-1. A sample chat session

## Further Exploration

You can do a lot to extend and enhance the basic server presented in this chapter:

- You could make a version with multiple chat rooms, and you could extend the command set to make it behave in any way you want.
- You might want to make the program recognize only certain commands (such as `login` or `logout`) and treat all other text entered as general chatting, thereby avoiding the need for a `say` command.
- You could prefix all commands with a special character (for example, a slash, giving commands like `/login` and `/logout`) and treat everything that doesn't start with the specified character as general chatting.
- You might want to create your own GUI client, but that's a bit trickier than it might seem. The GUI toolkit has one event loop, and the communication with the server may require another. To make them cooperate, you may need to use threading. (For an example of how this can be done in simple cases where the various threads don't directly access each other's data, see Chapter 28.)

## What Now?

Now you have your very own chat server. In the next project, you tackle a different type of network programming: CGI, the mechanism underlying most web applications (as discussed in Chapter 15). The specific application of this technology in the next project is *remote editing*, which enables several users to collaborate on developing the same document. You may even use it to edit your own web pages remotely.

# Chapter 12

## List Methods

# Some Python list methods

In the “Python: Introduction for Programmers” course we describe just a few methods of lists. This more complete document is for reference and interest; you do not need to memorise these for the course.

**These methods return a value and do not change the list.**

<code>count(value)</code>	How many times does <i>value</i> appear in the list? <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.count(2)</code> 3 <code>&gt;&gt;&gt; numbers</code> [1, 2, 3, 1, 2, 3]
<code>index(value)</code>	Where is the first place <i>value</i> appears in the list? <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.index(2)</code> 1 <code>&gt;&gt;&gt; numbers[1]</code> 2
<code>index(value, start)</code>	Where is the first place <i>value</i> appears in the list at or after <i>start</i> ? <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.index(2,1)</code> 1 <code>&gt;&gt;&gt; numbers.index(2,2)</code> 4 <code>&gt;&gt;&gt; numbers[4]</code> 2

**These methods change the list and do not return any value.**

<code>append(value)</code>	Stick a single value on the end of the list. <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.append(4)</code> <code>&gt;&gt;&gt; numbers</code> [1, 2, 3, 1, 2, 3, 4]
<code>extend(list)</code>	Stick several values on the end of the list. <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.extend([5,6,7])</code> <code>&gt;&gt;&gt; numbers</code> [1, 2, 3, 1, 2, 3, 4, 5, 6, 7]
<code>remove(value)</code>	Remove the first instance of a value from the list. <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.remove(2)</code> <code>&gt;&gt;&gt; numbers</code> [1, 3, 1, 2, 3]
<code>insert(index, value)</code>	Insert <i>value</i> so that it gets index <i>index</i> and move everything up one to make room. <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.insert(3, 5)</code> <code>&gt;&gt;&gt; numbers</code> [1, 2, 3, 5, 1, 2, 3] <code>&gt;&gt;&gt; numbers.insert(0, 6)</code> <code>&gt;&gt;&gt; numbers</code> [6, 1, 2, 3, 5, 1, 2, 3]
<code>reverse()</code>	Reverse the order of the list's items. <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.reverse()</code> <code>&gt;&gt;&gt; numbers</code> [3, 2, 1, 3, 2, 1]
<code>sort()</code>	Sort the items in the list. <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.sort()</code> <code>&gt;&gt;&gt; numbers</code> [1, 1, 2, 2, 3, 3]
<code>pop()</code>	This method, exceptionally returns a value (from the list) and changes the list itself. Removes the last item from the list and returns it. <code>&gt;&gt;&gt; numbers = [1, 2, 3, 1, 2, 3]</code> <code>&gt;&gt;&gt; numbers.pop()</code> 3 <code>&gt;&gt;&gt; numbers</code> [1, 2, 3, 1, 2]

# Chapter 13

## String Formattig

## Python formatting codes

The on-line python documentation for the complete set of formatting codes can be found at <http://docs.python.org/library/stdtypes.html#string-formatting-operations> and goes further than this quick document. We have tried to select the most useful codes here.

### Strings

We consider the string “Hello, world!”.

Formatting code	Hello, world!
%s	'Hello, world'
%20s	'Hello, world!'
%-20s	'Hello, world!      '
%3s	'Hello, world'

### Integers

We consider the integers 12,345 and -12,345.

Formatting code	12,345	-12,345
%d	'12345'	'-12345'
%20d	'          12345'	'          -12345'
%-20d	'12345          '	'-12345          '
%020d	'000000000000000012345'	'-000000000000000012345'
%+d	'+12345'	'-12345'
%+20d	'          +12345'	'          -12345'
%+-20d	'+12345          '	'-12345          '
%+020d	'+000000000000000012345'	'-000000000000000012345'
%3d	'12345'	'-12345'

## Floating point numbers

We consider the floating point numbers 12.34567 and -12.34.

The %f formatting code presents data in decimal notation. The %e code does it in exponential form.

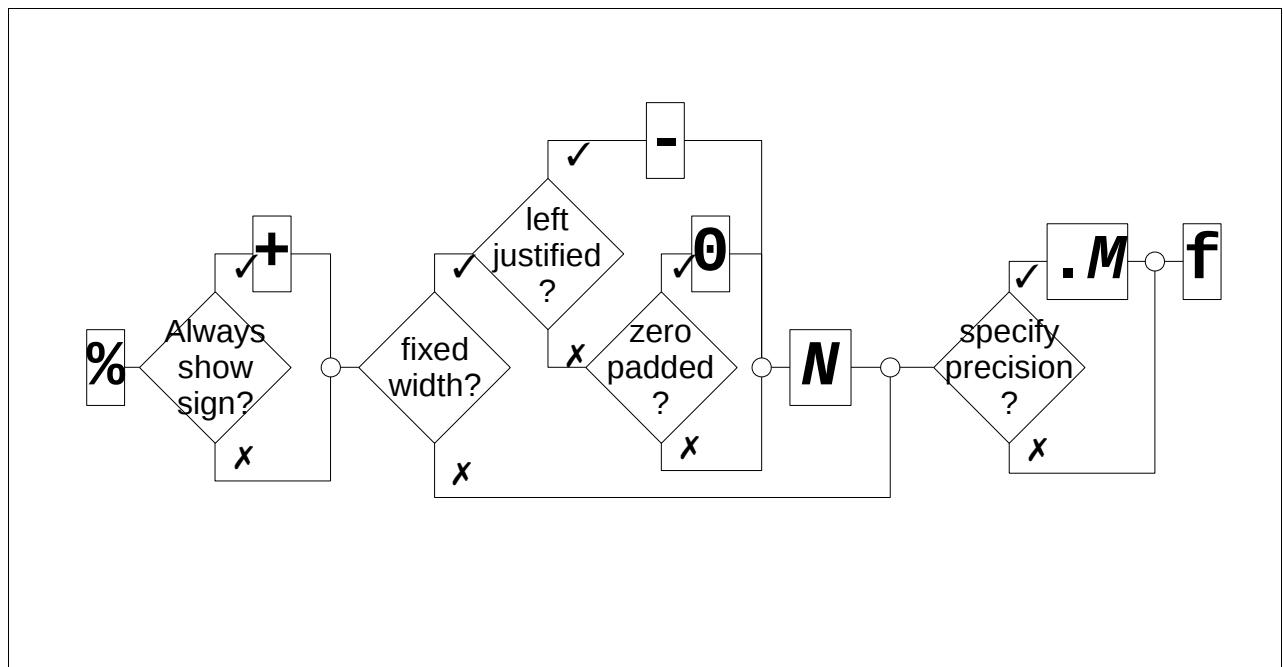
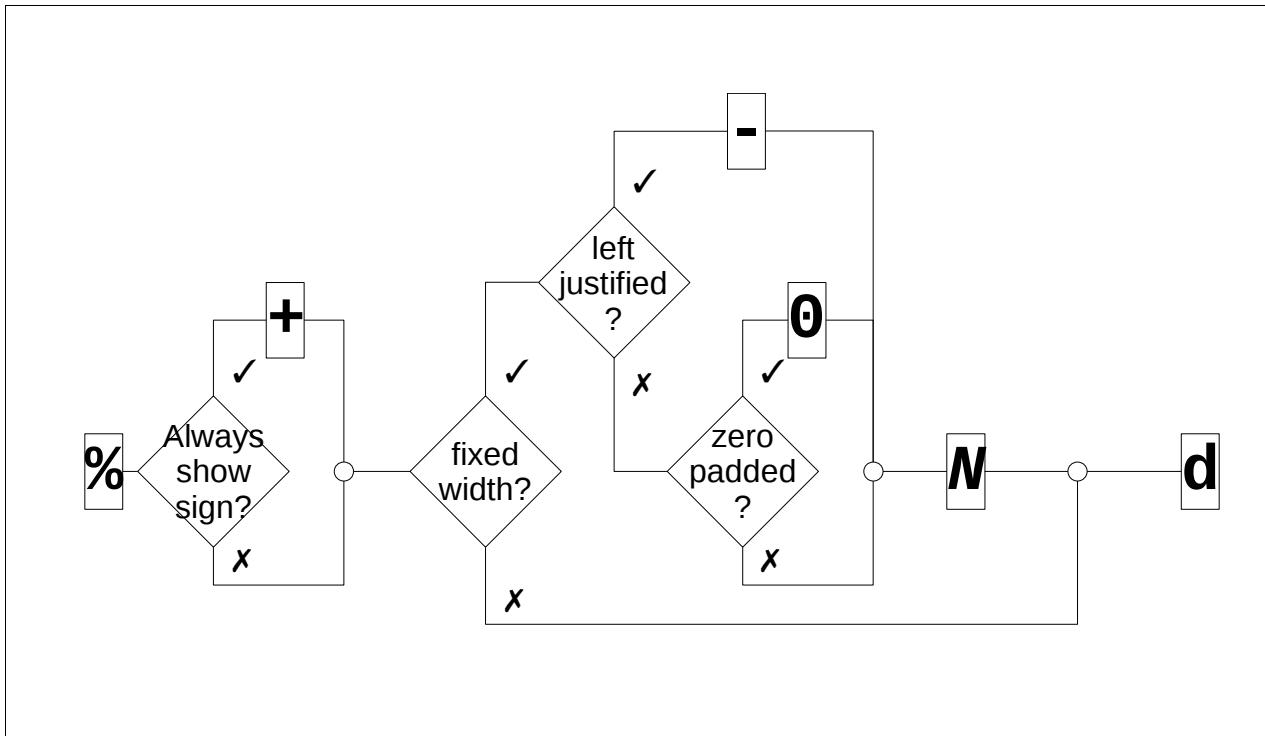
Formatting code	12.34567	-12.34
%f	'12.345670'	'-12.340000'
%20f	' 12.345670'	' -12.340000'
%-20f	'12.345670 '	'-12.340000 '
%020f	'000000000012.345670'	'-000000000012.340000'
%+f	'+12.345670'	'-12.340000'
%+20f	' +12.345670'	' -12.340000'
%+-20f	'+12.345670 '	'-12.340000 '
%+020f	'+000000000012.345670'	'-000000000012.340000'
%.4f	'12.3457'	'-12.3400'
%20.4f	' 12.3457'	' -12.3400'
%-20.4f	'12.3457 '	'-12.3400 '
%020.4f	'0000000000000012.3457'	'-0000000000000012.3400'
%+.4f	'+12.3457'	'-12.3400'
%+20.4f	' +12.3457'	' -12.3400'
%+-20.4f	'+12.3457 '	'-12.3400 '
%+020.4f	'+0000000000000012.3457'	'-0000000000000012.3400'

Formatting code	12.34567	-12.34
%e	'1.234567e+01'	'-1.234000e+01'
%20e	' 1.234567e+01'	' -1.234000e+01'
%-20e	'1.234567e+01 '	'-1.234000e+01 '
%020e	'00000001.234567e+01'	'-00000001.234000e+01'
%+e	'+1.234567e+01'	'-1.234000e+01'
%+20e	' +1.234567e+01'	' -1.234000e+01'
%+-20e	'+1.234567e+01 '	'-1.234000e+01 '
%+020e	'+00000001.234567e+01'	'-00000001.234000e+01'
%.4e	'1.2346e+01'	'-1.2340e+01'
%20.4e	' 1.2346e+01'	' -1.2340e+01'
%-20.4e	'1.2346e+01 '	'-1.2340e+01 '
%020.4e	'0000000001.2346e+01'	'-0000000001.2340e+01'
%+.4e	'+1.2346e+01'	'-1.2340e+01'
%+20.4e	' +1.2346e+01'	' -1.2340e+01'
%+-20.4e	'+1.2346e+01 '	'-1.2340e+01 '
%+020.4e	'+0000000001.2346e+01'	'-0000000001.2340e+01'

## Formatting graphs

Some people find flow charts easy to read. If you are one of these people you may find the following two graphs an aid to understanding formatting. If you're not; don't worry about it.

In both charts,  $N$  is the number of characters assigned to the formatted representation of the number. In the floating point chart,  $M$  is the number of decimal places.



# Chapter 14

## Built-in Modules

## Python modules on PWF Linux

This is an non-exhaustive list of Python modules on PWF Linux. Most are standard modules for the version of Python we run (2.6) and would be found on any similar Python installation. Those that have been installed specially for PWF Linux are marked with an asterisk.

An alphabetic list of every module shipped with Python can be found at

<http://docs.python.org/modindex.html>

though the quality of the documentation varies wildly.

Please note that these modules are not in alphabetic order, but grouped thematically. This is not meant to be a reference for you to look up what a module does (use the URL above for that) but rather a quick skim read to see what there is.

Name	*	Description
os		access to operating system-specific functions (see the course “Python: Operating System Access” for some of the ways this module can be used)
sys		access to common system functionality
platform		access to underlying platform’s identifying data
subprocess		call external commands and get access to their standard input, standard output, standard error and return code (available in Python 2.4 and later; see the course “Python: Operating System Access” for details)
tempfile		securely generate temporary files and directories (see the course “Python: Further Topics” for details)
getopt		parsing command lines, with <code>--verbosity=4</code> , <code>--verbose</code> and <code>-v</code> style options
math		access to the set of (floating point) mathematical functions defined by the C standard
cmath		the complex equivalent of math
random		pseudo-random number generators for various distributions
numpy	*	A set of functions and types suitable for numerical processing of arrays of various sorts of numbers (integers, floats, complex). (See the course “Python: Interoperation with Fortran” for examples of how this module might be used.) For more details on NumPy, see: <a href="http://numpy.scipy.org/">http://numpy.scipy.org/</a> and <a href="http://www.scipy.org/Documentation">http://www.scipy.org/Documentation</a>
scipy	*	A scientific computing package built on top of NumPy. For more details on SciPy, see: <a href="http://www.scipy.org/Documentation">http://www.scipy.org/Documentation</a>
re		regular expressions (see the course “Python: Regular Expressions” for details)
csv		encoding and decoding of data in comma separated value format as commonly used by spreadsheets and relational databases (see the course “Python: Further Topics” for details)
base64		encoding and decoding for Base64 encoded data, a format commonly used to transfer data files in email
binhex		encoding and decoding for files in binhex4 format, a format allowing representation of Macintosh files in ASCII
uu		encoding and decoding files in uuencode format, allowing arbitrary binary data to be transferred over ASCII-only connections

Name	*	Description
pickle		Serializing and de-serializing Python object structures for storage. (See the course “Python: Checkpointing” for details.)
cPickle		An implementation of the <code>pickle</code> module written in C rather than Python for improved performance. (Both modules provide equivalent structures.) The <code>pickle</code> module is better for testing and debugging; the <code>cPickle</code> module is better at run time. (See the course “Python: Checkpointing” for details.)
Gnuplot	*	Provides an interface to the gnuplot data and function plotting package, allowing the use of gnuplot from within Python. (See the course “Python: Further Topics” for examples of how this module might be used.) For more details on this module, see: <a href="http://gnuplot-py.sourceforge.net/">http://gnuplot-py.sourceforge.net/</a>
ploticus	*	Provides an interface to the ploticus API, allowing the use of ploticus-related functions from within Python. ploticus is an alternative to gnuplot for producing plots, charts and graphics from data. (See the course “Python: Further Topics” for examples of how this module might be used.) For more details on this module, see: <a href="http://www.srcc.lsu.edu/pyploticus.html">http://www.srcc.lsu.edu/pyploticus.html</a>
matplotlib	*	A plotting library that allows MATLAB®-style plotting in Python. (See the course “Python: Further Topics” for examples of how this module might be used.) For more details on this module, see: <a href="http://matplotlib.sourceforge.net/">http://matplotlib.sourceforge.net/</a>
Image	*	Main module of the Python Imaging Library (PIL). PIL provides fairly powerful image processing capabilities and supports a large number of image file formats. For more details on PIL, see: <a href="http://www.pythonware.com/products/pil/index.htm">http://www.pythonware.com/products/pil/index.htm</a>
sqlite3		modules for interfacing to a simple SQL database built around local files (shipped with Python 2.5 and later)
anydbm		generic interface to variants of the DBM database
bz2		interface for the bz2 compression library
gzip		interface for the zlib compression library for reading and writing gzip files
zlib		interface for the zlib compression library
zipfile		work with zip files
unittest		The Python unit testing framework. (See the course “Python: Unit Testing” for details.)



# **Part I**

## **Python Labs**



# Chapter 15

## Python Syntax

### 15.1 Variables and Data Types

Python was developed to be clear, powerful, and fun to use. We'll start with a quick note on what Python is and how it's used, then we'll jump right into writing code! This course assumes no prior knowledge in Python.

Python is a powerful, flexible programming language you can use in web/Internet development, to write desktop graphical user interfaces (GUIs), create games, and much more. Python is:

1. High-level, meaning reading and writing Python is really easy—it looks a lot like regular English!
2. Interpreted, meaning you don't need a compiler to write and run Python! You can write it here at Codecademy or even on your own computer (many are shipped with the Python interpreter built in—we'll get to the interpreter later in this lesson).
3. Object-oriented, meaning it allows users to manipulate data structures called objects in order to build and execute programs. We'll learn more about objects later.
4. Fun to use. Python is named after Monty Python's Flying Circus, and example code and tutorials often refer to the show and include jokes in order to make learning the language more interesting.

This course assumes no previous knowledge of Python in particular or programming/computer science in general.

#### 15.1.1 Variables

One of the most basic concepts in computer programming is the variable. A variable is a word/identifier that hangs onto a single value. For example, let's say you needed the number 5 for your program, but you're not going to use it immediately. You can set a variable, say `spam`, to grab the value 5 and hang onto it for later use, like this:

```
spam = 5
```

Declaring variables in Python is easy; you just write out a name/identifier, like spam, and use = to assign it a value, and you're done!

## INSTRUCTIONS

Set the variable my\_variable to the value 10.

### 15.1.2 Data Types

Great! We can now summon the value 10 by calling out the name my\_variable whenever we need it.

In this case, the data type of my\_variable is an integer (a positive or negative whole number). There are three data types in Python that are of interest to us at the moment: integers, floats (fractional numbers written with a decimal point, like 1.970), and booleans (which can be True or False).

Computer programs, in large part, are created to manipulate data. Therefore, it's important to understand the different types of data (or "datatypes") that we can incorporate into our programs.

Never use quotation marks (' or ") with booleans, and always capitalize the first letter! Python is case-sensitive (it cares about capitalization). We'll use quotation marks when we get to strings, which we'll cover in the next unit.

## INSTRUCTIONS

Set the following variables to the corresponding values:

- my\_int to the value 7
- my\_float to the value 1.23
- my\_bool to the value True

### 15.1.3 You've Been Reassigned

Great work. You now know how to declare variables in Python and set them to different values, and you've learned about three different types of values: integers, floats, and booleans.

You can reassign a variable at any point. If you first set my\_int to 7 but later want to change it to 3, all you have to do is tell Python my\_int = 3, and it'll change the value of my\_int for you.

Try it and see!

## INSTRUCTIONS

Change the value of my\_int from 7 to 3 in the editor.

## 15.2 Whitespace and Statements

Now that you're getting the hang of variables, values, and assignment (think of them like the subjects, objects, and verbs of English sentences), let's take a look at whitespace and statements: the sentences of our new Python language.

### 15.2.1 What's a Statement?

You can think of a Python statement as being similar to a sentence in English: it's the smallest unit of the language that makes sense by itself. Just like "I," "like," and "Spam" aren't statements by themselves, but "I like Spam" is, variables and data types aren't statements in Python, but they are the building blocks that form them.

To continue the sentence analogy, it's clear that we also need a kind of punctuation to make it obvious where one statement ends and another begins. If you're familiar with JavaScript, you know that statements end with a semicolon (;). In Python, statements are separated by whitespace. Just like you can't toss around semicolons wherever you want in JS, you can't throw whitespace around in Python.

This may take some getting used to, especially if you're coming from a programming language where whitespace doesn't matter.

### INSTRUCTIONS

Don't worry about understanding the code on the right; just hit "run" to see what happens. You should see an error message due to badly formatted code. We'll fix it in the next exercise!

```
def spam():
    eggs = 12
    return eggs

print spam()
```

### 15.2.2 Whitespace Means Right Space

Notice the error you got when you ran the code in the editor:

IndentationError: expected an indented block You'll get this error whenever your Python whitespace is out of whack. (If you've studied JavaScript, think of improper whitespace as improper use of ; or .) When your punctuation's off, your meaning can change entirely:

The peasant said, "The witch turned me into a newt!" "The peasant," said the witch, "turned me into a newt!" See what we mean?

### INSTRUCTIONS

Properly indent the code to the right by hitting the spacebar key on your keyboard four times on line 2 (before eggs) and another four times on line 3 (before return). Click "run" once you've done this.

### 15.2.3 A Matter of Interpretation

In the introduction to this unit, we mentioned that Python is an interpreted language (meaning it runs using an interpreter). In the context of Codecademy, the interpreter is the console/output window in the lower right corner of the page.

For now, think of the interpreter as a program that takes the code you write, checks it for syntax errors, and executes the statements in that code, line by line. It works as a go-between for you and the computer and lets you know the result of your instructions to the machine.

#### INSTRUCTIONS

Tell Python to assign the value `True` to the variable `spam` and `False` to the variable `eggs`.

## 15.3 Comments

Good comments make programs more readable and will help you diagnose problems when they arise. Get in the habit of commenting up your code!

### 15.3.1 Single Line Comments

**Single Line Comments** You may have noticed the instructions in the editor that begin with a `#` (pound or hash) symbol. These lines of code are called comments, and they aren't read by the interpreter—they don't affect the code at all. They're plain English comments written by the programmer to provide instructions or explain how a particular part of the program works.

Since this improves the readability of your code tremendously (and will help you debug programs more quickly, since you'll be able to tell at a glance what each part of the program is supposed to do), we encourage you to comment on your code whenever its purpose isn't immediately obvious.

#### INSTRUCTIONS

Write a comment on line 1 in the editor. Make sure it starts with `#!` (It can say anything you like.)

### 15.3.2 Multi-Line Comments

Sometimes you have to write a really long comment. `#` will only work across a single line, and while you could write a multi-line comment and start each line with `#`, that can be a pain.

If you want to write a comment across multiple lines, you can include the whole block in a set of triple quotation marks, like so:

```
""" I'm a lumberjack and I'm okay
I sleep all night and I work all day! """
```

**INSTRUCTIONS**

Write a multi-line comment in the editor. Include whatever text you want!

## 15.4 Math Operations

If all we could do in Python were declare variables and write comments, it wouldn't be very exciting. Thankfully, that's not the case—we can combine and manipulate data to create powerful, flexible programs to suit our needs.

### 15.4.1 Arithmetic Operators

Python's statements aren't limited to simple expressions of assignment like `spam = 3`; they can also include mathematical expressions using arithmetic operators.

There are six arithmetic operators we're going to focus on:

- Addition (+)
- Subtraction (-)
- Multiplication (\*)
- Division (/)
- Exponentiation (\*\*)
- Modulo (%)

**INSTRUCTIONS**

Let's start with addition. Set the variable `count_to` to the sum of `1 + 2`.

### 15.4.2 Subtraction

Good! Now let's try subtraction.

**INSTRUCTIONS**

We've clearly counted too far with our `count_to`—we've gotten to five but want `count_to` to be smaller. Tell Python to reassign `count_to` to `5 - 2`.

### 15.4.3 Multiplication

Perfect! Now let's try a little multiplication.

**INSTRUCTIONS**

The Knights Who Say "Ni!" have only said "Ni!" twice. Let's make it twenty times by multiplying `2 * 10`.

### 15.4.4 Division

On second thought, 20 "Ni!"s might be a bit much. Let's use division to get it down to 5.

#### INSTRUCTIONS

Set the value of ni to 5 (that is,  $20 / 4$ ).

### 15.4.5 Exponentiation

Excellent job!

All the arithmetic operations you've done so far have probably either been intuitive or have resembled work you've done in other programming languages (such as JavaScript). Exponentiation, however, might be a new one, so it bears some explaining.

The `**` operator raises the first number, the base, to the power of the second number, the exponent. So if you type `2 ** 3`, you get 8 (`2 ** 3` is the same as `2 * 2 * 2`, both of which equal 8).  $5^{**} 2 = 25$ ,  $2^{**} 4 = 16$ , and so on.

#### INSTRUCTIONS

Our lumberjack is super hungry and wants 100 eggs. Set eggs to 100 using exponentiation.

### 15.4.6 Modulo

Impressive! That was a lot of eggs. Hope you left room for spam!

Our final arithmetic operator is modulo (also called modulus). Modulo returns the remainder left over by integer division. So, if you ask the Python interpreter to evaluate `5 % 2`, it will return 1 (since 2 goes into 5 evenly two times, with 1 left over).  $10 \% 5$  is 0,  $113 \% 100$  is 13, and so on.

#### INSTRUCTIONS

Turns out you did leave room for spam—but not much! Set spam equal to 1 using modulo. You can use any two numbers that will leave a remainder of 1 to do this.

## 15.5 Review

### 15.5.1 Bringing It All Together

Nice work! So far you've learned about the following in Python:

- Variables, which are ways to store values for later use;
- Data types (such as integers, floats, and booleans);
- Whitespace (and why it's significant!);

- Statements (and how Python statements are like statements in regular English);
- Comments (and why they're good for your code!); and
- Arithmetic operations (including +, -, \*, /, \*\*, and

## INSTRUCTIONS

Let's put all our knowledge to work.

- Write a single-line comment on line 1. It can be anything! (Make sure it starts with # !)
- Declare a variable, monty, and set it equal to True.
- Declare another variable, python, and set it equal to 1.234.
- Declare a third variable, monty\_python, and set it equal to python squared.

## 15.6 Project: Tip Calculator

### 15.6.1 Your Favorite Meal

This project is designed to complement Unit 1: Python Syntax. It assumes familiarity with only the material covered in that course.

You've just feasted on a truly delicious meal of Spam and eggs. The diner's computer is down, however, so you'll need to compute the cost of your meal yourself.

Here's how it'll break down:

- Cost of meal: \$44.50
- Restaurant tax: 6.75%
- Tip: 15%

You'll want to apply the tip to the overall cost of the meal (including tax).

## INSTRUCTIONS

First, let's declare a variable meal and assign it the value 44.50.

### 15.6.2 The Tax

Good! Now let's create a variable for the tax percentage.

The tax on your meal at this diner is 6.75%. Because we'll be multiplying with floats and not percentages, however, you'll have to divide 6.75 by 100 in order to get the decimal form of the number. Do you understand why? (See the Hint if you're not sure.)

## INSTRUCTIONS

Create the variable tax and set it equal to the decimal value of 6.75

### 15.6.3 The Tip

Nice work! You received good service at this diner, so you'd like to leave a 15% tip on top the cost of the meal (including tax).

Before we compute the tip for your overall bill, let's set a variable for the tip (15%). Again, this is a percentage, so you'll need to divide 15.0 by 100 in order to get the decimal form of the tip.

## INSTRUCTIONS

Set the variable tip to 15% (in decimal form!) on line 5. (You can just type 0.15 directly, if you like.)

### 15.6.4 Reassign in a Single Line

Okay! We've got the three variables we need to perform our calculation, and we know a bunch of arithmetic operators that will be able to help us out.

We saw in Lesson 1 that we could reassign a variable when needed—for example, we could say `spam = 7` at one point in the program, then later change our minds and say `spam = 3`.

## INSTRUCTIONS

On line 7 in the editor, reassign `meal` to the value of `itself + itself * tax` (this will add the dollar amount of the tax to the cost of the meal). You're completely allowed to reassign a variable in terms of itself! We're only calculating the cost of meal and tax here—we'll get to the tip soon!

### 15.6.5 Second Verse, Same as the First

Well, almost the same. Here, we're going to introduce a new variable instead of resetting an existing variable.

Now that `meal`'s got the cost of the food + tax, let's introduce a new variable on line 8, `total`, that's equal to the new `meal + meal * tip`. (This should be very similar to what you just did in the last exercise.)

The code on line 10 will nicely format the value of `total` and will ensure it prints to the console with two numbers after the decimal. (We'll learn all about string formatting, the console, and the `print` keyword in Unit 2!)

## INSTRUCTIONS

Assign the variable `total` to the sum of `meal + meal * tip` on line 8, then hit "run" to see the total cost of your meal!

# Chapter 16

## Strings and Console Output

This tutorial will show you strings and console output for Python, including creating literal strings, calling a variety of methods of strings, and use the command "print".

### 16.1 Strings

Strings are an important and extremely common data type in Python. Here, we'll show you how to build and manipulate them!

#### 16.1.1 Step One: Strings

(This course assumes familiarity with the material presented in Lesson 1: Python Syntax.)

Another useful data type in Python is the string. Strings are, well, strings of characters, which is a more formal way of saying they're really just regular English phrases. They can include numbers, letters, and various symbols, like so: "We're #1!"

A string literal is a string created by literally just writing it down between quotation marks (' ' or " "). You have to use the same type of quotation mark on each end of the string, though—no 'string"s or "string's!

#### INSTRUCTIONS

Assign the string "Always look on the bright side of life!" to the variable brian.

#### 16.1.2 Step Two: Things

Excellent! Let's get a little practice in with strings. Set the following variables to the following phrases:

#### INSTRUCTIONS

- Set caesar to "Graham"
- Set praline to "John"

- Set viking to "Teresa"

### 16.1.3 Step Three: Escape!

Don't get too comfortable: you can't use just any symbol in a string, and some results can only be achieved by special characters. The practice of including these characters in a string requires that these strings be escaped, or marked as unique. The backslash character (\) does this work for us! We just put it directly before the character we want to escape.

#### INSTRUCTIONS

The string in the editor is broken: the apostrophe in I'm makes Python think the single-quote string ends there. Fix it by escaping the ' in I'm!

### 16.1.4 Access by Offset

Great work! (By the way, you could also have repaired the string by replacing the single quotes on the ends with double quotes, like this: "Help! Help! I'm being repressed!".)

Remember how we told you that strings were, technically speaking, strings of characters? Wouldn't it be nifty if you could get to each character in a string individually?

Well, you can!

Each character in a string has a subscript or offset, which is a fancy way of saying it has a number attached to it. The number starts at 0 for the leftmost character and increases by one as you move character-by-character to the right. Check out the diagram in the editor!

#### INSTRUCTIONS

```
"""
The string "PYTHON" has six characters,
numbered 0 to 5, as shown below:

+---+---+---+---+---+
| P | Y | T | H | O | N |
+---+---+---+---+---+
  0   1   2   3   4   5

So if you wanted "Y", you could just type
"PYTHON"[1] (always start counting from 0!)
"""

fifth_letter =
print fifth_letter
```

When you think you've got the hang of the code in the editor, set fifth\_letter equal to the fifth letter of the string "MONTY", like so: "MONTY"[:] (but replace the ? with the correct number).

## 16.2 String Methods

Modifying strings is useful, but doing it manually is a pain. Thankfully, Python includes a number of methods that can automate this work for us.

### 16.2.1 Four Methods to the Madness

Great work! Now we're going to talk about some of the methods that are available for use with strings.

We'll explain methods much more in future lessons, but for now, the takeaway is that string methods are pre-built pieces of code that perform specific tasks for strings.

We're going to focus on four string methods in this section:

- `len()`
- `lower()`
- `upper()`
- `str()`

Let's start with `len()`, which gets the length of a string!

#### INSTRUCTIONS

Create a variable called `parrot` and set it to the string "Norwegian Blue" (be sure to include the space and capitalize exactly as shown!). Then type `len(parrot)` on line 5, after the word `print`, like so: `print len(parrot)`. This will print out the number of letters in "Norwegian Blue"!

### 16.2.2 `lower()`

Well done! Again, `len()` returns the length—that is, the number of characters—of the string on which it's called.

Let's say that you don't want any capitalization in your string, though (in this case, "Norwegian Blue"). In this example, it's a tiny amount of work to manually change "Norwegian Blue" to "norwegian blue". But what if you wanted to convert thousands of words to all-lower case? Doing it manually would take forever.

#### INSTRUCTIONS

Python is all about automation! Call `lower()` on `parrot` (after the word `print`) on line 5 in the editor, like so: `parrot.lower()`. This will make the string all lower-case! (This is different from the way you call `len()`, and we'll explain the reasoning behind this before this section is over.)

### 16.2.3 `upper()`

Perfect! Now your string is 100% lower case.

## INSTRUCTIONS

Unfortunately, you just realized you actually need your string to be completely upper case, not lower. Call `upper()` on `parrot` (after the word `print` on line 5) in order to fix this in one fell swoop!

### 16.2.4 str()

It looks like you're really getting the hang of string methods in Python. In case you're getting a bit bored (and we know adjusting string capitalization isn't the most exciting thing in the world), try the `str()` method on for size!

The `str()` method returns a string containing a nicely printable representation of whatever you put between the parentheses. It makes strings out of non-strings! For example,

`str(2)` would turn 2 into "2".

## INSTRUCTIONS

Two steps here:

- Create a variable `pi` and set it to 3.14 on line 4.
- Call `str(pi)` on line 5, after the `print` keyword.

### 16.2.5 Dot Notation

As promised, we'll now explain the reason you use `len(string)` and `str(object)`, but dot notation (e.g. `"String".upper()`) for the rest.

Dot notation works on string literals (`"The Ministry of Silly Walks".upper()`) and variables assigned to strings (`ministry.upper()`) because these methods are specific to strings—that is, they don't work on anything else.

By contrast, `len()` and `str()` can work on a whole bunch of different objects (which we'll get to later), so they can't be tied just to strings with dot notation.

## INSTRUCTIONS

Let's do just a bit more practice. Call `len()` on `ministry` on line 4 and `upper()` on line 5 (do this after the `print` keyword each time).

## 16.3 Print

From time to time, we need to see the output of our Python programs. The "print" keyword makes this possible!

### 16.3.1 Printing with String Literals

The area to the right of these instructions is the editor, which is where we've been writing our code.

Python translates your instructions to instructions the computer can understand with an interpreter. You can think of the interpreter as a little program that ferries information between your Python code and the computer when you click "Save & Submit Code." The actual window to which the interpreter spits out the output of your code is the console (the window in the upper right).

If you're familiar with JavaScript, then you know that `console.log` logs the result of evaluating your code to the console; `print` is Python's version of `console.log`.

If you haven't studied JavaScript, never fear! All you need to know is that `print` prints the result of the interpreter's evaluation of your code to the console for you to see.

#### INSTRUCTIONS

Let's start with something simple. Try printing "Monty Python" to the console. The syntax looks like this:

```
print "Your string goes here"
```

Don't forget the quotes (' or ")!

### 16.3.2 Printing with Variables

Great! Now let's combine what we've learned about variables with our new `print` keyword.

#### INSTRUCTIONS

Declare a variable called `the_machine_goes` and assign it the string value "Ping!" on line 5. Make sure to type "Ping!" exactly as shown—complete with capital "P" and an exclamation point! Go ahead and print `the_machine_goes` on line 6.

## 16.4 Advanced Printing

Now that you know how to print output to the console, let's take a look at some of the more complex features supported by "`print`".

### 16.4.1 String Concatenation

You know about strings, and you know about arithmetic operators. But did you know some arithmetic operators work on strings?

If you use the `+` operator between two strings, it concatenates them (glues them together).

```
print "Monty " + "Python" will print out "Monty Python"!
```

## INSTRUCTIONS

Give it a go in the editor. print the concatenated strings "Spam ", "and ", "eggs" on line 3 to print the string "Spam and eggs" to the console.

### 16.4.2 Explicit String Conversion

Remember when we talked about the str() method a couple of lessons back, and how it turns non-strings into strings? The fancy name for that process is explicit string conversion.

You're explicitly telling Python, "Hey, I know this isn't a string, but I want to turn it into one." Contrast this with just putting quotes around a sequence of characters to make it a string.

Making a number into a string can let you glue together strings and numbers (which Python normally won't allow). Check it out:

```
print "I have " + str(2) + " coconuts!" will print "I have 2 coconuts!"
```

## INSTRUCTIONS

- Run the code as-is. You get an error!
- Use str() to turn 3.14 into a string, then run the code again.

### 16.4.3 String Formatting with %, Part 1

Awesome work so far. This is the last new thing to cover before we review!

We saw earlier that you can access individual characters in a string by offset, or, if you want to think about it this way, ID number. (Remember, "PYTHON"[1] is "Y", not "P"!).

Unfortunately, strings in Python are immutable—you can't change them once they're created.

However, there is a way you can work flexibility into your strings, and that's with string formatting. It uses the % symbol (don't confuse this with modulo!), and you can sort of think of it as a variable for your string.

## INSTRUCTIONS

Take a look at the code in the editor. What do you think it'll do? Click Save & Submit Code when you think you know.

```
string_1 = "Camelot"
string_2 = "place"

print "Let's not go to %s. 'Tis a silly %s." % (string_1, string_2)
```

### 16.4.4 String Formatting with %, Part 2

Did you see that? The % string formatter replaced the %s (the "s" is for "string") in our string with the variables in parentheses. (We could have done that by just putting "Camelot" and "place" in parentheses after the string, but we wanted to show you how it works with variables.)

The syntax went like this:

```
print "%s" % (string_variable)
```

You can have as many variables (or strings!) separated by commas between your parentheses as you like:

```
print "The %s who %s %s!" % ("Knights", "say", "Ni")
prints "The Knights who say Ni!"
```

## INSTRUCTIONS

For our grand finale, we're showing you a bit of new code. Don't worry if you don't get how it works yet; we'll explain it soon! For now, replace the \_\_\_\_s with the form of % you need to complete your quest: %s inside the string, and % to link the string with its arguments. Answer the questions in the console as they pop up!

```
name = raw_input("What is your name?")
quest = raw_input("What is your quest?")
color = raw_input("What is your favorite color?")

print "Ah, so your name is ___, your quest is ___, " \
"and your favorite color is ____." ___ (name, quest, color)
```

## 16.5 Review

Let's take a moment to go over what we've learned.

And Now, For Something Completely Familiar Great job! You've learned a lot in this unit, including:

What strings are, and how to create them literally (using ' ' or " ") or explicitly (using the str() method); string methods, such as len(), upper(), and lower(); the print keyword for outputting Python's evaluation of your code to the console; and advanced printing techniques using %. Let's wrap it all up!

### 16.5.1 INSTRUCTIONS

1. Create a variable called my\_string and set it to any string you'd like.
2. Go ahead and print its len()gth on line 4.
3. Go for the gold and print its .upper() case version on line 5.

## 16.6 Project: Date and Time

### 16.6.1 The `datetime` Library

In this small project, we'll create a program that experiments with Python's ability to give us the current date and time. This will give you some practice with printing strings, concatenation, and the `str()` explicit conversion function.

At the end of this section, you'll know how to print the date and time in the following format: mm/dd/yyyy hh:mm:ss.

On line 1, notice the statement from `datetime import datetime`. Importing special functionality into your programs will be covered in Unit 4's discussion of the `import` statement.

For now, just know that we're telling the Python interpreter to give our program the ability to retrieve the current date and time.

In the next exercise, we'll cover how to explicitly retrieve this information!

### INSTRUCTIONS

```
from datetime import datetime
```

### 16.6.2 Getting the Current Date and Time

To retrieve the current date and time, we can use a function called `datetime.now()` to get that information.

In a later course, you'll learn all about functions. For now, just know that `datetime.now()` calls on a piece of code that comes with Python that figures out the current date and time for us.

### INSTRUCTIONS

1. Create a variable called `now` and store the result of `datetime.now()` in it. We'll use this variable in the next exercise.
2. Go ahead and print the value of the variable `now`.

### 16.6.3 Extracting Information

Notice how we got an output of the form 2012-07-19 12:50:53.180759. That's pretty ugly.

Let's examine how to extract portions of the date and time to eventually print out a "prettier" form of this information.

Let's start by retrieving the month, day, and year from the result of `datetime.now()`. To do this, we can use our variable `now` in the following way: `current_year = now.year`.

Of course, the variable on the left-hand side of the assignment could be named anything.

The fact that we can extract parts of the date in such an elegant syntax is pretty awesome. As you could guess, we can use a similar syntax to extract the month and day.

Note: Don't worry about the details of the notation now.year. It's called dot notation and it's used to access data from an object. We mentioned this briefly in Unit 2 and will talk much more about objects later.

## INSTRUCTIONS

Go ahead and print out the current month, day, and year to the console on separate lines.

### 16.6.4 Hot Date

Great job printing out the date's components! In gearing up for our ultimate goal of printing out mm/dd/yyyy hh:mm:ss, let's tackle adding / slashes to the date's parts.

You might think to do something like:

print now.month, "/", now.day, "/", now.year However, this would incorrectly give you spaces between the slashes. Hence, the better solution is to use string concatenation (the + operator), covered in Unit 2.

As you'll see, it's not as simple as just using concatenation—mainly because concatenation only works with strings.

When you extract information like now.year, you end up with an integer (a positive or negative whole number). To convert an integer to a string, you can use the str() function. For example, if a variable x had the value 4 and we wanted to convert that into "4", you could type:

```
str(x)
```

## INSTRUCTIONS

Print out the current date in the pretty form of mm/dd/yyyy. It's totally okay if it comes out as m/d/yyyy.

### 16.6.5 Pretty Time

Nice work! Let's do the same for the parts of the time—namely, the hour, minute, and second.

As you might guess, we can also use our variable now to print out the time information. If you wanted to print out the current hour, you could do:

```
current_hour = now.hour
```

Just for clarification, our variable now contains the results of datetime.now(); there's nothing special about naming the variable "now." It's just for convenience!

**INSTRUCTIONS**

Similar to the last exercise, print out the current time in the pretty form of hh:mm:ss. Remember to use string concatenation. Note: It's also okay if you end up with h:m:s.

**16.6.6 Grand Finale**

So far, we've managed to prettily print the date and time separately. Let's combine the two!

**INSTRUCTIONS**

Print out the date and time together in the form: mm/dd/yyyy hh:mm:ss (Note that a space separates the date and time, so you'll need the + operator once more.)

# Chapter 17

## Conditionals and Control Flow

In this course, we'll learn how to create programs that generate different outcomes based on user input!

### 17.1 Introduction to Control Flow

"Control flow" is the order in which events occur in a program. This section will introduce the concept of control flow and the ways in which it can be altered.

#### 17.1.1 Go With the Flow

(This course assumes familiarity with the material presented in Unit 1: Python Syntax and Unit 2: Strings & Console Output. From here on out, take for granted that each new course assumes knowledge of the material presented in the previous courses.)

You may have noticed that the Python programs we've been writing so far have had sort of one-track minds. They compute the sum of two numbers or print something to the console, but they don't have the ability to pick one outcome over another—say, add two numbers if their sum is less than 100, or instead print the numbers to the console without adding them if their sum is greater than 100.

Control flow allows us to have these multiple outcomes and to select one based on what's going on in the program. Different outcomes can be produced based on user input or any number of factors in the program's environment. (The environment is the technical name for all the variables—and their values—that exist in the program at a given time.)

#### INSTRUCTIONS

Check out the code in the editor. To help keep you motivated, we've provided a glimpse into the not-so-distant future: the type of program you'll be able to write once you've mastered control flow. Click Save & Submit Code to see what happens!

```
def clinic():
    print "You've just entered the clinic!"
    print "Do you take the door on the left or the right?"
```

```

answer = raw_input("Type left or right and hit 'Enter'.").lower()
if answer == "left" or answer == "l":
    print "This is the Verbal Abuse Room, you heap of parrot droppings!"
elif answer == "right" or answer == "r":
    print "Of course this is the Argument Room, I've told you that already!"
else:
    print "You didn't pick left or right! Try again."
    clinic()

clinic()

```

## 17.2 Comparators

Comparators help programs make decisions by establishing relationships between variables: for instance, which values are greater than, less than, or equal to others.

### 17.2.1 Compare Closely!

Let's not get ahead of ourselves. First, we'll start with the simplest aspect of control flow: comparators. There are six of them, and we're willing to bet at least a few look familiar:

```

Equal to (==)
Not equal to (!=)
Less than (<)
Less than or equal to (<=)
Greater than (>)
Greater than or equal to (>=)

```

Note that `==` is used to compare whether two things are equal, and `=` is used to assign a value to a variable.

We hope you're familiar with the ideas of greater/less than and greater/less than or equal to. They work exactly as you think they would: they test to see if a number is (or is not) equal to, greater than (or equal to), or less than (or equal to) another number.

(If you're coming from the JavaScript track: there is no `====` in Python.)

### INSTRUCTIONS

Let's run through the comparators in the editor. Set each variable to either True or False depending on what you think the result of the evaluation above it will be. For example, `1 < 2` will be True, because one is less than two.

```

# Assign True or False as appropriate on the lines below!

# 17 < 118 % 100
bool_one =

# 100 == 33 * 3 + 1
bool_two =

# 19 <= 2**4
bool_three =

```

```
# -22 >= -18
bool_four =
# 99 != 98 + 1
bool_five =
```

### 17.2.2 Compare... Closelier

Excellent! It looks like you're comfortable with basic expressions and comparators.

But what about... extreme expressions and comparators?

(This exercise may seem unnecessary to you, but we can't tell you the number of problems caused in programs by incorrect order of operations or reversed `>`s and `<`s. Bugs like this can be a serious problem!)

#### Instructions

Let's run through the comparators in the editor one more time (these expressions are more complex than what you saw in the last exercise). Again, set each variable to either True or False depending on what you think the result of the evaluation above it will be.

```
# Assign True or False as appropriate on the lines below!

# 20 + -10 * 2 > 10 % 3 % 2
bool_one =

# (10 + 17)**2 == 3**6
bool_two =

# 1**2**3 <= -(-(-1))
bool_three =

# 40 / 20 * 4 >= -4**2
bool_four =

# 100**0.5 != 6 + 4
bool_five =
```

### 17.2.3 How the Tables Have Turned

Nice work!

Based on our comparisons, you've probably guessed that comparisons in Python generate one of two results: True or False. These are instances of a data type we mentioned briefly in Unit 1 called booleans, and they are the only two instances. Things aren't "sort of True" or "Falseish" or "maybe" in Python—they are True or False (and are always capitalized, unlike in JavaScript).

Let's reverse things a bit: we'll supply the boolean value (True or False), and you write an expression that evaluates appropriately.

#### Instructions

For each boolean value in the editor, write an expression that evaluates to that value. Feel free to write expressions that are as simple or as complex as you'd like!

Remember, though: simple is better than complex!

Remember, comparators are: ==, !=, >, >=, <, and <=.

```
# Create comparative statements as appropriate on the lines below!

# Make me true!
bool_one =

# Make me false!
bool_two =

# Make me true!
bool_three =

# Make me false!
bool_four =

# Make me true!
bool_five =
```

## 17.3 Boolean Operators

Boolean (or logical) operators help programs make decisions based on whether something AND something else is true, something OR something else is true, or something is NOT true.

### 17.3.1 To Be and/or Not to Be

Boolean operators (or logical operators) are words used to connect Python statements in a grammatically correct way—almost exactly as in regular English. There are three boolean operators in Python:

and, which means the same as it does in English; or, which means "one or the other OR BOTH" (it's not exclusively one or the other, the way it often is in English); not, which means the same as it does in English.

We want to stress this second case to you: if your mom tells you you can have Monty Python and the Holy Grail or Monty Python's Life of Brian, she probably means "one or the other, but not both." Python, on the other hand, would be totally fine with your picking both, so long as you don't pick neither. Python is cooler than your mom.

Boolean operators result (predictably) in boolean values—True or False. We'll go through the three operators one by one.

#### Instructions

Before we get started with and, take a look at the truth table in the editor. (This is for those of you who like to see the bigger picture before we dive into the details.) Don't worry if you don't completely get it yet—you will by the end of this section!

```
"""
Boolean Operators
```

```

True and True is True
True and False is False
False and True is False
False and False is False

True or True is True
True or False is True
False or True is True
False or False is False

Not True is False
Not False is True

"""

```

### 17.3.2 And

The boolean operator and only results in True when the expressions on either side of and are both true. An expression is any statement involving one or more variables and operators (arithmetic, logical, or boolean). For instance:

$1 < 2$  and  $2 < 3$  results in True because it is true that one is less than two and that two is less than three.

$1 < 2$  and  $2 > 3$  results in False because it is not true that both statements are true—one is less than two, but two is not greater than three.

#### Instructions

Let's practice a bit with and. Assign the boolean values beneath each expression as appropriate. This may seem overkill, but remember: practice makes perfect.

```

# Assign True or False as appropriate on the lines below!

# False and False
bool_one =

#  $-(-(-2)) == -2$  and  $4 \geq 16^{**0.5}$ 
bool_two =

#  $19 \% 4 != 300 / 10 / 10$  and False
bool_three =

#  $-(1^{**2}) < 2^{**0}$  and  $10 \% 10 \leq 20 - 10 * 2$ 
bool_four =

# True and True
bool_five =

```

### 17.3.3 Or

The boolean operator or only returns True when either (meaning one, the other or both!) of the expressions on each side of or are true. (It's only False when both expressions are False.) For example:

$1 < 2$  or  $2 > 3$  is True, even though two is not greater than three;  $1 > 2$  or  $2 > 3$  is False, because it is neither the case that one is greater than two nor that two is greater than three.

## Instructions

Time to practice with or!

```
# Assign True or False as appropriate on the lines below!

# 2**3 == 108 % 100 or 'Cleese' == 'King Arthur'
bool_one =

# True or False
bool_two =

# 100**0.5 >= 50 or False
bool_three =

# True or True
bool_four =

# 1**100 == 100**1 or 3 * 2 * 1 != 3 + 2 + 1
bool_five =
```

### 17.3.4 Not

The boolean operator not returns True for false statements and False for true statements. Remember, the only two boolean values are True and False!

For example:

not False will evaluate to True, as will not  $40 > 41$ . Applying not to expressions that would otherwise be true makes them False.

## Instructions

Last but not least, let's get some practice in with not.

```
# Assign True or False as appropriate on the lines below!

# not True
bool_one =

# not 3**4 < 4**3
bool_two =

# not 10 % 3 <= 10 % 2
bool_three =

# not 3**2 + 4**2 != 5**2
bool_four =

# not not False
bool_five =
```

### 17.3.5 This and That (or This, But Not That!)

Fun fact: boolean operators can be chained together!

It's important to know that boolean operators are not evaluated straight across from left to right all the time; just like with arithmetic operators, where / and \* are evaluated before + and - (remember Please Excuse My Dear Aunt Sally?), there is

an order of precedence or order of operations for boolean operators. The order is as follows:

not is evaluated first; and is evaluated next; or is evaluated last.

This order can be changed by including parentheses (()). Anything in parentheses is evaluated as its own unit.

For instance, True or not False and False returns True. Can you see why? If not, check out the Hint.

Best practice: always use parentheses (()) to group your expressions to ensure they're evaluated in the order you want. Remember: explicit is better than implicit!

### Instructions

Go ahead and assign True or False as appropriate for bool\_one through bool\_five. No math in this one!

```
# Assign True or False as appropriate on the lines below!

# False or not True and True
bool_one =

# False and not True or True
bool_two =

# True and not (False or False)
bool_three =

# not not True or False and not True
bool_four =

# False or not (True and True)
bool_five =
```

### 17.3.6 Mix 'n Match

Great work! We're almost done with boolean operators.

### Instructions

Finally, let's try it the other way 'round—we'll provide the expected result (True or False), and you use any combination of boolean operators you want to achieve that result.

Remember, the boolean operators are and, or, and not.

```
# Use boolean expressions as appropriate on the lines below!

# Make me false!
bool_one =

# Make me true!
bool_two =

# Make me false!
bool_three =

# Make me true!
```

```
bool_four =
# Make me true!
bool_five =
```

## 17.4 If, Else and Elif

"If," "else," and "elif" are conditional statements that round out the process by which Python programs make decisions—for example, "if there's Spam, eat it! If not, watch TV."

### 17.4.1 Conditional Statement Syntax

Remember when we showed you that whitespace in Python is significant? If not, write this down: whitespace in Python is significant. If you've learned any JavaScript here on Codecademy, you know that the block of code an if statement executes is bound by curly braces (). In Python, whitespace (tabs or spaces) does this work for us.

Here's an example of if statement syntax in Python:

```
if 8 < 9: print "Eight is less than nine!"
```

if is always followed by an expression, which is followed by a colon (:). The code block (the code to be executed if the expression evaluates to True) is indented four spaces.

This is also true for elif and else (which we'll get to in a moment). The full syntax would look something like this:

```
if 8 < 9: print "I get printed!" elif 8 > 9: print "I don't get printed." else: print
"I also don't get printed!"
```

### Instructions

Here's a piece of code that should look familiar: it's a snippet from the example we showed you in 1.1, "Introduction to Control Flow." If you think the print statement will print to the console, set response equal to 'Y'; otherwise, set response equal to 'N'.

```
response =
answer = "Left"
if answer == "Left":
    print "This is the Verbal Abuse Room, you heap of parrot droppings!"
# Will the above print statement print to the console?
# Set response to 'Y' if you think so, and 'N' if you think not.
```

### 17.4.2 If You're Having...

Let's get some practice in with if statements. Remember, the syntax looks like this:

```
if expression:
    # block line one
    # block line two
    # et cetera
```

## Instructions

Write two if statements in the editor: one that returns True, and a second that returns False. Use any expressions you like! (Don't worry about def and return just yet—we'll cover those in the next unit.)

```
# Write your two if statements below!

def true_function():
    if                      # Fill in your if statement here!
        return              # Make sure this returns True

def false_function():
    if                      # Fill in your if statement here!
        return              # Make sure this returns False
```

### 17.4.3 Else Problems, I Feel Bad for You, Son...

The else statement in Python is the complement to the if statement. While an if statement will return control of the program to the next line of code after the if code block even if there's no else statement, it's considered a good habit to pair each if with an else.

An if/else pair says to Python: "If this expression is true, run this indented code block; otherwise, run this code after the else statement."

Remember when we said "set response equal to 'Y', otherwise, set it to 'N'" in the first exercise of this section? That was a kind of if/else statement!

## Instructions

Complete the else statements to the right. Remember to capitalize your booleans, and note the indentation for each line! It's important.

```
# Write your two else statements below!
answer = "'Tis but a scratch!"

def black_knight():
    if answer == "'Tis but a scratch!":
        return True
    else:                      # Fill in your else statement here!
        return              # Make sure this returns False

def french_soldier():
    if answer == "Go away, or I shall taunt you a second time!":
        return True
    else:                      # Fill in your else statement here!
        return              # Make sure this returns False
```

### 17.4.4 I Got 99 Problems, But a Switch Ain't One

"Elif" is short for "else if." It means exactly what it sounds like: "otherwise, if the following code is true, do this!"

If you're coming from JavaScript, you may know that you have two choices when you have a chain of conditional statements: a bunch of else if statements, or a switch statement. Python simplifies this for you: there's only elif.

#### Instructions

Fill in the elif statement in the editor such that it returns True for the answer "I like Spam!"

You'll want to check whether answer equals the "I like Spam!" string after elif.

```
# Write your two elif statements below!
answer = "I like Spam!"

def feelings_about_spam():
    if answer == "I hate Spam!":
        return False
    elif: # Check the answer variable here!
        return # Make sure this returns True
    else:
        return False
```

## 17.5 Review

Let's take some time to review what we've covered in this unit.

### 17.5.1 The Big If

Really great work! Here's what you've learned in this unit:

- Basics of control flow;
- Comparators (such as `>`, `<`, and `==`);
- Boolean operators (`and`, `or`, and `not`);
- And conditional statements (`if`, `else`, and `elif`).

Let's get to the grand finale.

#### Instructions

Write an if statement in the\_flying\_circus(). It must include:

- `and`, `or`, `or not`;
- `==`, `!=`, `<`, `<=`, `>`, or `>=`;
- an if, elif, AND else statement;
- it must return True when evaluated.

# Chapter 18

## Project: PygLatin

In this project we'll put together all of the Python skills we've learned so far including string manipulation and branching. We'll be building a Pyg Latin translator. (That's Pig Latin for Python Programmers!)

### 18.1 PygLatin Part 1

This project will put together all the Python concepts we've been learning to build an English to PYg Latin Translator.

#### 18.1.1 Break It Down

When you start a big project like this, it's important to take some time to break the problem into individual steps. Then you can tackle (and test) one step at a time rather than trying to write a huge program all at once!

Let's think about the PygLatin problem. Pig Latin is a language where we take the first letter of a word and put it on the end while also adding a vowel sound. So dog becomes "ogday". What are the steps we need to take?

1. Ask the user to input a word in English
2. Check to make sure the user entered a valid word
3. Convert the word from English to Pig Latin
4. Display the translation result

Notice that some of the steps can themselves be broken down into individual steps. For example, we'll want to think through the algorithm for step #3 before we start coding.

A little bit of time invested in thinking through the decomposition of and algorithms for your program can save you a LOT of frustration down the road!

Get a piece of paper and work out an algorithm for step #3 of the project.

## Instructions

All set? When you're ready to get coding, click the Run button to continue. Since we took the time to write out the steps for our solution, you'll know what's coming next!

### 18.1.2 Ahoy! (or Should I Say Ahoyay!)

This project will be a workout, so let's warm up by printing a welcome message for our translator users.

## Instructions

Use Python to print "Welcome to the English to Pig Latin translator!" to the console.

### 18.1.3 Input!

If we're going to translate an English word into Pig Latin, the first thing we're going to need is a word.

Python can ask for input from the user with the `raw_input` command. For example, if you type:

```
name = raw_input("What's your name? ")
```

Python will print:

```
What's your name? >
```

Once you type something and hit Enter, Python will store whatever you typed in the `name` variable.

## Instructions

Below your existing code, prompt the user to enter a word, and store it in a variable called `original`. The prompt message can be whatever you want!

### 18.1.4 Check Yourself!

Whenever you ask a user for input, it's a good idea to check the result that you get before you use it in your program.

In this case, we want to make sure that the input is something we can reliably translate into Pig Latin. That is, we want to make sure we got something resembling an English word.

The first thing we can check is that there are characters in our string. Let's check to make sure the string isn't empty. How can we check to make sure that our `original` variable isn't empty?

### Instructions

Write an if statement that checks to see if the string is not empty.

- If the string is not empty, print the user's word.
- Otherwise (else), print "empty" if the string is empty.

Make sure you thoroughly test your code. You'll want to make sure you run it multiple times and test both an empty string and a string with characters. When you are confident that your code works, continue on to the next exercise.

### 18.1.5 Check Yourself... Some More

Great! Now we know we have a non-empty string. Let's be even more thorough, though. After all, a user might try to be tricky and enter something like 8675309 which, since it is numbers not letters, would not make sense in PygLatin.

Let's add to our input validation and make sure that the word the user enters contains only alphabetical characters. You can use the `isalpha()` function to check this. To check if the string "J123" within a variable `x` is alphabetical, you could do:

```
x = "J123"  
x.isalpha() # False
```

### Instructions

Add onto your if condition to check that the word is also composed of all alphabetical characters. You should end up with a single if-else statement that makes sure original is a non-empty alphabetical string.

Make sure to check both an all-alphabetical string and one with letters and numbers!

### 18.1.6 Pop Quiz!

Teachers sometimes give pop quizzes to make sure their students are mastering current material. Similarly, when you're working on a big project, you should periodically take time to really put your program to the test.

When you finish one part of your program, it's important to test it multiple times, using a variety of inputs.

### Instructions

Take some time right now to test your current code. Make sure you try some inputs that should pass the word test and some that should fail. Enter some strings that contain non-alphabetical characters and an empty string.

When you're convinced your code is ready to go, click Save & Submit Code to move on to the next step!

## 18.2 PygLatin Part 2

Let's continue to build our PygLatin translator!

### 18.2.1 Ay B C

Now that we know we have a good word, we can get ready to start translating to Pig Latin! Let's quickly review the rules for translation:

1. If the original word starts with a vowel, you append the suffix 'ay' to the end of the word.

Example: anaconda → anacondaay

2. If the original word starts with a consonant, you move the first letter of the word to the end, and then append the suffix 'ay'.

Example: python → ythonpay

Let's create a variable to hold our translation suffix.

#### Instructions

Create a variable named pyg and set it equal to the suffix 'ay'.

### 18.2.2 Word Up

Since the translation rules depend on the first letter of the word we are translating, we need to grab it so we can check if it is a consonant or a vowel. To simplify things, we'll also go ahead and make sure that all the letters in our word are lowercase to make things a little easier.

To convert a word to all lowercase letters, we can use lower(). For example, to convert the string value "FOO" in a variable x to lowercase, we could do the following:

```
x = "FOO"  
x.lower() # "foo"
```

You'll want to think about where to put the code that you write in this step. It really only makes sense to do these steps if you already know that you have a useable word, so make sure you put this code inside the if/else block.

#### Instructions

- Convert the variable original to all lowercase letters. Store the result in a variable named word.
- Create a new variable called first that holds the first letter of word.

```
pyg = 'ay'

original = raw_input('Enter a word: ')

if len(original) > 0 and original.isalpha():
    print original
else:
    print 'empty'
```

### 18.2.3 E-I-E-I-O

Now that we have access to the first letter of our word, we need to check to see if it is a vowel or a consonant. Since there are way fewer vowels than consonants, it is easier to explicitly check for a vowel. (The vowels in English are: a, e, i, o and u.)

Again, it only makes sense to do this check if we already know that we have a "good" word (one that isn't empty and is all alphabetical characters). You have a couple of options on how to organize your code.

You could add onto your existing if condition to check that something is a good word and starts with a vowel. But then you'd have to add an elif to check if the word is a good word and starts with a consonant.

In this case, it makes more sense to nest the new if block inside the if part of the existing if/else. This means that the whole if/else that you will create here goes inside the if part of your existing if/else block.

A nested if/else looks like this:

```
if condition:
    if other_condition:
        # Do something
    else:
        # Do something else!
else:
    # Do yet another thing
```

#### Instructions

Add a new if/else block nested inside of your existing one.

The new if should check to see if the first letter of word is a vowel. If it is, your code should print "vowel". If the first letter of word is not a vowel, your code should print "consonant".

You can remove the print original line from your code.

### 18.2.4 I'd Like to Buy a Vowel

Now that we have the logical structure in place to check for a vowel, let's put in the code to do the actual translation.

Remember that if a word starts with a vowel, you translate it to Pig Latin by appending our suffix ("ay") to the end of the word.

We'll want to make sure we output the result of the translation so we can check our work.

## Instructions

Create a new variable called `new_word` that contains the result of appending 'ay' to the end of word. (Remember that this suffix is stored in the variable `pyg`.)

Make sure you are only doing this in the case where the word starts with a vowel. Don't worry about the consonants just yet, we'll get to them in the next step!

After the translation, print the new word so you can check your work. (This should replace the print 'vowel' bit of your code.) Make sure to test your code with a word that starts with a vowel!

### 18.2.5 Almost Oneday!

Now that we have the vowel case working, all that's left is to tackle the consonant case (our inner else, since a letter can only be a consonant or a vowel).

The rule for translation here is a little trickier. You have to remove the first letter from the word, move it to the end, and then append the 'ay' suffix.

The most appropriate way to get the remainder of the string after removing the first letter is to use slicing. If you have a string `s`, you can get the "slice" of `s` from `i` to `j` using `s[i:j]`. This gives you the characters from position `i` to `j`.

For example, if `s = "foo"`, then `s[0:2]` gives you "fo". Think about how to use this technique to get the rest of the string minus the first character.

## Instructions

Inside the else part of your if/else block that checks the first letter of the word, set the `new_word` variable equal to the translation result for a word that starts with a consonant.

Replace the print 'consonant' bit with print `new_word`. Make sure to test your code with a word that starts with a consonant!

### 18.2.6 Testing, Testing, is This Thing On?

Yay! You should have a fully functioning Pig Latin translator. Just to make sure everything is working smoothly, make sure you test your code thoroughly.

You'll also want to go back through and take out any print statements that you were using to help debug intermediate steps of your code. While you're cleaning things up—now might be a good time to add some comments too! Making sure your code is clean, commented, and fully functional is just as important as writing it in the first place.

## Instructions

When you're sure your translator is working just the way you want it, click Save & Submit Code to finish this project.

```
pyg = 'ay'
original = raw_input('Enter a word: ')
```

```
if len(original) > 0 and original.isalpha():
    print original
else:
    print 'empty'
```



# Chapter 19

## Functions

A function is a reusable section of code written to perform a specific task in a program. We gave you a taste of functions in Unit 3; here, you'll learn how to create your own.

### 19.1 Introduction to Functions

It's... functions!

#### 19.1.1 Documentation: a PSA

Codecademy is a great educational tool—you can think of it as an online classroom for learning a programming language. It's important to remember, however, that the emphasis should be on your approach to problems and learning to think like a programmer, and not on memorizing every single method or nuance of syntax to be found in a given programming language.

Think of it this way: if you're going to Germany, you wouldn't say to yourself, "Hey, I took a year of German in college, I don't need my dictionary or translation app!" You would totally bring those things to help remind you of vocabulary and syntax in case you were to get stuck; even professional translators keep dictionaries and grammar guides handy for unusual words and tricky constructions.

Much like professional translators, professional programmers refer to documentation when they're not clear on best practices, forget how a certain method works, or need to look up syntax. Python's documentation can be found [here](#). We encourage you to read through it on your own!

#### Instructions

Part of Python's documentation are the PEPs, or "Python Enhancement Proposals." PEP 20, "The Zen of Python," is something of an easter egg hidden in the interpreter (you may have caught us quoting from it in instructional text or hints).

Type `import this` in the editor to see what happens (we'll learn about the `import` keyword later in this lesson). Scroll to read all the text output to the console.

## What Good are Functions?

A function is a reusable section of code written to perform a specific task in a program. You might be wondering why you need to separate your code into functions, rather than just writing everything out in one giant block. You might have guessed the answer(s) already, but here are some of the big ones:

1. If something goes wrong in your code, it's much easier to find and fix bugs if you've organized your program well. Assigning specific tasks to separate functions helps with this organization.
2. By assigning specific tasks to separate functions (an idea computer scientists call separation of concerns), you make your program less redundant and your code more reusable—not only can you repeatedly use the same function in a single program without rewriting it each time, but you can even use that function in another program.
3. When we learn more about objects, you'll find out there are a lot of interesting things we can do with functions that belong to those objects (called methods).

## Instructions

Check out the code in the editor. If you completed the Tip Calculator project, you'll remember going through and calculating tax and tip in one chunk of program. Here you can see we've defined two functions: tax to calculate the tax on a bill, and tip to compute the tip.

See how much of the code you understand at first glance (we'll explain it all soon). When you're ready, click Save & Submit Code to continue.

```
def tax(bill):
    """ Adds 8% tax to a restaurant bill """
    bill *= 1.08
    print "With tax: %f" % bill
    return bill

def tip(bill):
    """ Adds 15% tip to a restaurant bill """
    bill *= 1.15
    print "With tip: %f" % bill
    return bill

meal_cost = 100
meal_with_tax = tax(meal_cost)
meal_with_tip = tip(meal_with_tax)
```

### 19.1.2 Ample Examples

All right! Now that you know what functions are good for, it's time to see one in action. (Hopefully it's familiar to you!)

You'll see in the editor a variable, length, assigned to the result or output of a function, len(). This function is built into Python (we'll see how to define our own functions in a moment—you got a sneak peek in the last exercise). Here's how the code to the right works:

1. It first evaluates the right hand-side of the assignment. To evaluate the function call, it looks at the long string in parentheses.
2. Since it's a string literal, it doesn't need to be evaluated (that is, it's not an expression that Python has to figure out, like  $1 + 1$ ). So, this string is fed directly as an input to the `len()` function.
3. `len()` accepts this input and returns (or outputs) an integer representing the length of the literal string (in this case 45, including spaces and punctuation).

We've tossed in a `print` statement to show you that the result is, in fact, 45.

### Instructions

Replace the string literal inside `len`'s parentheses with a stringified version of the number 45. That is, don't just type "45", but actually use a string function to turn 45 into a string.

## 19.2 Function Syntax

Functions are great for automating tasks you're going to use over and over, or for making it easy to refer to large sections of your program using one specific name. But how are they put together?

### 19.2.1 Function Junction

Functions are defined using the keyword `def` (short for "define"). Functions have three parts:

The header, which includes the `def` keyword, the name of the function, any arguments the function takes inside parentheses `()`, and a colon `:`. (We'll get to arguments in the next exercise); An optional docstring, which is a triple-quoted, multi-line comment that briefly explains what the function does; And the body, which is the code block that describes the procedures the function carries out. The body is indented (much like for `if`, `elif`, and `else` statements).

Here's an example of what the syntax would look like for a simple function, `ni_sayer`, that just prints "Ni!" to the console:

```
def ni_sayer():
    """Prints 'Ni!' to the console."""
    print "Ni!"
```

(This is not a very good example of a docstring—ideally, the docstring should explain something that isn't blindingly obvious.)

### Instructions

Time for you to make your own simple function. Go ahead and create a function, `spam`, that prints the string "Eggs!" to the console. Be sure to use the capitalization

and punctuation shown! Don't forget to include a docstring of your own choosing (just remember to enclose it in triple quotes).

```
# Define your spam function starting on line 5. You
# can leave the code on line 11 alone for now—we'll
# explain it soon!

# Define the spam function above this line.
spam()
```

## 19.2.2 Call and Response

Defining a function is all well and good, but it's not much use to you unless you call it. That's the code you saw on line 11 in the previous exercise: when Python saw `spam()`, it understood that to mean: "Look for the function called `spam` and execute the code inside it." The parentheses after the function name let Python know that `spam` is the name of a function.

### Instructions

We've set up a function, `square`, in the editor to the right. Call it on the number 10 on line 9 to see what it does! (That is, put 10 between the parentheses of `square()`.)

```
def square(n):
    """Returns the square of a number."""
    squared = n**2
    print "%d squared is %d." % (n, squared)
    return squared

# Call the square function on line 9! Make sure to
# include the number 10 between the parentheses.
```

## 19.2.3 No One Ever Does

If a function takes arguments, we say it accepts or expects those arguments. For instance, if the function `no_one` takes a single argument, "The Spanish Inquisition", we would say that `no_one` expects "The Spanish Inquisition".

Ha!

To be precise, the argument is the piece of code you actually put between the function's parentheses when you call it, and the parameter is the name you put between the function's parentheses when you define it. For instance, when we defined `square` in the previous exercise, we gave it the parameter `n` (for "number"), but passed it the argument 10 when we called it.

You can think of parameters as nicknames the function definition gives to arguments, since it doesn't know ahead of time exactly what argument it's going to get.

The syntax for a function that just prints out the argument it expects would look something like the below.

Function definition:

```
def no_one(sentence):
    print sentence
```

Calling the function:

```
no_one("The Spanish Inquisition")
```

And the console would display:

```
"The Spanish Inquisition"
```

which is the value the parameter sentence takes on when you call no\_one and pass the argument "The Spanish Inquisition".

### Instructions

Make sense? Good! Check out the function in the editor, power. It should take two arguments, a base and an exponent, and raise the first to the power of the second. It's currently broken, however, because its parameters are missing.

Replace the \_\_\_\_s with the parameters base and exponent and call power on a base of 37 and a power of 4.

```
def power(___, ___): # Add your parameters here!
    result = base**exponent
    print "%d to the power of %d is %d." % (base, exponent, result)

power() # Add your arguments here!
```

## 19.2.4 Splat!

Speaking of not knowing what to expect: your functions not only don't know what arguments they're going to get ahead of time, but occasionally, they don't even know how many arguments there will be.

Let's say you have a function, favorite\_actor, that prints out the argument it receives from the user. It might look something like this:

```
def favorite_actor(name):
    """ Prints out your favorite actor """
    print "Favorite actor is: " , name
```

This is great for just one actor, but what if you want to print out the user's favorite actors, without knowing how many names the user will put in?

The solution: splat arguments. Splat arguments are an arbitrary number of arguments, and you use them by preceding your parameter with a \*. This says to Python, "Hey man, I don't know how many arguments there are about to be, but it

could be more than one." The convention is to use \*args, but you can use just about any name you like with a \* before it.

### Instructions

Replace the \_\_\_\_\_s in the function to the right with the appropriate code and click Save & Submit Code to see who your favorite actors are (or, at least, who they should be). Remember to include a \* before your parameter (between the parentheses on line 1).

You don't need the \* when you replace the \_\_\_\_\_ on line 3—just the parameter name you chose on line 1.

```
def favorite_actors(____):
    """ Prints out your favorite actorS (plural!) """
    print "Your favorite actors are:" , _____

favorite_actors("Michael Palin", "John Cleese", "Graham Chapman")
```

### 19.2.5 Functions Calling Functions

So far, we've seen functions that can print text to the console or do simple arithmetic, but functions can be much more powerful than that. For example, it's completely permissible for a function to call another function.

### Instructions

Check out the two functions in the editor: one\_good\_turn and deserves\_another. The first function adds 1 to number it gets as an argument, and the second adds 2.

In the body of deserves\_another, change the function so that it always adds 2 to the output of one\_good\_turn.

```
def one_good_turn(n):
    return n + 1

def deserves_another(n):
    return n + 2
```

### 19.2.6 Practice Makes Perfect

You never really know how to do something until you do it yourself. We're taking the training wheels off now: time for you to define and call functions all on your lonesome. No examples, no code in the editor. Not even a hint! (Well, okay, one hint.)

And while we're thinking of it: you can always practice your functions (or any Python code) in either the scratch pad. Experimenting in the Labs is a great way to reinforce what you've learned in Codecademy lessons.

### Instructions

1. Define a function called cube that takes a number and returns the cube of that number. (Cubing a number is the same as raising it to the third power).
2. Define a second function called by\_three that takes one number as an argument. If that number is evenly divisible by 3, by\_three should call cube on that number. If the number is not evenly divisible by 3, cube should return False.

So, for example, by\_three should take 9, determine it's evenly divisible by 3, and pass it to cube, who returns 729 (the result of  $9^{**}3$ ). If by\_three gets 4, however, it should return False and leave it at that.

## 19.3 Importing Modules

Remember that scene in The Matrix where Neo learns kung fu by downloading a program directly into his brain? Importing modules: just like that.

### 19.3.1 I Know Kung Fu

Remember import this from the first exercise in this course? That was an example of importing a module. A module is a file that contains definitions—including variables and functions—that you can use. It would clutter up the interpreter to keep all these variables and functions around all the time, so you just import the module you want when you need something from it.

### Instructions

Before we try any fancy importing, let's see what Python already knows about square roots. On line 3 in the editor, ask Python to

```
print sqrt(25)
```

which we would expect to equal five.

### 19.3.2 Generic Imports

Did you see that? Python said: "NameError: name 'sqrt' is not defined." Python doesn't know what square roots are—yet.

There is a Python module named math that includes a number of useful variables and functions, and (as you've probably guessed) sqrt() is one of those functions. In order to get to it, all you need is the import keyword. When you simply import a module this way, it's called a generic import.

## Instructions

You'll need to do two things here:

1. Type import math on line 2 in the editor, and
2. Insert math. (that's math, followed by a period) before sqrt(). This tells Python not only to import math, but to get the sqrt() function from within math.

Once you've done this, hit "run" to see what Python now knows.

### 19.3.3 Function Imports

Nice work! Now Python knows how to take the square root of a number (as well as how to do everything contained in the math module).

Importing the entire math module is kind of annoying for two reasons, though: first, we really only want the sqrt function, and second, we have to remember to type math.sqrt() any time we want to retrieve that function from the math module.

Thankfully, it's possible to import only certain variables or functions from a given module. Pulling in just a single function from a module is called a function import, and it's done using the from keyword, like so:

```
from module import function
```

where "module" and "function" are replaced by the names of the module and function you want. The best part is, now you only have to type sqrt() to get the square root of a number—no more math.sqrt()!

## Instructions

Let's import the sqrt function from math again, only this time, let's only get the sqrt function. (You don't need the () after sqrt in the from math import sqrt bit.)

### 19.3.4 Universal Imports

Great! We've found a way to handpick the variables and functions we want from the modules that contain them.

What if we want a large selection (or all) of the variables and functions available in a module? We can import module, but there's another option.

When you import math, you're basically saying: "Bring the Math Box to my apartment so I can use all the cool stuff in it." Whenever you want a tool in math, you have to go to the box and pull out the thing you want (which is why you have to type math.name for everything—even though the box is in your apartment, all the cool stuff you want is still in that box).

When you choose from math import sqrt, you're saying: "Bring me only the square root tool from the Math Box, and don't bring the Math Box to my apartment." This means you can use sqrt without reference to math, but if you want

anything else from math, you have to import it separately, since the whole Math Box isn't in your apartment for you to dig through.

The third option is to say: "Don't bring the Math Box to my apartment, but bring me absolutely every tool in it." This gives you the advantage of having a wide variety of tools, and since you have them in your apartment and they're not all still stuck in the Math Box, you don't have to constantly type `math.name` to get what you want.

The syntax for this is:

```
from module import *
```

If you're familiar with CSS, you've seen that `*` can stand for "every selector," and it serves a similar function in Python: it stands in for every variable and function name in a module.

### Instructions

Use the power of `from module import *` to import everything from the `math` module on line 3 of the editor.

#### 19.3.5 Here Be Dragons

Here's something we've learned in life (and not just from programming): just because you can do something doesn't mean that you should.

Universal imports may look great on the surface, but they're not a good idea for one very important reason: they can fill your program with a ton of variable and function names, but without the safety of those names still being associated with the module(s) they came from.

If you have a function of your very own named `sqrt` and you import `math`, your function is safe: there is your `sqrt` and there is `math.sqrt`, and ne'er the twain shall meet. If you do `from math import *`, however, you have a problem: namely, two different functions with the exact same name.

Even if your own definitions don't directly conflict with names from imported modules, if you import `*` from several modules at once, there won't be any way for you to figure out which variable or function came from where. It'd be like having someone dump a ton of random stuff from a bunch of different boxes in your apartment, then throwing the boxes away so you can't even see where the stuff came from.

For these reasons, it's best to stick with either `import module` and suffer the inconvenience of having to type `module.name`, or just import specific variables and functions from various modules as needed.

### Instructions

Line 1 and line 3 in the editor should look familiar to you; line 2 is an example of some of the cool stuff you'll be able to do in future lessons. In a nutshell, this code will show you everything available in the `math` module.

Click Save & Submit Code to check it out (you'll see sqrt, along with some other useful things like pi, factorial, and trigonometric functions). Feel free to spend a few minutes playing around with them!

## 19.4 Built-In Functions

Just because you didn't define a function for Python doesn't mean it doesn't know it. Python has a number of built-in functions you can use to write all sorts of powerful, elegant programs.

### 19.4.1 On Beyond Strings

Now that you understand what functions are and how to import modules, it's worth showing you some of the cool functions that are built in to Python (no modules required!).

You already know about some of the built-in functions we've used on (or to create) strings, such as .upper(), .lower(), str(), and len(). These are great for doing work with strings, but what about something a little more analytic?

#### Instructions

Check out the code in the editor. What do you think it'll do? Click Save & Submit Code when you think you have an idea.

```
def biggest_number(*args):
    print max(args)
    return max(args)

def smallest_number(*args):
    print min(args)
    return min(args)

def distance_from_zero(arg):
    print abs(arg)
    return abs(arg)

biggest_number(-10, -5, 5, 10)
smallest_number(-10, -5, 5, 10)
distance_from_zero(-10)
```

### 19.4.2 max()

The max() function takes any number of arguments and returns the largest one. ("Largest" can have odd definitions here, so it's best to use max() on things like integers and floats, where the results are straightforward, and not on other objects, like strings.)

For example, max(1,2,3) will return 3 (the largest number in the set of arguments).

### Instructions

Try out the `max()` function on line 3 of the editor. You can provide any number of integer or float arguments to `max()`.

```
# Set maximum to the max value of any set of numbers on line 3!
maximum =
print maximum
```

### 19.4.3 min()

As you might imagine, `min()` does the opposite of `max()`—given a series of arguments, it returns the smallest one.

### Instructions

Go ahead and set `minimum` equal to the `min()` of any set of integers or floats you'd like.

```
# Set minimum to the min value of any set of numbers on line 3!
minimum =
print minimum
```

### 19.4.4 abs()

The `abs()` function returns the absolute value of the number it takes as an argument—that is, that number's distance from 0 on an imagined number line, regardless of whether it's positive or negative. For instance, 3 and -3 are both equally far from 0, and thus have the same absolute value: 3. The `abs()` function always returns a positive value, and unlike `max()` and `min()`, it can only take a single number.

### Instructions

Set `absolute` equal to the absolute value of -42 on line 3. (This may seem basic, but bear with us—you'll see the value of the exercise soon enough.)

```
# Set absolute to the absolute value of -42 on line 3!
absolute =
print absolute
```

### 19.4.5 type()

Finally, the `type()` function does something very interesting: it returns the type of the data it receives as an argument. If you ask Python to do the following:

```
print type(42)
print type(4.2)
print type('spam')
print type({'Name': 'John Cleese'})
print type((1,2))
```

Python will output:

```
<type 'int'>
<type 'float'>
<type 'unicode'>
<type 'dict'>
<type 'tuple'>
```

(The 'unicode' type is a special type of string.)

You're already familiar with integers, floats, and strings; you'll learn about dictionaries and tuples in later lessons.

### Instructions

Have Python print out the type of an int, a float, and a unicode string in the editor. You can pick any values on which to call type(), so long as they produce one of each.

## 19.5 Review

Let's take some time to go over what we've covered in this unit.

### 19.5.1 Review: Functions

Up until now, the review section of each unit has been a single exercise. As you progress through the Python courses, you'll see longer review sections (starting with this one)—this is to ensure you have ample practice as you're exposed to more (and more complex) aspects of the language.

Okay! Let's review functions. Again, training wheels are off, but feel free to take a peek at earlier exercises if you need a refresher.

### Instructions

Write a function, shut\_down, that takes one parameter (you can use anything you like; in this case, we'd use s for string). The shut\_down function should return "Shutting down..." when it gets "Yes", "yes", or "YES" as an argument, and "Shutdown aborted!" when it gets "No", "no", or "NO".

If it gets anything other than those inputs, the function should return "Sorry, I didn't understand you."

### 19.5.2 Review: Modules

Good work! Now let's see what you remember about importing modules (and, specifically, what's available in the math module). Instructions

Import the math module in whatever way you prefer. Call its sqrt function on the number 13689 and print that value to the console.

### 19.5.3 Review: Built-In Functions

Perfect! Last but not least, let's review some of the built-in functions you've learned about in this lesson.

#### Instructions

This is a two-parter: first, define a function, distance\_from\_zero, with one parameter (choose any parameter name you like).

Second, have that function do the following:

1. Check the type of the input it receives.
2. If the type is int or float, the function should return the absolute value of the function input.
3. If the type is any other type, the function should return "Not an integer or float!"



# Chapter 20

## Taking a Vacation

Hard day at work? Rough day at school? Take a load off with a programming vacation!

### 20.1 A Review of Function Creation

A quick review of writing functions.

#### 20.1.1 Before We Begin

Before we start the lesson, we are just going to do a quick review of functions in Python. Remember, functions are callable blocks of code that we can use over and over again. Using functions saves us time, lines of code, and confusion when writing long programs.

##### Instructions

For a warm up, write a function called `answer` that takes no input and returns the value 42.

#### 20.1.2 Finding Your Identity

Unfortunately for us, 42 is not the answer to every question in the universe.

##### Instructions

Write the function `identity` that takes the input `x` and returns `x`. You do not need to call the function.

#### 20.1.3 Call Me Maybe?

Remember that you can call functions to make computationally challenging tasks much easier.

### Instructions

Call the function `cube` with an input of 27 on line 4. Print the result to the console after you call the function.

```
def cube(x):
    return x**3
```

### 20.1.4 Function and Control Flow

Remember that functions often have to react differently depending on the input they receive.

### Instructions

Write a function called `is_even` that takes one input, `x`, and returns the string "yep" if the input is even and "nope" otherwise. You do not need to call the function.

### 20.1.5 Problem Solvers

This final review exercise will involve applying functions to a real life problem.

Let's try writing a function to solve a traditional math question.

For this exercise, you'll need to import the `math` module discussed in Unit 4. This can be achieved by typing `import math` at the top of your program.

### Instructions

Write a function called `area_of_circle` that takes `radius` as input and returns the area of a circle. The area of a circle is equal to `pi` times the radius squared. (Use the `math.pi` in order to represent Pi.)

## 20.2 Planes, Hotels and Automobiles

It's vacation time! Let's see how much this trip will cost you!

### 20.2.1 Planning Your Trip

When planning a vacation, it is very important to know exactly how much you are going to spend. With the aid of programming, this task becomes very easy.

We will break your trip down into 3 main costs and then put them together in one big function at the end.

### Instructions

First, write a function called `hotel_cost` that takes the variable `nights` as input. The function should return how much you have to pay if the hotel costs 140 dollars for every night that you stay.

### 20.2.2 Getting There

To get to your location, you are going to need to take a plane ride.

#### Instructions

Below your existing code, write a function called `plane_ride_cost` that takes a string, `city`, as input. The function should return a different price depending on the location. Below are the valid destinations and their corresponding round-trip prices.

- "Charlotte": 183
- "Tampa": 220
- "Pittsburgh": 222
- "Los Angeles": 475

### 20.2.3 Transportation

Now when you arrive at your destination, you are going to need a rental car in order for you to get around. Luckily for you, the rental car company you use gives discounts depending on how many days you rent the car.

#### Instructions

Below your existing code, write a function called `rental_car_cost` that takes `days` as input and returns the cost for renting a car for said number of days. The cost must abide by the following conditions:

1. Every day you rent the car is \$40.
2. If you rent the car for 3 or more days, you get \$20 off your total.
3. If you rent the car for 7 or more days, you get \$50 off your total. (This does not stack with the 20 dollars you get for renting the car over 3 days.)

### 20.2.4 Pull it Together

Great! Now that you've got your 3 main costs figured out, it's time to put them together in order to find the total cost of your trip.

#### Instructions

Below your existing code, write a function called `trip_cost` that takes two inputs, `city` and `days`. `city` should be the city that you are going to visit and `days` should be the number of days that you are staying.

Have your function return the sum of the `rental_car_cost`, `hotel_cost`, and `plane_ride_cost` functions with their respective inputs.

### 20.2.5 Hey, You Never Know!

You can't expect to only spend money on the plane ride, hotel, and rental car when going on a vacation. While the above items will cover the majority of your expenses, there also needs to be room for additional costs like fancy food or souvenirs.

The amount of money that you wish to spend on additional luxuries is completely up to you.

#### Instructions

Make it so that your trip\_cost function takes a third parameter, spending\_money. Just modify the trip\_cost function to do just as it did before, except add the spending money to the total that it returns.

### 20.2.6 Plan Your Trip!

#### Instructions

Now that you have it all together, print out the cost of a trip to "Los Angeles" for 5 days with an extra 600 dollars of spending money.

## 20.3 Return to Base

Welcome back home! It's time to see how much you owe.

### 20.3.1 Coming Back Home

Welcome back home! It looks like you've had an amazing vacation.

Unfortunately for you, one too many drinks and a bit too much time at the casino have put you a bit over-budget.

#### Instructions

Go ahead and print out how far over budget you went if you spent 2734.23 on your trip.

```
def hotel_cost(nights):
    return nights * 140

def plane_ride_cost(city):
    if city == "Charlotte":
        return 183
    elif city == "Tampa":
        return 220
    elif city == "Pittsburgh":
        return 222
    else:
        return 475

def rental_car_cost(days):
    cost = days * 40
    if days >= 7:
        cost = cost - 50
```

```

    elif days >= 3:
        cost = cost - 20
    return cost

def trip_cost(city, days, spending_money):
    return rental_car_cost(days) + hotel_cost(days) + plane_ride_cost(city)

# You were planning on taking a trip to LA
# for five days with $600 of spending money.
print trip_cost("Los Angeles", 5, 600)

```

### 20.3.2 Gotta Give Me Some Credit

Yikes! That is a bit more than you intended to spend. You put your hotel bill on your credit card and now you don't have the money to pay for it up front.

It looks like you're going to have to break it up into monthly payments. Programming might aid you in handling this mess.

#### Instructions

Call the hotel\_cost function with 5 days as the input and store the result in a variable called bill.

```

def hotel_cost(nights):
    return nights * 140

```

### 20.3.3 At a Bare Minimum

First, we need to know what the minimum payment is that you can make each month.

Note that the minimum payment you can make is 2% of your total balance. So the minimum payment you can make with a rate of 2% and a balance of 1,000 would be 20 dollars ( $1000 * 0.02$  equals 20.)

#### Instructions

Below your existing code, write a function called get\_min that takes balance and rate as inputs and returns the minimum payment that you can make with your total balance. balance and rate should both be given as numbers and not percentages, so 2% is 0.02.

Go ahead and print out the minimum payment of your bills with a 2% rate, as calculated by your get\_min function.

### 20.3.4 Something of Interest

All credit cards charge interest in proportion to your current balance. It's important that we know exactly how much we still owe after making a payment.

### Instructions

Write a function called `add_monthly_interest` that takes the input `balance` and returns your balance with interest added to it.

Assume your interest is 15%. This means that we need to add on 15% to whatever balance is passed in!

Note that 1/12th of your interest multiplied by your balance is equal to the amount of interest you pay each month (or at least close, anyway). So, `addmonthlyinterest(100)` should return 101.25.

### 20.3.5 Paying Up

Now it's time for you to make a function that computes how much you still owe after every monthly payment.

You'll create the function `make_payment` as described below. The function should return how much you still owe after you make an arbitrary payment at the beginning of the month.

Note: you calculate how much you still owe by subtracting your payment from your total balance and then adding interest using your `add_monthly_interest` function.

### Instructions

Finish the function `make_payment` that takes the inputs `payment` and `balance`.

Have the function return the string "You still owe: `x`", with `x` being the amount that you still owe. Remember to add interest to the final amount!

```
def hotel_cost(nights):
    return nights * 140

bill = hotel_cost(5)

def add_monthly_interest(balance):
    return balance * (1 + (0.15 / 12))

def make_payment(payment, balance):

    return "You still owe: " + str(new_balance)
```

### 20.3.6 Run It

#### Instructions

To finish up, make a \$100 first payment on your hotel bill using your `make_payment` function. Remember that you stayed at the hotel for 5 days.

Finally, print that result to the console!

```
def hotel_cost(nights):
    return nights * 140
```

```
bill = hotel_cost(5)

def add_monthly_interest(balance):
    return balance * (1 + (0.15 / 12))

def make_payment(payment, balance):

    return "You still owe: " + str(new_balance)
```



# Chapter 21

## Lists and Dictionaries

Lists and dictionaries are powerful tools you can use to store, organize, and manipulate all kinds of information.

### 21.1 Lists

A Python list is just a sequence of pieces of information. You can use lists to store strings, numbers, and more!

#### 21.1.1 Introduction to Lists

Lists are a datatype you can use to store a collection of different pieces of information as a sequence under a single variable name. (Datatypes you've already learned about include strings, numbers, and booleans.)

You can assign items to a list with an expression of the form `list_name = [item_1, item_2]`, with the items in between brackets. A list can also be empty: `empty_list = []`.

#### Instructions

The list `zoo_animals` has three items (check them out on line 1). Go ahead and add a fourth! Just enter the name of your favorite animal (as a "string") on line 1, after the final comma but before the closing `]`.

```
zoo_animals = ["pangolin", "cassowary", "sloth", ]  
# One animal is missing!  
  
if len(zoo_animals) > 3:  
    print "The first animal at the zoo is the " + zoo_animals[0]  
    print "The second animal at the zoo is the " + zoo_animals[1]  
    print "The third animal at the zoo is the " + zoo_animals[2]  
    print "The fourth animal at the zoo is the " + str(zoo_animals[3])
```

### 21.1.2 Access by Index

You can access an individual item on the list by its index. An index is like an address that identifies the item's place in the list. The index appears directly after the list name, in between brackets, like this: `list_name[index]`.

List indices begin with 0, not 1! You access the first item in a list like this: `list_name[0]`. The second item in a list is at index 1: `list_name[1]`. Computer scientists love to start counting from zero.

#### Instructions

Write a statement that prints the result of adding the second and fourth items of the list. Make sure to access the list by index!

```
numbers = [5, 6, 7, 8]
print "Adding the numbers at indices 0 and 2..."
print numbers[0] + numbers[2]
print "Adding the numbers at indices 1 and 3..."
# Your code here!
```

### 21.1.3 New Neighbors

A list index behaves like any other variable name! It can be used to access as well as assign values.

You saw how to access a list index like this:

```
zoo_animals[0]
# Gets the value "pangolin"
```

You can see how assignment works on line 5:

```
zoo_animals[2] = "hyena"
# Changes "sloth" to "hyena"
```

#### Instructions

Write an assignment statement that will replace the item that currently holds the value "tiger" with another animal (as a string). It can be any animal you like.

```
zoo_animals = ["pangolin", "cassowary", "sloth", "tiger"]
# Last night our zoo's sloth brutally attacked the poor tiger and ate it whole.

# The ferocious sloth has been replaced by a friendly hyena.
zoo_animals[2] = "hyena"

# What shall fill the void left by our dear departed tiger?
# Your code here!
```

## 21.2 List Capabilities and Functions

Great work with lists! Now let's learn about some of the cool things we can do with them.

### 21.2.1 Late Arrivals & List Length

A list doesn't have to have a fixed length—you can add items to the end of a list any time you like! In Python, we say lists are mutable: that is, they can be changed.

You can add items to lists with the built-in list function `append()`, like this:

```
list_name.append(item)
```

Check it out: we've appended a string to `suitcase` on line 2.

You can get the number of items in a list with the `len()` function (short for "length"), like so:

```
len(list_name)
```

### Instructions

Append three more items to the `suitcase` list. (Maybe bring a bathing suit?) Then, set `list_length` equal to the length of `suitcase`.

```
suitcase = []
suitcase.append("sunglasses")

# Your code here!

list_length = # Set this to the length of suitcase

print "There are %d items in the suitcase." % list_length
print suitcase
```

### 21.2.2 List Slicing

If you only want a small part of a list, that portion can be accessed using a special notation in the index brackets. `list_name[a:b]` will return a portion of `list_name` starting with the index `a` and ending before the index `b`.

If you tell Python `my_list = [0, 1, 2, 3]`, then `my_list[1:3]` will return the list `[1, 2]`, leaving the original list unchanged! Check it out:

```
my_list = [0, 1, 2, 3]
my_slice = my_list[1:3]
print my_list
# Prints [0, 1, 2, 3]
print my_slice
# Prints [1, 2]
```

### Instructions

Use list slicing to make a list called first that's composed of just the first two items from suitcase, a list called middle containing only the two middle items from suitcase, and a list called last made up only of the last two items from suitcase.

```
suitcase = ["sunglasses", "hat", "passport", "laptop", "suit", "shoes"]

first =      # The first two items
middle =     # Third and fourth items
last =       # The last two items
```

### 21.2.3 Slicing Lists and Strings

You can slice a string exactly like a list! In fact, you can think of strings as lists of characters: each character is a sequential item in the list, starting from index 0.

If your list slice includes the very first or last item in a list (or a string), the index for that item doesn't have to be included. Here's an example:

```
my_list[:2]
# Grabs the first two items
my_list[3:]
# Grabs the fourth through last items
```

### Instructions

Assign each variable a slice of animals that spells out that variable's name.

```
animals = "catdogfrog"
cat =      # The first three characters of animals
dog =      # The fourth through sixth characters
frog =     # From the seventh character to the end
```

### 21.2.4 Maintaining Order

You can search through a list with the `index()` function. `my_list.index("dog")` will return the first index that contains the string "dog". An error will occur if there is no such item.

Items can be added to the middle of a list (instead of to the end) with the `insert()` function. `my_list.insert(4, "cat")` adds the item "cat" at index 4 of `my_list`, and moves the item previously at index 4 and all items following it to the next index (that is, they all get bumped forward by one).

### Instructions

Use the `index()` function to assign `duck_index` the index of the string equal to "duck". Then insert the string "cobra" at that index.

```

animals = ["aardvark", "badger", "duck", "emu", "fennec fox"]
duck_index =      # Use index() to find "duck"

# Your code here!

print animals # Observe what prints after the insert operation

```

### 21.2.5 For One and All

If you want to do something with every item in the list, you can use a for loop. If you've learned about for loops in JavaScript, pay close attention! They're different in Python.

Here's the syntax:

```

for variable in list_name:
    # Do stuff!

```

A variable name follows the for keyword; it will be assigned the value of each list item in turn. in list\_name designates list\_name as the list the loop will work on. The line ends with a colon (:) and the indented code that follows it will be executed once per item in the list.

#### Instructions

Write a statement in the indented part of the for loop that prints a number equal to  $2 * \text{number}$  for every list item.

```

my_list = [1,9,3,8,5,7]

for number in my_list:
    # Your code here

```

### 21.2.6 More with 'for'

If your list is a jumbled mess, you may need to sort() it. my\_list.sort() will sort the items in my\_list in increasing numerical/alphabetical order.

It's worth noting that sort() does not return a new list; instead, your existing my\_list is sorted in place (the sorted version replaces the unsorted version).

#### Instructions

Write a for loop that populates square\_list with items that are the square ( $x ** 2$ ) of each item in start\_list. Then sort square\_list!

```

start_list = [5, 3, 1, 2, 4]
square_list = []

# Your code here!

```

```
print square_list
```

## 21.3 Dictionaries

What if you want to store information using something other than the preset index values of 0, 1, 2, and so on? Dictionaries can do that for you!

### 21.3.1 This Next Part is Key

A dictionary is similar to a list, but you access values by looking up a key instead of an index. A key can be any string or number. Dictionaries are enclosed in curly braces, like so:

```
d = {'key1': 1, 'key2': 2, 'key3': 3}
```

This is a dictionary called `d` with three key-value pairs. The key '`'key1'`' points to the value 1, '`'key2'`' to 2, and so on.

Dictionaries are great for things like phone books (pairing a name with a phone number), login pages (pairing an e-mail address with a username), and more!

### Instructions

Print the values stored under the '`'Sloth'`' and '`'Burmese Python'`' keys. Accessing dictionary values by key is just like accessing list values by index:

```
residents['Puffin']
# Gets the value 104

# Assigning a dictionary with three key-value pairs to residents:
residents = {'Puffin' : 104, 'Sloth' : 105, 'Burmese Python' : 106}

print residents['Puffin'] # Prints Puffin's room number

# Your code here!
```

### 21.3.2 New Entries

A new key-value pair in a dictionary is created by assigning a new key, like so:

```
dict_name[new_key] = new_value
```

An empty pair of curly braces is an empty dictionary, just like an empty pair of `[]` is an empty list.

The length `len()` of a dictionary is the number of key-value pairs it has. Each pair counts only once, even if the value is a list. (That's right: you can put lists inside dictionaries!)

### Instructions

Add at least three key-value pairs to the menu variable, with the dish name (as a "string") for the key and the price (a float or integer) as the value. Here's an example:

```
menu[ 'Spam' ] = 2.50
```

```
menu = {} # Empty dictionary
menu[ 'Chicken Alfredo' ] = 14.50 # Adding new key-value pair
print menu[ 'Chicken Alfredo' ]

# Your code here: Add some dish-price pairs to menu!

print "There are " + str(len(menu)) + " items on the menu."
print menu
```

### 21.3.3 Changing Your Mind

Like lists, dictionaries are mutable (they can be changed). Items can be removed from a dictionary with the `del` command:

```
del dict_name[key_name]
```

will remove the key `key_name` and its associated value from the dictionary.

A new value can be associated with a key by assigning a value to the key, like so:

```
dict_name[key] = new_value
```

### Instructions

Delete the 'Sloth' and 'Bengal Tiger' items from `zoo_animals` using `del`.

Set the value associated with 'Rockhopper Penguin' to anything other than 'Arctic Exhibit'.

```
# key - animal_name : value - location
zoo_animals = { 'Unicorn' : 'Cotton Candy House',
'Sloth' : 'Rainforest Exhibit',
'Bengal Tiger' : 'Jungle House',
'Atlantic Puffin' : 'Arctic Exhibit',
'Rockhopper Penguin' : 'Arctic Exhibit'}
# A dictionary (or list) declaration may break across multiple lines

# Removing the 'Unicorn' entry. (Unicorns are incredibly expensive.)
del zoo_animals['Unicorn']

# Your code here!
```

```
print zoo_animals
```

### 21.3.4 It's Dangerous to Go Alone! Take This

Finally, `my_list.remove(value)` will remove the the first item from `my_list` that has a value equal to `value`. The difference between `del` and `.remove()` is:

- `del` deletes a key and its value based on the key you tell it to delete.
- `.remove()` removes a key and its value based on the value you tell it to delete.

#### Instructions

Final challenge! Add code that modifies the dictionary `inventory` in the following ways:

1. Add a key to `inventory` called '`pocket`'
2. Set the value of '`pocket`' to be a list consisting of the strings '`seashell`', '`strange berry`', and '`lint`'
3. `.sort()` the items in the list stored under the '`backpack`' key
4. Remove '`dagger`' from the list of items stored under the '`backpack`' key
5. Add 50 to the number stored under the '`gold`' key

```
inventory = {'gold' : 500,
'pouch' : ['flint', 'twine', 'gemstone'], # Assigned a new list to 'pouch' key
'backpack' : ['xylophone', 'dagger', 'bedroll', 'bread loaf']}

# Adding a key 'burlap bag' and assigning a list to it
inventory['burlap bag'] = ['apple', 'small ruby', 'three-toed sloth']

# Sorting the list found under the key 'pouch'
inventory['pouch'].sort()

# Here the dictionary access expression takes the place of a list name

# Your code here!
```

# Chapter 22

## Project: A Day at the Supermarket

Let's manage our own supermarket and buy some goods along the way!

### 22.1 Looping with Lists and Dictionaries

Learn how to use the for loop with lists and dictionaries.

#### 22.1.1 BeFOR We Begin

Before we begin our exercise, we should go over the Python for loop one more time. For now, we are only going to go over the for loop in terms of how it relates to lists and dictionaries. We'll explain more cool for loop uses in later courses.

for loops allow us to iterate through all of the elements in a list from the left-most (or zeroth element) to the right-most element. A sample loop would be structured as follows:

```
a = [1, 2, 3, 4, ...]
for x in a:
    # Do something for every x
```

This loop will run all of the code in the indented block under the for x in a: statement. The item in the list that is currently being evaluated will be x. So running the following:

```
for item in [1, 3, 21]:
    print item
```

would print 1, then 3, and then 21. The variable between for and in can be set to any variable name (currently item), but you should be careful to avoid using the word "list" as a variable, since that's a reserved word (that is, it means something special) in the Python language.

## Instructions

Use a for loop to print out all of the elements in the list names.

```
names = [ "Adam" , "Alex" , "Mariah" , "Martine" , "Columbus" ]
```

### 22.1.2 This is KEY!

You can also use a for loop on a dictionary to loop through its keys with the following:

```
d = { "foo" : "bar" }

for key in d:
    print d[key] # prints "bar"
```

Note that dictionaries are unordered, meaning that any time you loop through a dictionary, you will go through every key, but you are not guaranteed to get them in any particular order.

## Instructions

Use a for loop to go through the webster dictionary and print out all of the definitions.

```
webster = {
    "Aardvark" : "A star of a popular children's cartoon show." ,
    "Baa" : "The sound a goat makes." ,
    "Carpet": "Goes on the floor." ,
    "Dab": "A small amount."
}

# Add your code below!
```

### 22.1.3 Control Flow and Looping

The blocks of code in a for loop can be as big or as small as they need to be. While looping, you may want to perform different actions depending on the particular item in the list. This can be achieved by combining your loops with control flow (if/else statements) that might resemble the following:

```
for item in numbers:
    if condition:
        # Do something
```

Make sure to keep track of your indentation or you may get confused!

## Instructions

Loop through list a and only print out the even numbers.

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

### 22.1.4 Lists + Functions

Functions can also take lists as inputs and perform various operations on those lists.

#### Instructions

Write a function called `fizz_count` that takes a list `x` as input and returns the count of the string “fizz” in that list.

For example, `fizz_count(["fizz","buzz","fizz"])` should return 2. Check out the Hint if you need help!

### 22.1.5 String Looping

As we’ve mentioned, strings are like lists with characters as elements. You can loop through strings the same way you loop through lists! While we won’t ask you to do that in this section, we’ve put an example in the editor of how looping through a string might work.

#### Instructions

Run the code to see string iteration in action!

```
for letter in "Codecademy":
    print letter

# Empty lines to make the output pretty
print
print

word = "Programming is fun!"

for letter in word:
    # Only print out the letter i
    if letter == "i":
        print letter
```

## 22.2 Owning a Store

Learn how to run your very own supermarket!

### 22.2.1 Your Own Store!

Okay—on to the core of our project.

Congratulations! You are now the proud owner of your very own Codecademy brand supermarket!

The first thing that you need to do is figure out what your inventory is (and how much each item costs). Instructions

Set your prices to the following values. Put these values in a dictionary called `prices` and write your dictionary in the format.

```
"banana": 4  
"apple": 2  
"orange": 1.5  
"pear": 3
```

Yeah, this place is really expensive. (Your supermarket subsidizes the zoo from the last course.)

### 22.2.2 Investing in Stock

Good work! As a store manager, you're also in charge of keeping track of your stock/inventory. Instructions

Create a stock dictionary with the values below.

```
"banana": 6  
"apple": 0  
"orange": 32  
"pear": 15
```

### 22.2.3 Keeping Track of the Produce

Now that you have all of your product info, you need to make a formal document with all of your inventory information. This will not only give you a big-picture sense of your stock situation, but will help you figure out which items you need to order (and which ones you may have over-ordered last time).

#### Instructions

Print out all of the items in your store along with their prices. Print the answer in the following format:

```
item  
price: x  
stock: x
```

So when you print out the data for apples, print out:

```
apple  
price: 2  
stock: 0
```

Each of these values should be in a different print statement. Remember to convert numbers to strings before trying to combine them, and check the Hint if you need help.

### 22.2.4 Something of Value

For paperwork and accounting purposes, let's record the total value of your inventory. It's nice to know what we're worth!

#### Instructions

Loop through your dictionaries in order to figure out how much money you would make if you sold all of the food in stock. Print that value into the console!

```
prices = {
    "banana": 4,
    "apple": 2,
    "orange": 1.5,
    "pear": 3
}

stock = {
    "banana": 6,
    "apple": 0,
    "orange": 32,
    "pear": 15
}
```

## 22.3 Shopping Trip!

Let's buy some goods at the supermarket.

### 22.3.1 Shopping at the Market

Great work! Now we're going to take a step back from the management side and take a look through the eyes of the shopper.

In order for customers to order online, we are going to have to make a consumer interface. Don't worry: it's easier than it sounds!

#### Instructions

First, make a list (not a dictionary!) with the name groceries. Insert a "banana", "orange", and "apple".

### 22.3.2 Making a Purchase

Good! Now you're going to need to know how much you're paying for all of the items on your grocery list.

Remember how to compute a rolling sum? If not, here's a reminder: `total += price` is the same as `total = total + price`. Evaluating the right hand side grabs the values of those variables, adds those values and then stores it back into the variable `total`.

### Instructions

Write a function `compute_bill` that takes a parameter `food` as input and computes your bill by looping through your food list and summing the costs of each item in the list.

For now, go ahead and ignore whether or not the item you're billing for is in stock.

```
groceries = ["banana", "orange", "apple"]

stock = {"banana": 6,
         "apple": 0,
         "orange": 32,
         "pear": 15
        }

prices = {"banana": 4,
          "apple": 2,
          "orange": 1.5,
          "pear": 3
         }

# Write your code below!
```

### 22.3.3 Stocking Out

Now you need your `compute_bill` function to take the stock/inventory of a particular item into account when computing the cost.

Ultimately, if an item isn't in stock, then it shouldn't be included in the total. You can't buy or sell what you don't have!

### Instructions

Do the following for your `compute_bill` function:

1. Let your function take a list of groceries as input.
2. Do not add the price of an item into your list if it is out of stock.
3. After you buy an item, subtract one from its stock.
4. If an item is in your list multiple times you must repeat this process multiple times.

```
groceries = ["banana", "orange", "apple"]

stock = {"banana": 6,
         "apple": 0,
         "orange": 32,
         "pear": 15
        }

prices = {"banana": 4,
          "apple": 2,
          "orange": 1.5,
```

```
    "pear": 3
}
# Write your code below!
```

### 22.3.4 Let's Check Out!

Perfect! You've done a great job with lists and dictionaries in this project. You've practiced:

- Using for loops with lists and dictionaries
- Writing functions with loops, lists, and dictionaries
- Updating data in response to changes in the environment (for instance, decreasing the number of bananas in stock by 1 when you sell one).

Thanks for shopping at the Codecademy supermarket!

### Instructions

```
groceries = ["banana", "orange", "apple"]

stock = {
    "banana": 6,
    "apple": 0,
    "orange": 32,
    "pear": 15
}

prices = {
    "banana": 4,
    "apple": 2,
    "orange": 1.5,
    "pear": 3
}
# Write your code below!
```



# Chapter 23

## Lists and Functions



# Chapter 24

## Loops



# Chapter 25

## Exam Statistics



# Chapter 26

## Advanced Topics in Python



## Chapter 27

# Introduction to Classes



# Chapter 28

## File Input and Output



# **Appendix A**

## **Resources**

This chapter includes resources that are complimentary to the information presented in this book, and that are useful for further reading. Resources are divided by sections, based on their categories.