CS303e Course Introduction

Chapman: I didn't expect a kind of Spanish Inquisition.

Cardinal Ximinez[Palin]: NOBODY expects the Spanish Inquisition! Our chief weapon is surprise...surprise and fear...fear and surprise.... Our two weapons are fear and surprise...and ruthless efficiency.... Our **three** weapons are fear, surprise, and ruthless efficiency...and an almost fanatical devotion to the Pope.... Our **four**...no... **Amongst** our weapons.... Amongst our weaponry...are such diverse elements as fear, surprise....

Mike Scott

scottm@cs.utexas.edu

www.cs.utexas.edu/~scottm/cs303e

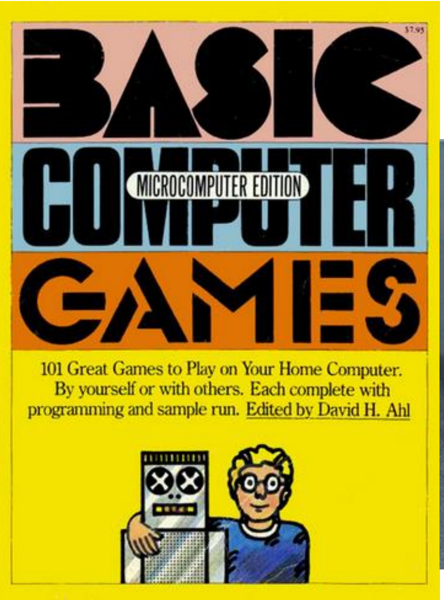
Agenda

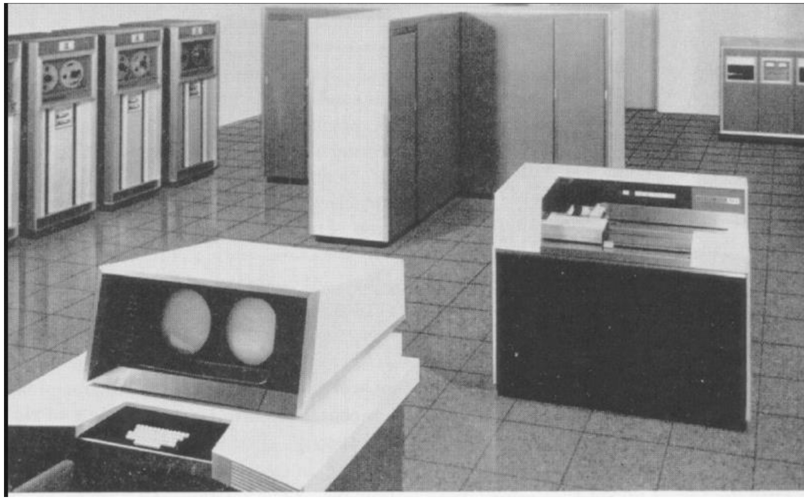
- Overview of:
  - this course
  - the elements of computing program
- Course logistics including:
  - how to get help
  - the schedule
  - tips for success

Who Am I

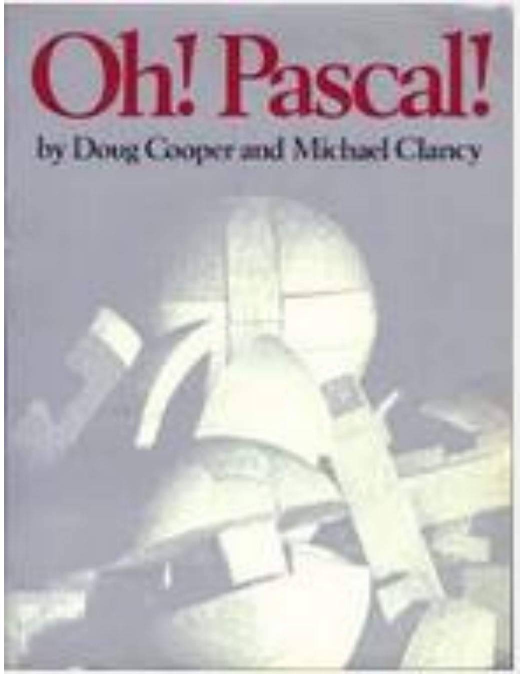
- Lecturer in CS department since 2000
- Undergrad Stanford, MSCS RPI
- US Navy for 8 years, submarines
- 2 years Round Rock High School

My Path to CS





```
 10 INPUT "What is your name: "; U$
 20 PRINT "Hello "; U$
 25 REM
 30 INPUT "How many stars do you want: "; N
 35 S$ = ""
 40 FOR I = 1 TO N
 50 S$ = S$ + "*"
 55 NEXT I
 60 PRINT S$
 65 REM
 70 INPUT "Do you want more stars? "; A$
 80 IF LEN(A$) = 0 THEN GOTO 70
 90 A$ = LEFT$(A$, 1)
100 IF (A$ = "Y") OR (A$ = "y") THEN GOTO 30
110 PRINT "Goodbye ";
120 FOR I = 1 TO 200
130 PRINT U$; " ";
140 NEXT I
150 PRINT
```



Intro to Programming

- Learn to design and implement computer programs to solve problems.
- I assume you have NEVER written a single line of code

1. output, fstrings
2. identifiers
3. errors (syntax, runtime, logic)
4. reserved words
5. variables, operators, computations
6. constants
7. built in math functions
8. conditional execution
9. boolean logic
10. iteration, repetition
11. programmer defined functions
12. Strings
13. lists
14. lists of lists (matrices)
15. files
16. exceptions
17. dictionaries
18. objects and classes (programmer defined data types)
19. recursion
20. sorting and searching

Programing and CS

- A tool for doing the cool stuff in CS
- You can't create a self driving vehicle without the software to control the vehicle

Programming

- Start simple ….
- … but get complex by end of the class

```
>>>
>>>
>>>
= RESTART: C:/Users/scottm/Documents/303e/_Su 20/example programs
/assignments/Initials.py

   MDS

   MMMM     MMMM        DDDDDDDD            SSSSSSSSS
   MMMM     MMMM        DDDDDDDDDD         SSSSSSSSSS
   MM  MM  MM  MM       DD       DDD      SSSS
   MM  MM  MM  MM       DD        DD      SSSS
   MM  MM  MM  MM       DD        DD        SSSSSS
   MM   MMMM   MM       DD        DD          SSSSSS
   MM   MMMM   MM       DD        DD              SSSS
   MM    MM    MM       DD       DDD              SSSS
   MM    MM    MM   ..  DDDDDDDDDD   ..   SSSSSSSSSS    ..
   MM    MM    MM   ..  DDDDDDDD     ..   SSSSSSSSS     ..

>>>
>>>
>>>
```

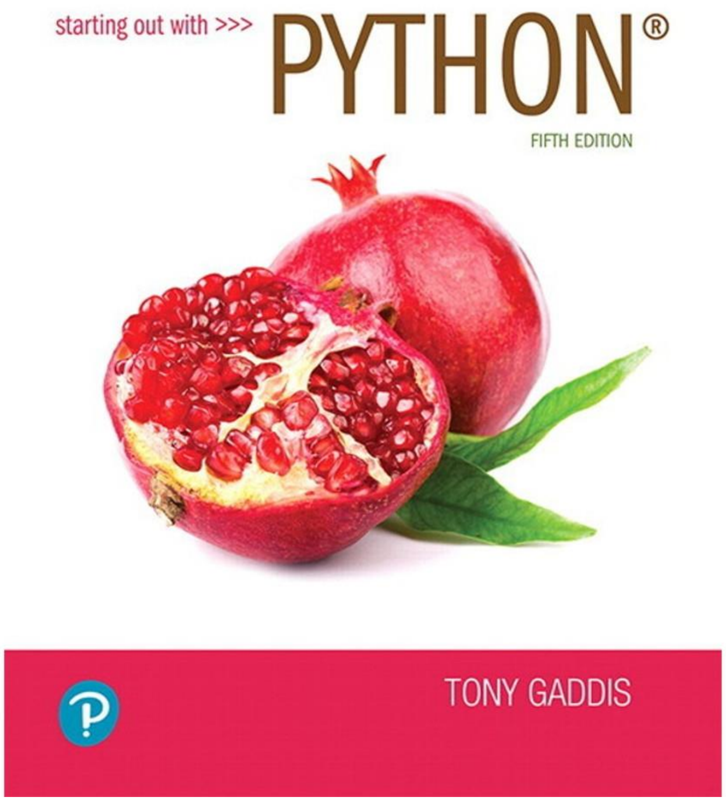
```
P2
250 360
255
 104 103 105  99  92  92  94 100 100
 109 103 105 100  96  92  93  94  88
 109 105  99  98  98 102 105  93  88
 106 110 104  99 103 112 112  97  91
 113 113 112 103 105 109 107  91  90
 121 121 119 107 103  99  95  90  90
 114 117 112 105 105  95  92  93  94
 101 107 103 103 105  96  92  96  97
  94 108 110 104 103 101  97  99 100
  94 112 123 106  99  98  95  96  96
 100 116 123 107  99 100  95 101  96
 104 115 113 105  99 100  97 104 104
 103 107 108 107 103 101  93  92  96
 103 107 104 107 106 102  97  91  90
 109 107 104 105 108 104 105 105  98
 106 110 109 106 108 105 107 109 108
 106 120 127 123 117 108 107 111 111
```

Startup

- If you have not already done so …
- … complete the items on the class start-up page
- http://www.cs.utexas.edu/~scottm/cs303e/handouts/startup.htm

Book

- book is required
  - we follow it quite closely
- programming assignments, limited to features from the book up to a given chapter
- suggested exercises



Graded Course Components

- Programming projects
  - 13 projects, 10 or 20 points : **210 points**
- Exams
  - Midterm, In class Wednesday, July 3, 11:30 am – 1:30 pm **400 points**
  - Final, Thursday, August 1, 7 - 10 pm **400 points**
- Extra credit
  - CS background survey on Canvas. **10 points**
  - course survey completion, **10 points**

210 + 400 + 400 + 10 + 10 = 1030

- Programming Assignments capped at 200 pts
  - 30 points of “slack”, including extra credit
- No points added! Grades based on 1000 points, not 1030
- Final point total = **min(200, sum of points on programs + background survey completion + instructor end of course survey) + midterm exam score + final exam score**

Letter Grades

- Final grade determined by final point total

>= 925 ->   A
900 - 924 -> A-
875 - 899 -> B+
825 - 874 -> B
800 - 824 -> B-
775 - 799 -> C+
725 - 774 -> C
700 - 724 -> C-
675 - 699 -> D+
625 - 674 -> D
600 - 624 -> D-
<= 599   ->   F

In Class Exercises - Grade Bump

- Recall: Final point total = **min(200, sum of points on programs + background survey completion + instructor end of course survey) + midterm exam score + final exam score**
- Each lecture shall have an in-class programming exercise. 21 total. Completing these may help you get bumped to the next higher grade if you are close to a cutoff.
- 1 point added for every 2 exercises completed with reasonable effort
  - rounded up
- **For example, you end up with 893 points per the formula above. You complete 14 or more of the 21 in class exercises with a reasonable attempt. You grade shall be bumped from B+ to A-.**

Assignments

- Start out simple but get more challenging
- **Individual – do your own work**
- **Programs checked automatically with plagiarism detection software, MOSS**
- Turn in the right thing - correct name, correct format or you will lose points / slip days
- Slip days
  - 8 for term, max 1 per assignment
  - don’t use frivolously
- Graded on correctness and *program hygiene* (style, best practices), typical 60% / 40% split

Getting Help

- Post to Ed (link on Canvas).
  - can make anonymous to other students
  - can post to instructors only
  - do not post more than 2 lines of code on a public post
- Help Hours
  - check schedule
  - Most help hours in person in GDC 3.202
  - A few help hours via Zoom, check the Canvas course page and the Zoom tab for links

Succeeding in the Course

- Randy Pausch,
  CS Professor at CMU said:

- *"When I got tenure a year early at Virginia, other Assistant Professors would come up to me and say, 'You got tenure early!?!?! What's your secret?!?!?' and I would tell them, 'Call me in my office at 10pm on Friday night and I'll tell you.' "*

- *"A lot of people want a shortcut. I find the best shortcut is the long way, which is basically two words: **work hard**."*

Succeeding in the Course - Meta

- "Be the first penguin" Randy Pausch
  - Ask questions!!!
  - lecture, Piazza, help hours

- "It is impossible to be perfect" Captain Symons
  - Mistakes are okay.
  - That is how we learn.
  - Trying to be perfect means not taking risks.
  - no risks, no learning

Succeeding in the Course - Concrete

- Whole course is cumulative!
- Material builds on itself
  - failure to understand a concept leads to bigger problems down the road, so …
- do the readings
- come to class
- start on assignments early
- get help from the teaching staff when you get stuck on an assignment
- participate on the class discussion group
- ask questions and get help when needed
- **DO MORE PRACTICE PROBLEMS -> Book, CodingBat, Professor Bulko's Site**

Succeeding in the Course

- Cannot succeed via memorization.
- The things I expect you to do are **not** rote.
  - programming is a skill
  - you cannot memorize your way through the material and the course
- Learn by doing.
- If you are brand new to programming or have limited experience I ***strongly*** recommend you do ***lots and lots of practice problems.***

CS303E: Elements of Computers and Programming

Python

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from Dr. Bill Young's Slides

Last updated: May 23, 2023

Some Thoughts about Programming

"The only way to learn a new programming language is by writing programs in it." –B. Kernighan and D. Ritchie

"Computers are good at following instructions, but not at reading your mind." –D. Knuth

"Programming is not a spectator sport." - Bill Young

Program:

n. A magic spell cast over a computer allowing it to turn one's input into error messages.

tr. v. To engage in a pastime similar to banging one's head against a wall, but with fewer opportunities for reward.

What is Python?

Python is a high-level programming language developed by Guido van Rossum in the Netherlands in the late 1980s. It was released in 1991.

Python has twice received recognition as the language with the largest growth in popularity for the year (2007, 2010).

It's named after the British comedy troupe Monty Python.

What is Python?

Python is a simple but powerful ***scripting*** language. It has features that make it an excellent first programming language.

- Easy and intuitive mode of interacting with the system.
- Clean syntax that is concise. You can say/do a lot with few words.
- Design is compact. You can carry the most important language constructs in your head.
- There is a very powerful library of useful functions available.

You can be productive quite quickly. You will be spending more time solving problems and writing code, and less time grappling with the idiosyncrasies of the language.

What is Python?

Python is a **general purpose** programming language. That means you can use Python to write code for any programming tasks.

- Python was used to write code for: the Google search engine
  - mission critical projects at NASA
  - programs for exchanging financial transactions at the NY Stock Exchange
  - the grading scripts for this class

What is Python?

Python can be an **object-oriented** programming language. Object-oriented programming is a powerful approach to developing reusable software. More on that later!

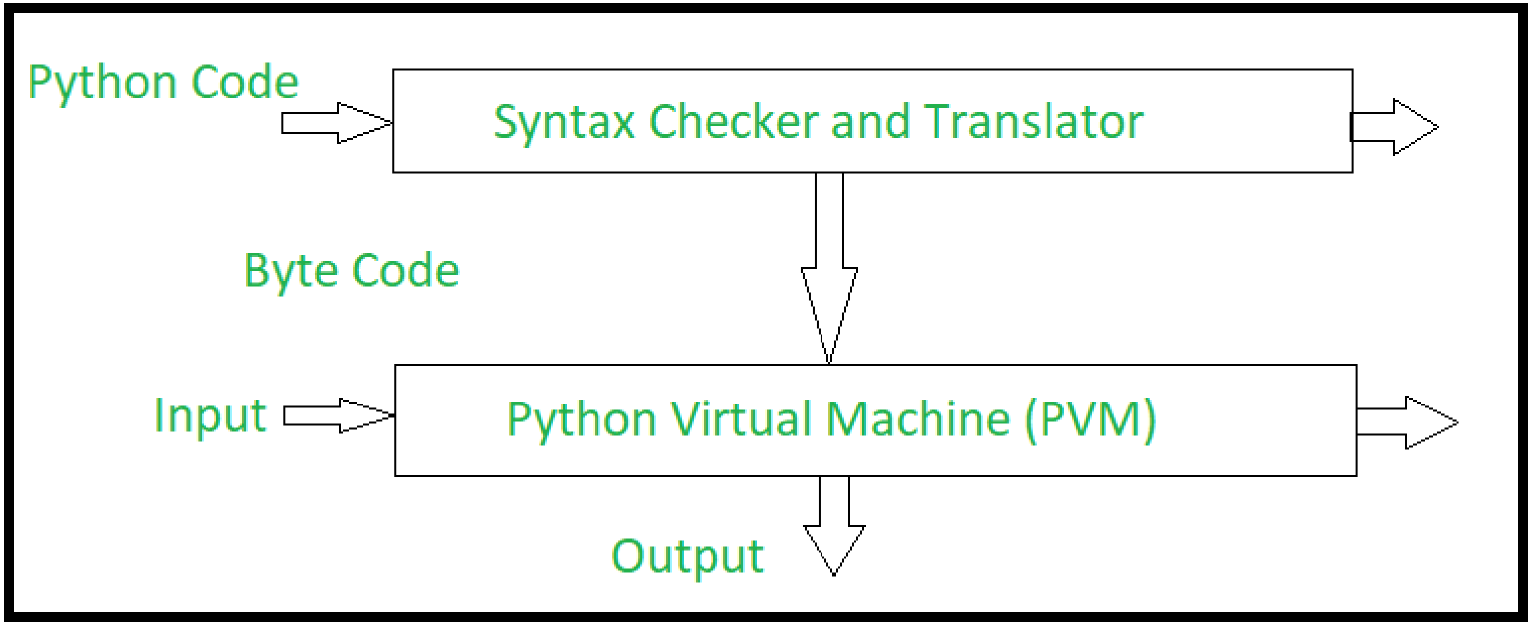
Python is **interpreted**, which means that Python code is translated and executed one statement at a time.

This is different from other languages such as C which are ***compiled*****, the code is converted to machine code and then the program can be run after the compilation is finished.**

The Interpreter

Actually, Python is always translated into **byte code**, a lower level representation.

The byte code is then interpreted by the Python Virtual Machine.



Getting Python

To install Python on your personal computer / laptop, you can download it for free at: www.python.org/downloads

- There are two major versions: Python 2 and Python 3. Python 3 is newer and *is not backward compatible with Python 2.* Make sure you're running Python 3.8.
- It's available for Windows, Mac OS, Linux.
- If you have a Mac, it *may* already be pre-installed.
- It should already be available on most computers on campus.
- It comes with an editor and user interface called IDLE.
- I strongly recommend downloading and installing the PyCharm, Educational version, IDE.

A Simple Python Program: Interactive Mode

This illustrates using Python in **interactive mode** from the command line. *Your command to start Python may be different.*

```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:0
D64)] on win32
Type "help", "copyright", "credits" or "license()" f
>>> print('Hello World!')
Hello World!
>>> print('Hook \'em Horns!')
Hook 'em Horns!
>>> print((10.5   +   2   *   3)   /   45   -   3.5)
-3.1333333333333333
>>>
```

Here you see the prompt for the OS/command loop for the Python interpreter read, eval, print loop.

A Simple Python Program: Script Mode

Here's the "same" program as I'd be more likely to write it. Enter the following text using a text editor into a file called, say, MyFirstProgram.py. This is called *script mode*.

In file my_first_program.py:

```
def main():
    # Display two messages.
    print('Hello World!')
    print('Hook \'em Horns!')

    # Evaluate an arithmetic expression :
    print((10.5 + 2 * 3) / 45 - 3.5)


main()
```

A Simple Python Program

```
Hello World!
Hook 'em Horns!
-3.1333333333333333


Process finished with exit code 0
```

This submits the program in file my_first_program.py to the Python interpreter to execute.

This is better, because you have a file containing your program and you can fix errors and resubmit without retyping a bunch of stuff.

Aside: About Print

If you do a computation and want to display the result use the `print` function.

You can print multiple values with one print statement:

```
>>> print('The value is: ', 2 * 10)
The value is:  20
>>> print(3 + 7, 3 - 10)
10 -7
>>> 3 + 7
10
>>> 3 - 10
-7
>>> 3 + 7, 3 - 10
(10, -7)
>>>
```

Notice that if you're computing an expression in interactive mode, *it will display the value without an explicit* `print`.

Python will figure out the type of the value and print it appropriately. This is very handy when learning the basics of computations in Python.

Another aside: Binary Numbers, Base 2 Numbers

- The vast majority of computer systems use digital storage
- Some physical phenomena that is interpreted to be a 0 or 1
  - abstraction, pretending something is different, simpler, than it really is
- also known as binary representations
- 1 bit -> 1 binary digit, a 0 or a 1
- 1 byte -> 8 bits
- binary numbers, base 2 numbers

Base 2 Numbers

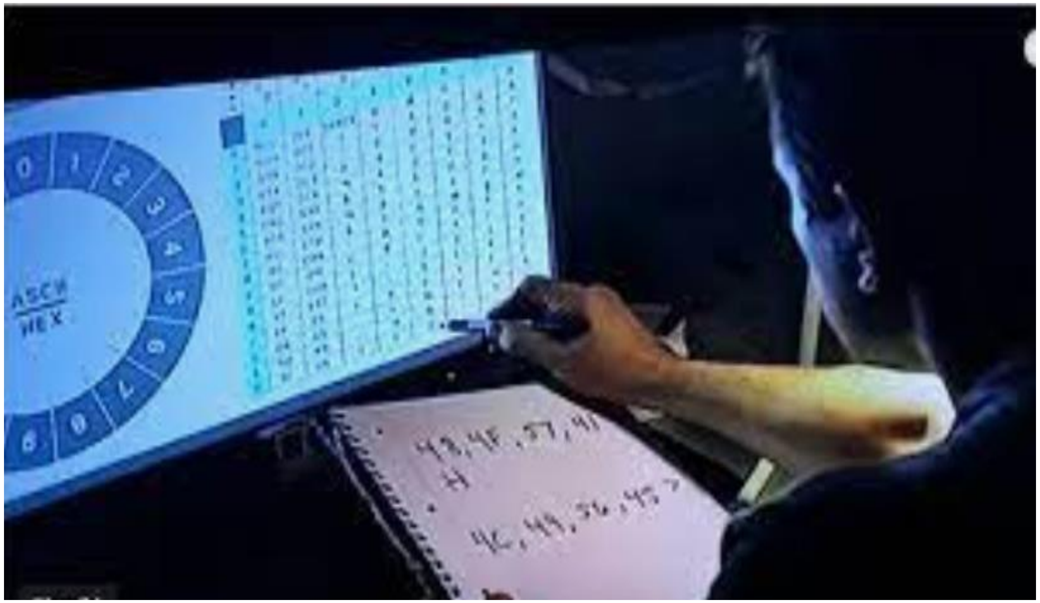
- 5372_{10}
- $= (5 * 1{,}000) + (3 * 100) + (7 * 10) + (2 * 1)$
- $= (5 * 10^3) + (3 * 10^2) + (7 * 10^1) + (2 * 10^0)$
- Why do we use base 10? 10 fingers?
- Choice of base is somewhat arbitrary
- In computing we also use base 2, base 8, and base 16 depending on the situation
- In base 10, 10 digits, 0 - 9
- In base 2, 2 digits, 0 and 1

Base 2 Numbers

- 1011011_2
- $= (1 * 64) + (0 * 32) + (1 * 16) + (1 * 8) + (0 * 4) + (1 * 2) + (1 * 1) = 91$
- $= (1 * 2^6) + (0 * 2^5) + (1 * 2^4) + (1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) = 91$
- Negative numbers and real numbers are typically stored in a non-obvious way
- If the computer systems only stores 0s and 1s how do we get digital images, characters, colors, sound, …
- Encoding

Encoding

- Encoding is a system or standard that dictates what "thing" is representing by what number
- Example ASCII or UTF-8
- This number represents this character
- First 128 numbers of ASCII and UTF-8 same
- 32 -> space character
- 65 -> capital A
- 97 -> lower case a
- 48 -> digit 0



Computer Memory

- Recall, 1 bit -> a single 0 or 1
- 1 byte = 8 bits
- A typical laptop or desktop circa 2023
- … has 4 to 32 Gigabytes of RAM, also known as main memory.
    - 1 Gigabyte -> 1 billion bytes
- The programs that are running store their instructions and data (typically) in the RAM
- … have 100s of Gigabytes up to several Terabytes (trillions of bytes) in secondary storage. Long term storage of data, files
- Typically spinning disks or solid state drives.

The Framework of a Simple Python Program

Define your program in file Filename.py:

```
def main():

    Python statement
    Python statement
    Python statement
        ...
    Python statement
    Python statement
    Python statement

main()
```

Defining a function called main.

These are the instructions that make up your program. *Indent all of them the same amount (usually 4 spaces).*

This says to execute the function main.

To run it:

```
> python file_name.py
```

This submits your program in file_name.py to the Python interpreter.

Typically, if your program is in file hello.py, you can run your program by typing at the command line:

```
> python hello.py
```

You can also create a *stand alone script*. On a Unix / Linux machine you can create a *script* called hello.py containing the first line below (assuming that's where your Python implementation lives):

```
#!/lusr/bin/python3
# The line above may vary based on your system
print('Hello World!')
```

Program Documentation

Documentation refers to comments included within a source code file that explain what the code does.

- Include a **file header:** a summary at the beginning of each file explaining what the file contains, what the code does, and what key feature or techniques appear.
- You shall always include your name, email, grader, and a brief description of the program.

```
# File: <NAME OF FILE>
# Description: <A DESCRIPTION OF YOUR PROGRAM>
# Assignment Number: <Assignment Number, 1 - 13>
#
# Name: <YOUR NAME>
# EID:  <YOUR EID>
# Email: <YOUR EMAIL>
# Grader: <YOUR GRADER'S NAME Carolyn OR Emma or Ahmad>
#
# On my honor, <YOUR NAME>, this programming assignment is my own work
# and I have not provided this code to any other student.
```

Program Documentation

- Comments shall also be interspersed in your code:
  - Before each function or class definition (i.e., program subdivision);
  - Before each major code block that performs a significant task;
  - Before or next to any line of code that may be hard to understand.

```
sum = 0
# sum the integers [start ... end]
for i in range( start, end + 1):
    sum += i
```

Don't Over Comment

Comments are useful so that you and others can understand your code. Useless comments just clutter things up:

```
x = 1          # assign 1 to x
y = 2          # assign 2 to y
```

Programming Style

Every language has its own unique syntax and *style*. This is a C program.

Good programmers follow certain *conventions* to make programs clear and easy to read, understand, debug, and maintain. We have conventions in 303e. Check the assignment page.

```c
#include <stdio.h>

/* print table of Fahrenheit to Celsius
   [C = 5/9(F-32)] for fahr = 0, 20, ...,
       300 */

main()
{
  int fahr, celsius;
  int lower, upper, step;

  lower = 0;      /* low limit of table */
  upper = 300;    /* high limit of table */
  step = 20;      /* step size */
  fahr = lower;
  while (fahr <= upper) {
    celsius = 5 * (fahr-32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

Programming Style

Some **important** Python programming conventions:

- Follow variable and function naming conventions.
- Use meaningful variable/function names.
- Document your code **effectively**.
- Each level indented the same (4 spaces).
- Use blank lines to separate segments of code inside functions.
- 2 blank lines before the first line of function (the function header) and after the last line of code of the function

We'll learn more elements of style as we go.

Check the assignments page for more details.

Errors:

Remember: "Program: *n.* A magic spell cast over a computer allowing it to turn one's input into error messages."

We will encounter three types of *errors* when developing our Python program.

syntax errors: these are ill-formed Python and caught by the interpreter prior to executing your code.

```
>>> 3 = x
  File "<stdin>", line 1
SyntaxError: can't assign to
literal
```

These are typically the easiest to find and fix.

Errors: Runtime

runtime errors: you try something illegal while your code is executing

```
>>> x = 0
>>> y = 3
>>> y / x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Almost Certainly It's Our Fault!

At some point we all say: "My program is obviously right. The interpreter / Python must be incorrect / flaky / and it hates me."

"As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

-Sir Maurice V Wilkes

Errors: Logic

logic errors: `Calculate 6! (6 * 5 * 4 * 3 * 2 * 1)`
your program runs but returns an incorrect result.

```
>>> prod = 0
>>> for x in range(1, 6):
...     prod *= x
>>> print(prod)
0
```

This program is syntactically fine and runs without error. But it probably doesn't do what the programmer intended; it always returns 0 no matter the values in range. How would you fix it?

Logic errors are typically the hardest errors to find and fix.

Try It!

“The only way to learn a new programming language is by writing programs in it.” –B. Kernighan and D. Ritchie

Python is wonderfully accessible. If you wonder whether something works or is legal, just try it out.

Programming is not a spectator sport! Write programs! Do exercises!

CS303E: Elements of Computers and Programming

Simple Python

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: June 5, 2023

"Once a person has understood the way variables are used in programming, they have understood the quintessence of programming."

-Professor Edsger W. Dijkstra

Simple Program: Body Mass Index

- **B**ody **M**ass **I**ndex or **BMI** is a quick calculation based on height and mass (weight) used by medical professionals to broadly categorize people .
- Formula:

$$\text{BMI} = \frac{\text{mass}_{\text{kg}}}{\text{height}_{\text{m}}^2} = \frac{\text{mass}_{\text{lb}}}{\text{height}_{\text{in}}^2} \times 703$$

- Quick tool to get a rough estimate if someone is underweight, normal weight, overweight, or obese
- Write an interactive program that gets the name, height, and weight of a user and calculates BMI.

Assignment Statements

An assignment in Python has form:

<variable> = <expression>

This means that variable is *assigned* **value.** i.e., after the assignment, **variable** "contains" **value.**

The equals sign is NOT algebraic equality.
It causes an action! The *expression* on the right is evaluated and the result is assigned to the variable on the left.

```
>>> x = 17.2
>>> y = -39
>>> z = x * y - 2
>>> print( z )
-672.8
```

Variables

A **variable** is a named memory location (in the RAM typically) used to store values. We'll explain shortly how to name variables.

Unlike some programming languages, Python variables do not have fixed data types.

```
// Ccode
int x = 17;        // variable x has type int
x = 5.3;           // illegal
```

```
# Python code
x = 17             # x gets int value 17
x = 5.3            # x gets float value 5.3
```

A variable in Python actually holds a *pointer* to a class object, rather than the object itself.
A variable exists at a particular *address.* Each memory location (4 or 8 bytes typically circa 2021) has an address or location. A number that specifies that location in memory

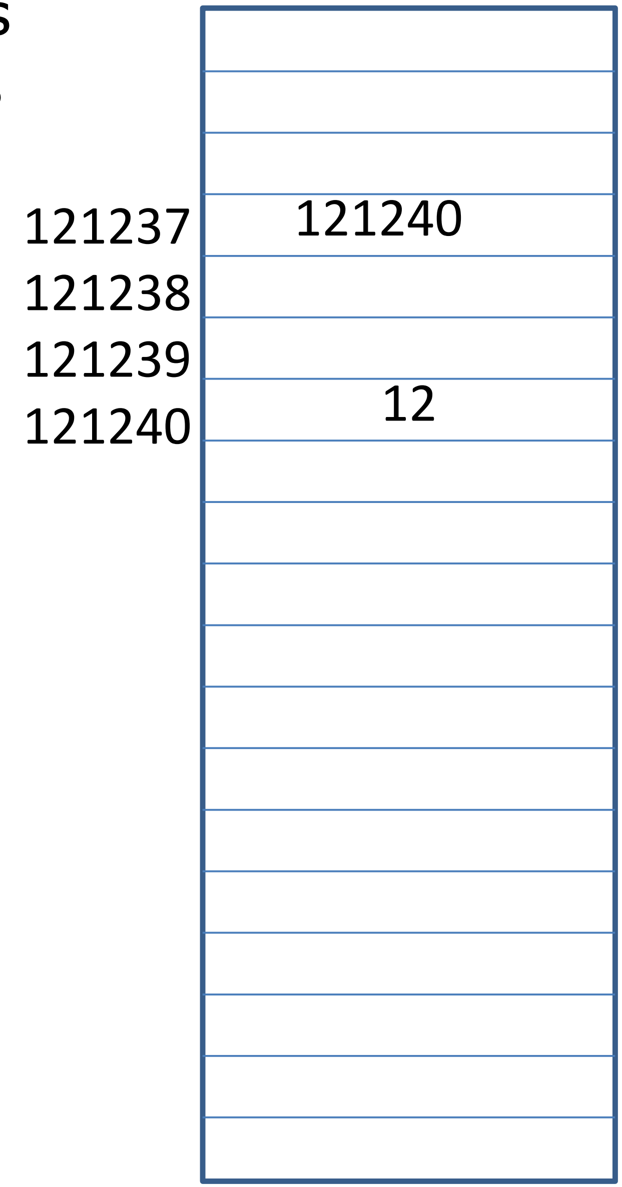
What's a Pointer?

- Also called references, but pointers and references have differences that are beyond the scope of this class.
- A variable exists at a particular *address.* Each memory location (4 or 8 bytes typically circa 2021) has an address or location. A number that specifies that location in memory.
    - Just like the address of a house or building on a street
- So a variable is just a name in our program for a spot in the RAM that stores a value.
- But Python (for reasons we don't want to talk about now) has a bit of " bureaucracy" when a **variable** is bound to a **value**

x = 12

let's assume the variable x is at memory

location 121237



Types in Python

Is it correct to say that there are no types in Python?

Yes and no. It is best to say that Python is "dynamically typed." Variables in Python are untyped, but values have associated data types (actually classes). In some cases, you can convert one type to another.

Most programming languages assign types to both variables and values. This has its advantages and disadvantages.

What do you think the advantages are of requiring variables to declare the data type of a variable?

You can create a new variable in Python by assigning it a value. *You don't have to declare variables' types, as in many other programming languages.*

```
>>> x = 3                    # creates x, assigns int
>>> print(x)
3
>>> x = "abc"                # re-assigns x a string
>>> print(x)
abc
>>> x = 3.14                 # re-assigns x a float
>>> print(x)
3.14
>>> y = 6                    # creates y, assigns int
>>> x * y                    # uses x and y
18.84
```

Meaning of a Variable

```
x = 17        # Defines and initializes x
y = x + 3     # Defines y and initializes y
z = w         # Runtime error if w undefined
```

This code defines three variables x, y and z. Notice that on the *left hand side* of an assignment the variable is created (if it doesn't already exist), and given a value.

On the *right hand side* of an assignment is an expression. When the assignment statement is run the expression shall be evaluated and the resulting value will be bound to the variable on the left hand side.

Naming Variables

Below are (most of) the rules for naming variables:

- Variable names must begin with a letter or underscore (_) character.
- After that, use any number of letters, underscores, or digits.
- Case matters: "score" is a different variable than "Score."
- You can’t use *reserved words*; these have a special meaning to Python and cannot be variable names.

Python Reserved Words.

Also known as Keywords.

and, as, assert, break, class, continue, ***def****, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, nonlocal, None, not, or, pass, raise, return, True, try, while, with, yield*

IDLE, PyCharm, and other IDEs display reserved words in a different color to help you recognize them.

Not Reserved, but avoid using names of common functions

- A function is a subprogram.
- Python has many built in functions we will use.
- Function names like `print` are *not* reserved words. But using them as variable names is *a very bad idea* because it redefines them.

```
>>> x = 12
>>> print(x)
12
>>> print = 37
>>> print(x)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(x)
TypeError: 'int' object is not callable
>>>
```

Naming Variables

```
>>> ___ = 10                          # not standard but legal
>>> _123 = 11                         # also not standard
>>> ab_cd = 12                        # fine
>>> ab|c = 13                         # illegal character
  File "<stdin>", line 1
SyntaxError: can't assign to operator
>>> assert = 14                       # assert is reserved
  File "<stdin>", line 1
    assert = 14
           ^
SyntaxError: invalid syntax
>>> max_value = 100                   # good
>>> print = 8                         # legal but ill-advised
>>> print( "abc" )                    # we've redefined print
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Naming Variables

In addition to the rules, there are also some conventions that programmers follow and we expect you to follow in CS303e:

- Variable names shall begin with a lowercase letter.
- Choose meaningful names that describe how the variable is used. This helps with program readibility.

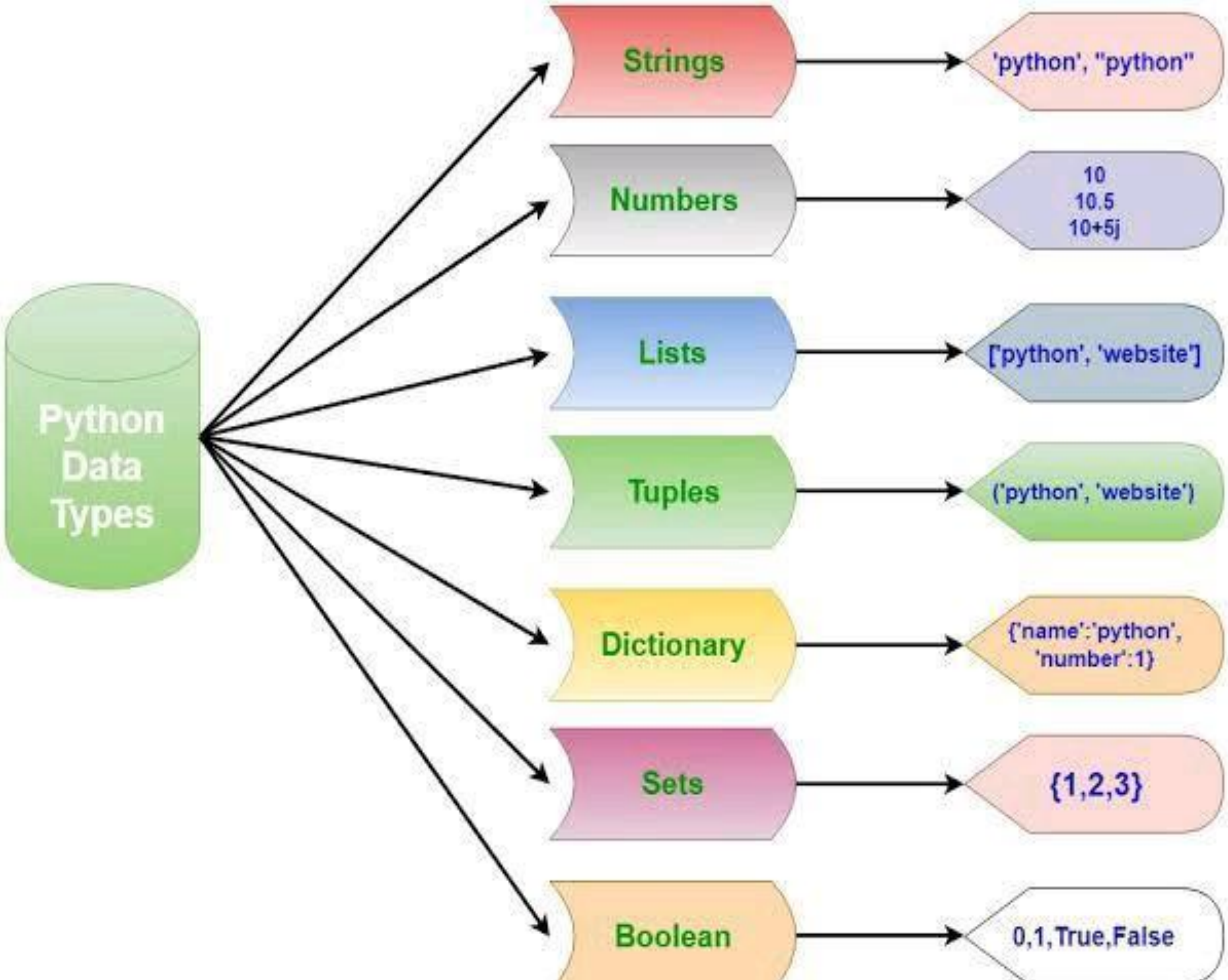
  Use max rather than m.
  Use num_columns rather than c.
- Use underscores to separate multiple words
- loop variables are often i, j, etc.

```
for i in range(1, 20):
    print(i)
```

rather than:

```
for some_value in range(1, 20):
    print(some_value)
```

Common Python Data Types



What is a Data Type?

A **data type** is a categorization of values.

Data Type	Description	Example
int	integer. An immutable number of unlimited magnitude	42
float	A real number. An immutable floating point number, system defined precision	3.1415927
str	string. An immutable sequence of characters	'Wikipedia'
bool	boolean. An immutable truth value	True, False
tuple	Immutable sequence of mixed types.	(4.0, 'UT', True)
list	Mutable sequence of mixed types.	[12, 3, 12, 7, 6]
set	Mutable, unordered collection, no duplicates	{12, 6, 3}
dict	dictionary a.k.a. maps, A mutable group of (key, value pairs)	{'k1': 2.5, 'k2': 5}

Others we likely won't use in 303e:
complex, bytes, frozenset

The `type` Function

```
>>> x = 17
>>> type(x)
<class 'int'>
>>> y = -20.9
>>> type(y)
<class 'float'>
>>> type(w)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'w' is not defined
>>> lst = [1, 2, 3]
>>> type(lst)
<class 'list'>
>>> type(20)
<class 'int'>
>>> type( (2, 2.3 ) )
<class 'tuple'>
>>> type('abc')
<class 'str'>
>>> type( {1, 2, 3} )
<class 'set'>
>>> type(print)
<class 'builtin_function_or_method'>
```

- *Class* is another name for data type.
- Data type is a categorization or classification
- "What kind of thing is the value this variable refers to?"

Three Common Data Types

Three data types we will use in many of our early Python programs are:

`int`: signed integers (whole numbers)

- Computations are exact and *of unlimited size*
- Examples: 4, -17, 0

`float`: signed real numbers (numbers with decimal points) Large

- range, but fixed precision
- Computations are approximate, not exact Examples:
- 3.2, -9.0, 3.5e7

`str`: represents text (a string)

- We use it for input and output We'll see
- more uses later Examples: "Hello, World!",
- 'abc'

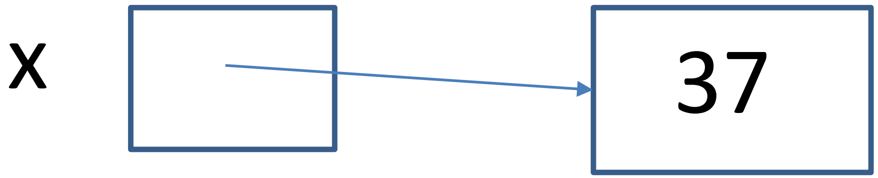
These are all *immutable.* The value cannot be altered.

Immutable

- It may appear some values are mutable
  - they are not
  - rather variables are mutable and can be bound (refer to) different values
- Note, how the id of x (*similar to its address*) has changed

```
>>> x = 37
>>> x
37
>>> id(x)
140711339416352
>>> x = x + 10
>>> x
47
>>> id(x)
140711339416672
```

x = 37



x = x + 10

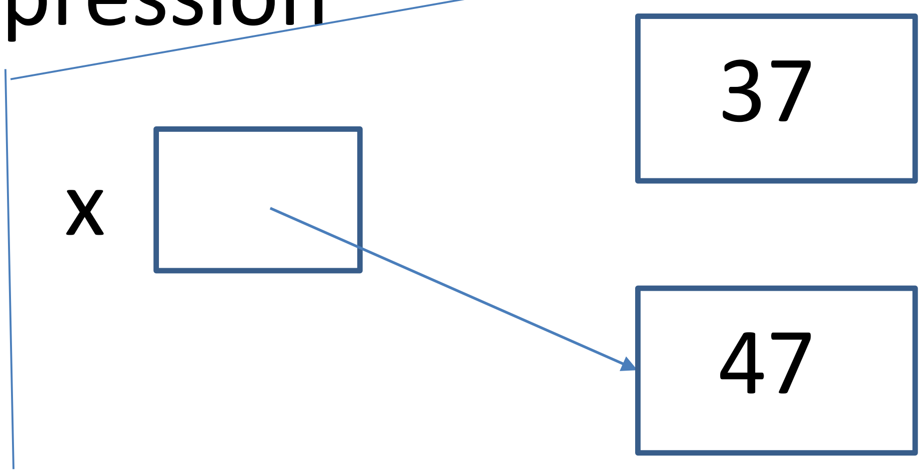
substitute in the value x is referring to

x = 37 + 10

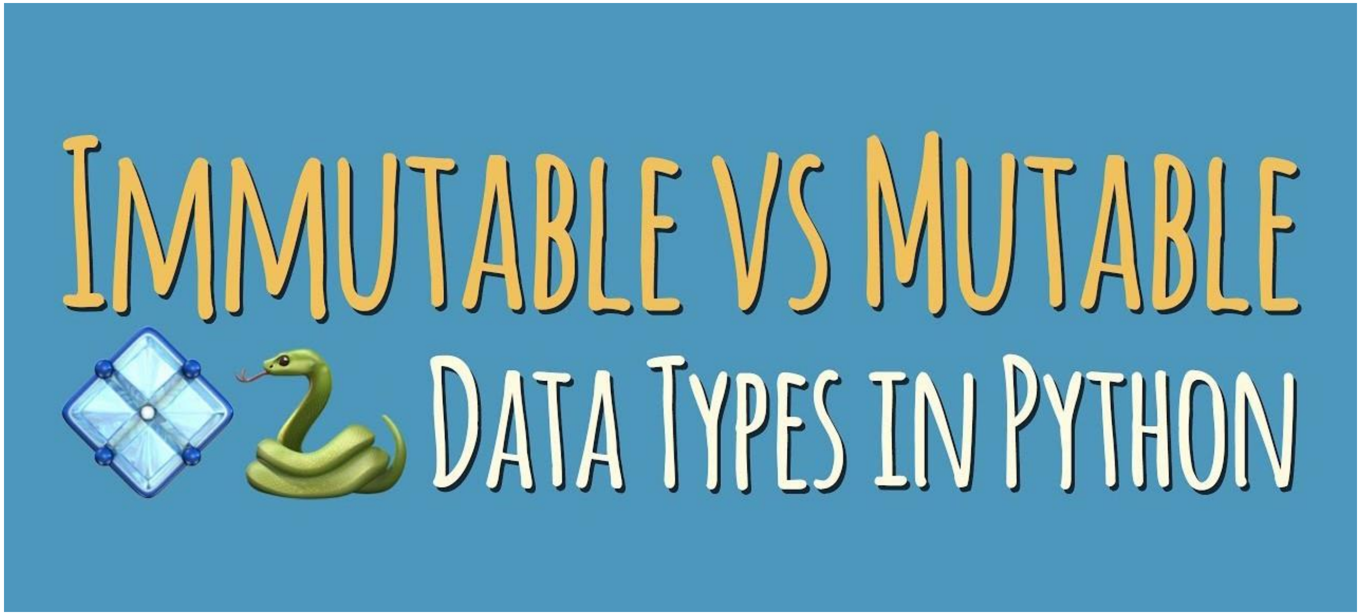
evaluate the expression

x = 47

so now …



Mutable vs. Immutable



An **immutable** value is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.

A **mutable** values is one that can be changed; e.g., sets, lists, etc.

What Immutable Means

- An **immutable** object is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.
- It also means that *there is typically only one copy of the object in memory.*
- Whenever the system encounters a new reference to 17, say, it creates a pointer (references) to the already stored value 17.
- Every reference to 17 is actually a pointer to the *only* copy of 17 in memory. Ditto for "abc".
- If you do something to the object that yields a new value (e.g., uppercase a string), you're actually creating a new object, not changing the existing one.

Immutability

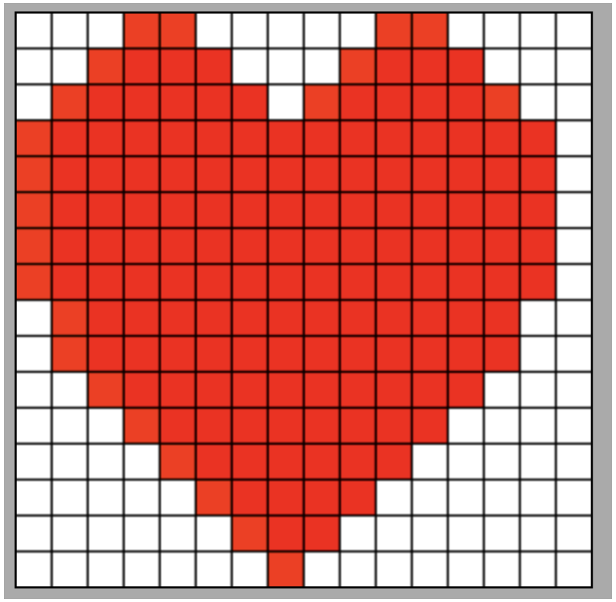
```
>>> x = 17             # x holds a pointer to the object 17
>>> y = 17             # so does y
>>> x is y             # x and y point to the same object
True
>>> id(x)              # the unique id associated with 17
10915008
>>> id(y)
10915008
>>> s1 = "abc"         # creates a new string
>>> s2 = "ab" + "c"    # creates a new string (?)
>>> s1 is s2           # actually it doesn't!
True
>>> id(s1)
140197430946704
>>> id(s2)
140197430946704
>>> s3 = s2.upper()       # uppercase s2
>>> print(s3)
ABC
>>> id(s3)             # this is a new string
140197408294088
```

Let's Take a Break

Review from chapter 1

Fundamental fact: *all data* in the computer is stored as a series of bits (0s and 1s) in the memory.

That's true whether you're storing numbers, letters, documents, pictures, movies, sounds, programs, etc. *Everything!*



A key problem in designing any computing system or application is deciding how to *represent* the data we care about as a sequence of bits.

How is Data Stored: Digital Images

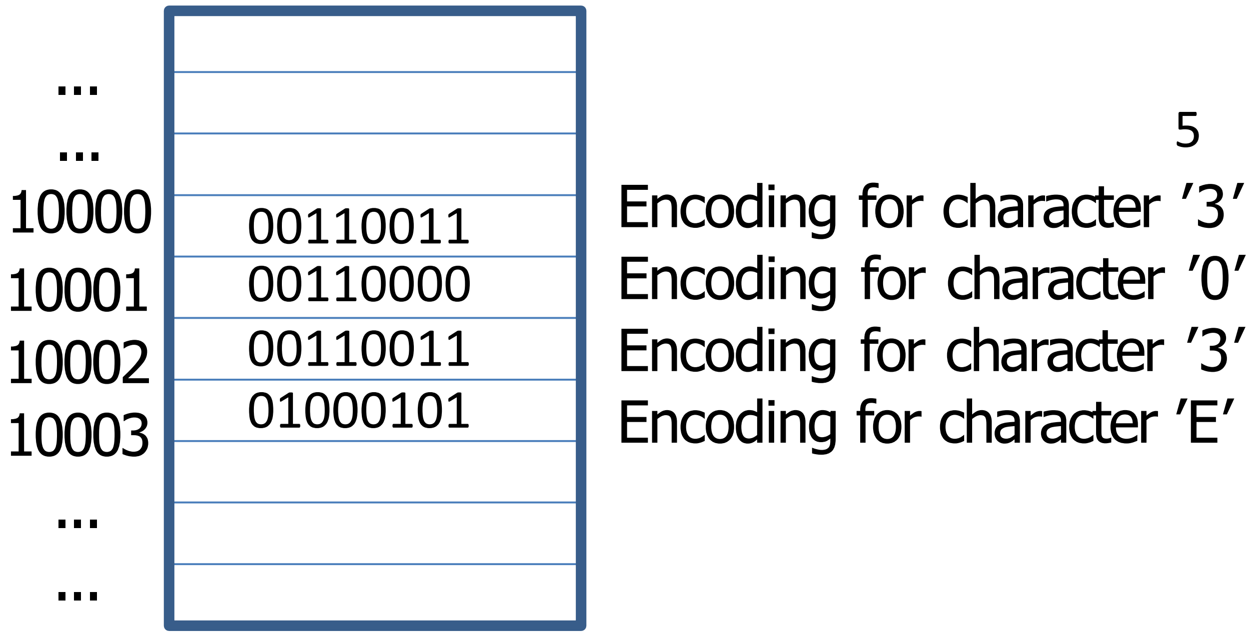
For example, images can be stored digitally in any of the following formats (among others):

- JPEG: Joint Photographic Experts Group
- PNG: Portable Network Graphics
- GIF: Graphics Interchange Format
- TIFF: Tagged Image File
- PDF: Portable Document Format
- EPS: Encapsulated Postscript

Most of the time, we won't need to know how data is stored in the memory. The computer will take care of that for us.

Standards?

The memory can be thought of as a big array of **bytes**, where a byte is a sequence of 8 bits. Each memory address has an **address** (0..maximum address) and **contents** (8 bits).



A byte is the smallest unit of storage a programmer can address. We say that the memory is *byte-addressable*.

Contemporary computer systems may have addressability of 4 or 8 bytes instead of single bytes,

Representation Example: ASCII

The standard way to represent *characters* in memory is ASCII. The following is part of the ASCII (American Standard Code for Information Interchange) representation:

032	sp	048	0	064	@	080	P	096	`	112	p
033	!	049	1	065	A	081	Q	097	a	113	q
034	"	050	2	066	B	082	R	098	b	114	r
035	#	051	3	067	C	083	S	099	c	115	s
036	$	052	4	068	D	084	T	100	d	116	t
037	%	053	5	069	E	085	U	101	e	117	u
038	&	054	6	070	F	086	V	102	f	118	v
039	'	055	7	071	G	087	W	103	g	119	w
040	(	056	8	072	H	088	X	104	h	120	x
041	)	057	9	073	I	089	Y	105	i	121	y
042	*	058	:	074	J	090	Z	106	j	122	z
043	+	059	;	075	K	091	[	107	k	123	{
044	,	060	<	076	L	092	\	108	l	124	|
045	-	061	=	077	M	093	]	109	m	125	}
046	.	062	>	078	N	094	^	110	n	126	~
047	/	063	?	079	O	095	_	111	o	127	⌂

The standard ASCII table defines 128 character codes (from 0 to 127), of which, the first 32 are control codes (non-printable), and the remaining 96 character codes are printing characters.

How is Data Stored

- Characters or small numbers can be stored in one byte. If data can't be stored in a single byte (e.g., a large number), it must be split across a number of adjacent bytes in memory.
- The way data is encoded in bytes varies
  - depending on: the data type
  - the specifics of the computer
- *Most of the time, we won't need to know how data is stored in the memory.* The computer will take care of that for us.

Formats of Data Types

- It would be nice to look at the character string "25" and do arithmetic with it.
- However, the `int` 25 (a number) is represented in binary in the computer by: 00011001. Why?
- And the string "25" (two characters) is represented by: 00110010 00110101. Why?
- `float` numbers are represented in an even more complicated way, since you have to account for an exponent. (Think "scientific notation.") So the number 25.0 (or $2.5 * 10^1$) is represented in yet a third way.

Data Type Conversion - Using Built in Functions

- Python provides functions to *explicitly* convert numbers from one type to another:

  float (< number, variable, string >)
  int (<number, variable, string >)
  str (<number, variable >)

- Note: int *truncates*, meaning it throws away the decimal point and anything that comes after it. If you need to *round* to the nearest whole number, use:

  round (<number or variable >)

Conversion Examples

```
float(17)
17.0
>>> str(17)
'17'
>>> int(17.75)                  # truncates
17
>>> str(17.75)
'17.75'
>>> int("17")
17
>>> float("17")
17.0
>>> round(17.1)
17
>>> round(17.6)
18
round(17.5)                     # round to even
18
>>> round(18.5)                 # round to even
18
```

Conversion Examples

If you have a string that you want to (try to) interpret as a number, you can use eval.

```
>>> eval("17")
17
>>> eval("17 + 3")
20
>>> eval(17 + 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: eval() arg 1 must be a string,
   bytes or code object
```

What was wrong with the last example?

Be Cautious Using `eval`

- Using the function `eval` is considered dangerous, especially when applied to user input.
- `eval` passes its argument to the Python interpreter, and a malicious (or careless) user could input a command string that could:
  - delete all of your files,
  - take over your machine, or
  - some other horrible thing.
- **Use `int()` or `float()` is you want to convert a string input into one of these types.**

Arithmetic Operations

Here are some useful operations you can perform on numeric data types.

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Float division	1 / 2	0.5
//	floor division	1 // 2	0
**	Exponentiation	4 ** 0.5	2.0
%	Remainder	20 % 3	2

(x % y) is often referred to as "x mod y"

Integer Division

- Floor Division specified with the // operator
- … goes to the *floor* on a number line
- Discards the remainder from the division operation.

```
>>> 37 // 10
3
>>> 17 // 20
0
>>> 2.5 // 2.0
1.0
>>> -22 // 7
-4
>>> -22 // -7
3
```

Modulo Operator

- % is the Modulo operator
- x % y evaluates to the remainder of x // y
- "The floor division and modulo operators are connected by the following identity:"

```
>>> 37 % 10
7
>>> 17 % 20
17
>>> -22 % 7
6
>>> -22 % -7
-1
```

```
x == (x // y) * y + (x % y)
```

- **B**ody **M**ass **I**ndex or **BMI** is a quick calculation based on height and mass (weight) used by medical professionals to broadly categorize people .
- Formula:

$$\mathrm{BMI} = \frac{\mathrm{mass}_{\mathrm{kg}}}{\mathrm{height}_{\mathrm{m}}^2} = \frac{\mathrm{mass}_{\mathrm{lb}}}{\mathrm{height}_{\mathrm{in}}^2} \times 703$$

- Quick tool to get a rough estimate if someone is underweight, normal weight, overweight, or obese
- Write an interactive program that gets the name, height, and weight of a user and calculates BMI.

Simple Input

- Obtain input from the user by calling a built in Python function named **`input`**.
- Just like we can send information (arguments) to **`print`**, we can send information (again, arguments) to **`input`**.
    - The argument is a prompt that will be displayed.
- Trying reading a height and weight from the user and calculating BMI.
- What happens?

- More built in functions to convert from String data type to int or float data type. **`int(), float()`**

Simple Program: Pythagorean Triples

In file pythagoreanTriple.py:

```
""" The sides of a right triangle satisfy the relation:
       a**2 + b**2 = c**2.
    Test whether the three integers in variables a, b, c
    form a pythagorean triple, i.e., satisfy this relation.
       """

a = 3
b = 4
c = 5
ans = ( a**2 + b**2 == c**2 )
print("a:", a, "b:", b, "c:", c, \
      "is" if ans else "is not", \
      "a pythagorean triple" )
```

```
> python pythagoreanTriple.py
a: 3 b: 4 c: 5 is a pythagorean triple
```

Note, print can take multiple values.

Default separator is a space,

default end is a newline

Augmented Assignment Operators

Python (like C, Java, C++...) provides a shorthand syntax for some common assignments:

`i += j`	functionally the same as	`i = i + j`
`i -= j`	functionally the same as	`i = i - j`
`i *= j`	functionally the same as	`i = i * j`
`i /= j`	functionally the same as	`i = i / j`
`i //= j`	functionally the same as	`i = i // j`
`i %= j`	functionally the same as	`i = i % j`
`i **= j`	functionally the same as	`i = i ** j`

```
>>> x = 2.4
>>> x *= 3.7    # functionally same as x = x * 3.7
>>> print(x)
8.88
```

Mixed-Type Expressions

Most arithmetic operations behave as you would expect for numeric data types.

- Combining two floats results in a float.
- Combining two ints results in an int (except for /). Use // for integer division.
- Dividing two ints gives a float. E.g., `2 / 5` yields 2.5.
- Combining a float with an int usually yields a float.

Python will figure out what the result will be and return a value of the appropriate data type.

Mixed Type Expressions

```
>>> 5 * 3 - 4 * 6                # (5 * 3) - (4 * 6)
-9
>>> 4.2 * 3 - 1.2
11.400000000000002               # approximate result
>>> 5 // 2 + 4                   # integer division
6
>>> 5 / 2 + 4                    # float division
6.5
```

Special Assignment Statements

Simultaneous assignments:

```
m, n = 2, 3
```

means the same as:

```
m = 2
n = 3
```

With the caveat that these happen *at the same time*.

What does the following do?

```
i, j = j, i
```

Multiple assignments:

```
i = j = k = 1
```

means the same as:

```
k = 1
j = k
i = j
```

Note that these happen right to left.

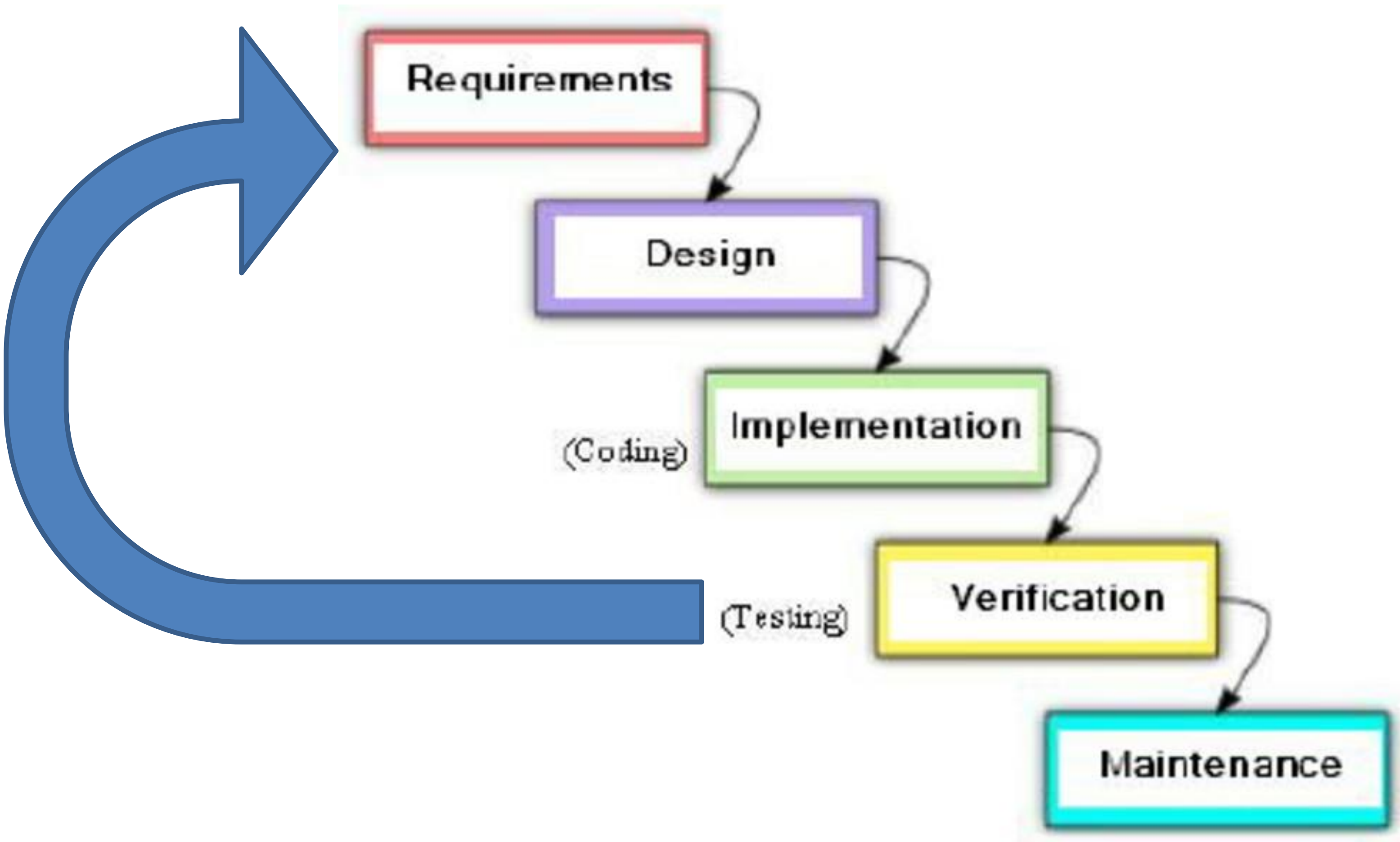
Advice on Programming

Think before you code!
Think before you code!
Think before you code!

- Don't jump right into writing code.
- Think about the overall process of solving your problem; write it down.
- Refine each part into subtasks.
  Subtasks may require further refinement.
- Code and test each subtask before you proceed.
- Add `print` statements to view intermediate results.

Advice on Programming

Software development is typically done via an iterative process. You'll do well to follow it, except on the simplest programs.



CS303E: Elements of Computers and Programming

Conditionals and Boolean Logic

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: May 31, 2023

Booleans

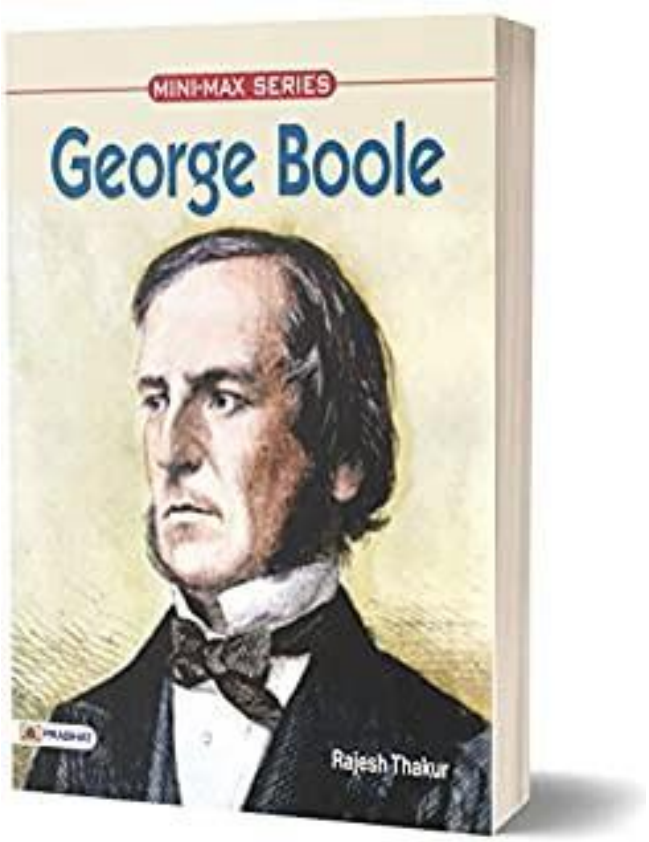
So far we've been considering *straight line code*, meaning executing one statement after another.

a.k.a*. sequential flow of control*

But often in programming, you need to ask a question, and *do different things* based on the answer.

Boolean values are a useful way to refer to the answer to a yes/no question.

The Boolean **literal values** are the values: True, False.
A Boolean **expression** evaluates to a Boolean value.



Using Booleans

```
>>> import math
>>> b = ( 30.0 < math.sqrt( 1024 ))
>>> print( b)
True
>>> x = 1                 # statement
>>> x < 0                 # boolean expression
False
>>> x >= -2               # boolean expression
True
>>> b = ( x == 0 )        # statement containing
                          # boolean expression
>>> print (b)
False
```

Booleans are implemented in the bool class.

Booleans

Internally, Python uses 0 to represent False and anything not 0 to represent True. You can convert from Boolean to int using the int function and from int to Boolean using the bool function.

```
>>> b1 = (-3 < 3)
>>> print(b1)
True
>>> bool(1)
True
>>> bool(0)
False
>>> bool(4)
True
>>>
```

Boolean Context

In a **Boolean context**—one that expects a Boolean value—False, 0, "" (the empty string), and None all is considered False and *any other value* is considered True.

```
>>> bool("xyz")
True
>>> bool(0.0)
False
>>> bool("")
False
>>> if 4: print("xyz")        # boolean context
xyz
>>>if 4.2: print("xyz")
xyz
>>> if "ab": print("xyz")
xyz
```

This may be confusion but can be very useful in some programming situations.

Comparison Operators

The following comparison (or relational) operators are useful for comparing numeric values:

Operator	Meaning	Example
<	Less than	x < 0
<=	Less than or equal	x <= 0
>	Greater than	x > 0
>=	Greater than or equal	x >= 0
==	Equal to	x == 0
!=	Not equal to	x != 0

Each of these returns a Boolean value, True or False.

```
>>> x = 10
>>> (x == math.sqrt(100))
True
>>> (x = math.sqrt(100))
SyntaxError: invalid syntax
```

What happened on that last line?

Caution

Be very careful using "==" when comparing *floats*, because float arithmetic is approximate.

```
>>> (1.1 * 3 == 3.3)
False                              # What happened?
>>> 1.1 * 3
3.3000000000000003
```

The problem: converting decimal 1.1 to binary yields a *repeating* binary expansion: 1.000110011 . . . = 1.00011. That means *it can't be represented exactly* in a fixed size binary representation.

Thought for the day. Some rational numbers are repeating decimals in one base, but not in others. $1/3 = 0.33333\ldots_{10} = 0.1_3$

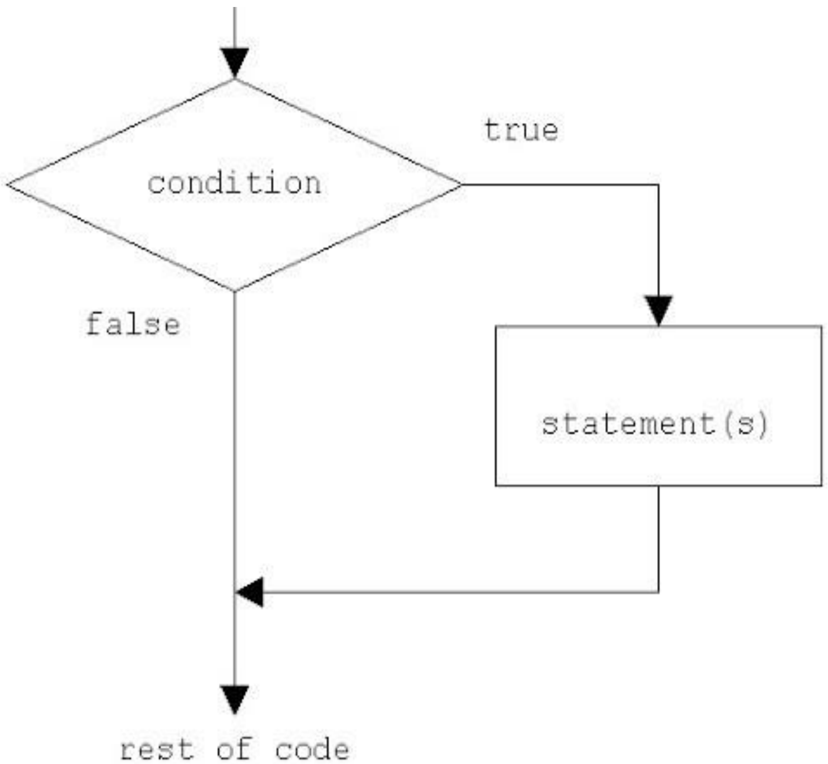
One Way If Statements

It's often useful to be able to perform an action *only if* some conditions is true.

General form:

```
if boolean-expression:
    statement(s)
```

Note the colon after the boolean-expression.
All of the statements controlled by the if must be indented the same amount.




```
if y != 0:
    z = ( x / y )
```

If Statement Example

In file if_example.py:

```
def main():
    # A very uninteresting  function to
    # illustrate  if statements.
    x = int(input(" Input an integer or 0 to do nothing: "))
    if (x != 0):
        print('The number you entered was',
              x, '. Thank you!')
```

Would "if x:" have worked instead of "if (x != 0):"?

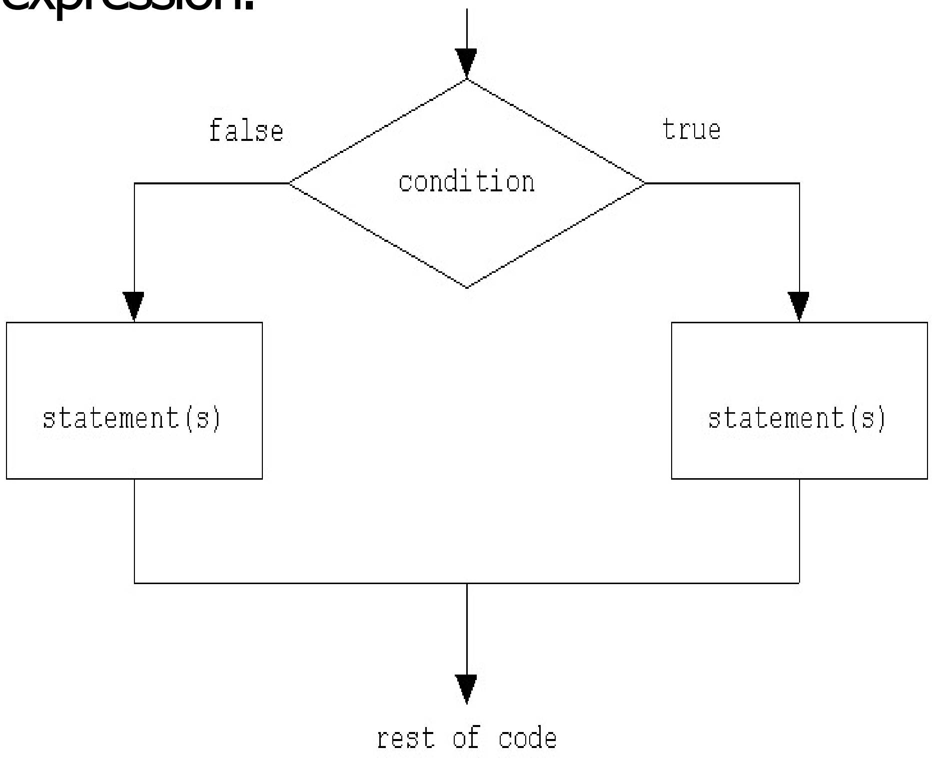
```
>>> runfile('C:/Users/scottm/PycharmProjects/As
Input an integer or 0 to do nothing: >? 10
The number you entered was 10 . Thank you!
>>> runfile('C:/Users/scottm/PycharmProjec
Input an integer or 0 to do nothing: >? 0
```

Two-way If-else Statements

A two-way **If-else** statement executes one of two actions, depending on the value of a Boolean expression.

General form:

```
if boolean-expression:
    true-case-statement(s)
else:
    false-case-statement(s)
```



Note the colons after the boolean-expression and after the `else`. All of the statements in *both* if and else branches should be indented the same amount.

If-else Statement: Example

In file compute_circle_area.py:

```
import math


def main():
    # Estimate area of circle based on radius from user
    radius = float(input("Enter the radius of a circle: "))
    if (radius >= 0):
        area = math.pi * radius ** 2
        print('A circle with a radius of ', radius,
              'has an area of ', area)
    else:
        print('Negative radius entered: ', radius)


main()
```

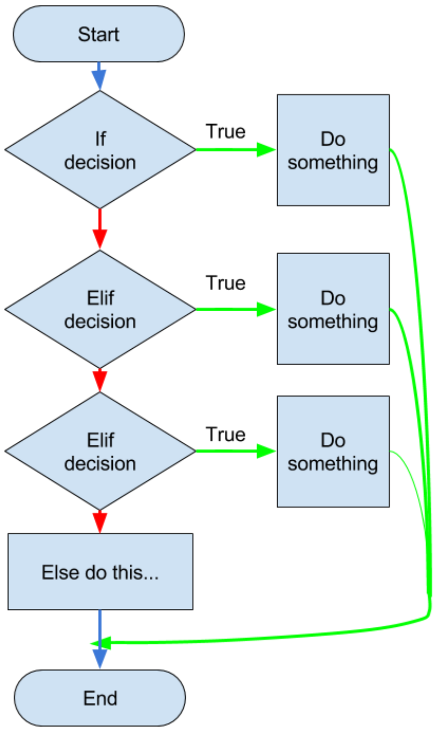
```
Enter the radius of a circle: 4.3
A circle with a radius of 4.3 has an area of 58.088048

Enter the radius of a circle: -3.75
Negative radius entered: -3.75
```

Multiway if-elif-else Statements

If you have multiple options, you can use if-elif-else statements.

General Form:

```
if boolean-expression1:
    statement(s)
elif boolean-expression2:
    statement(s)
elif boolean-expression3:
    ...
else:                    # optional
    statement(s)
```



You can have any number of `elif` branches with their conditions. The else branch is optional.

Sample Program: Calculate US Federal Income Tax

Simplified US Federal Income Tax Table

Source: https://www.nerdwallet.com/article/taxes/federal-income-tax-brackets

Single filers

Tax rate	Taxable income bracket	Tax owed
10%	$0 to $9,875	10% of taxable income
12%	$9,876 to $40,125	$987.50 plus 12% of the amount over $9,875
22%	$40,126 to $85,525	$4,617.50 plus 22% of the amount over $40,125
24%	$85,526 to $163,300	$14,605.50 plus 24% of the amount over $85,525
32%	$163,301 to $207,350	$33,271.50 plus 32% of the amount over $163,300

income_tax.py

```python
# Ask user for income and calculate US Federal income tax for 2021.
# Tax rates and income bracket data from
# https://www.nerdwallet.com/article/taxes/federal-income-tax-brackets
def main():
    income = int(input('Enter 2021 income: '))
    print()
    if income <= 9_875:
        tax = income * 0.1
        bracket = "10%"
    elif income <= 40_125:
        tax = 987.5 + (income - 9_875) * 0.12
        bracket = "12%"
    elif income <= 85_525:
        tax  = 4_617.50 + (income - 40_125) * 0.22
        bracket = "22%"
    elif income <= 163_300:
        tax = 14_605.50 + (income - 85_525) * 0.24
        bracket = "24%"
    else:
        tax = 33_271.50 + (income - 163_300) * 0.32
        bracket = "32%"

    print('An income of', income, 'places you in the',
          bracket, 'income bracket.')
    print('The US Federal tax on an income of', income,
          'is', tax)
```

Break

Maybe take a break?

Logical Operators

Python has **logical operators** (and, or, not) that can be used to make compound Boolean expressions.

not : logical negation

and : logical conjunction

or : logical disjunction

Operators **and** and **or** are always evaluated using *short circuit evaluation*.

$$(\ x \ \% \, 100 \ == \ 0 \ ) \ \ \text{and} \ \ \text{not} \ (\ x \ \% \, 400 \ == \ 0 \ )$$

Truth Tables

And: $(\mathrm{A} \text{ and } \mathrm{B})$ is True whenever both A is True and B is True.

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Not: $\text{not } \mathrm{A}$ is True whenever A is False.

A	not A
False	True
True	False

Or: $(\mathrm{A} \text{ or } \mathrm{B})$ is True whenever either A is True or B is True.

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Remember that "is True" really means "is not False, the empty string, 0, or None."

Short Circuit Evaluation

Notice that (A and B) is False, if A is False; it doesn't matter what B is.
So there's no need to evaluate B, if A is False!

Also, (A or B) is True, if A is True; it doesn't matter what B is.
So there's no need to evaluate B, if A is True!

```
>>> x = 13
>>> y = 0
>>> legal = (y == 0 or x / y > 0)
>>> print(legal)
True
```

Python doesn't evaluate B if evaluating A is sufficient to determine the value of the expression. *That's important sometimes.*
This is called *short circuiting* the evaluation.
Stopping early when answer it know.

Boolean Operators

In a Boolean context, Python doesn't always return True or False, just something equivalent. What's going on in the following?

```
>>> "" and 14
''                                # equivalent to False
>>> bool("" and 14)
False                             # coerced to False
>>> 0 and "abc"
0                                 # equivalent to False
>>> bool(0 and "abc")
False                             # coerced to False
>>> not(0.0)                      # same as not( False )
True
>>> not(1000)                     # same as not( True )
False
>>> 14 and ""
''                                # equivalent to False
>>> 0 or "abc"                    # same as False or True
'abc'                             # equivalent to True
>>> bool(0 or 'abc')              # coerced to True
True
```

Leap Years

Here's a concise way to do a Leap Year computation:

```
# Determine if year entered is a leap year or not.
def main():
    year = int(input('Enter a year: '))
    is_leap_year = ((year % 4 == 0)
                    and (not (year % 100 == 0) or (year % 400 == 0)))

    if is_leap_year:
        print(year, "is a leap year.")
    else:
        print(year, 'is not a leap year.')


main()
```

Note the use of outer parenthesis on the assignment to is_leap_year to avoid the use of the continuation character, "\".

Leap Years Revisited

```
>python LeapYear2.py
Enter a year: 2000
Year 2000 is a leap year.
>python LeapYear2.py
Enter a year: 1900
Year 1900 is not a leap year.
>python LeapYear2.py
Enter a year: 2004
Year 2004 is a leap year.
>python LeapYear2.py
Enter a year: 2005
Year 2005 is not a leap year.
```

Conditional Expressions

A Python **conditional expression** returns one of two values based on a condition.

Consider the following code:

```
# Set parity according to num
if (num % 2 == 0):
    parity = "even"
else:
    parity = "odd"
```

This sets variable `parity` to one of two values, “even” or “odd”.

An alternative is:

```
parity = "even" if ( num % 2 == 0 ) else "odd"
```

Conditional Expression

General form:

```
expr-1 if boolean-expr else expr-2
```

It means to return `expr-1` if `boolean-expr` evaluates to True, and to return `expr-2` otherwise.

```
# find maximum of x and y
max = x if (x >= y) else y
```

Conditional Expression

Use of conditional expressions can simplify your code.

In file test_sort.py:

```
# Determine if 3 numbers are in sorted ascending order.
def main():
    x = float(input("Enter first number: "))
    y = float(input("Enter second number: "))
    z = float(input("Enter second number: "))
    print('Ascending' if (x <= y) and (y <= z)
            else 'Not Ascending')


main()
```

```
Enter first number: 12
Enter second number: 57
Enter second number: 109
Ascending
```

```
Enter first number: -26.6
Enter second number: 0.72
Enter second number: -12.75
Not Ascending
```

Operator Precedence

Arithmetic expressions in Python attempt to match widely used mathematical rules of precedence. Thus,

$$3 + 4 * (5 + 2)$$

is interpreted as representing:

$$(3 + (4 * (5 + 2))).$$

That is, we perform the operation within parenthesis first, then the multiplication, and finally the addition.

To make this happen we *precedence rules* are enforced.

Precedence

The following are the precedence rules for Python, with items higher in the chart having higher precedence.

Operator	Meaning
+, -	Unary plus, minus, like - 3, +12
**	Exponentiation
not	logical negation
*, /, //, %	Multiplication, division, integer division, modulus
+, -	Binary plus, minus
<, <=, >, >=	Comparison
==, !=	Equal, not equal
and	Conjunction
or	Disjunction

Precedence Examples

```
>>> -3 * 4
-12
>>> - 3 + - 4
-7
>>> 3 + 2 ** 4
19
>>> 4 + 6 < 11 and 3 - 10 < 0
True
>>> 4 < 5 <= 17        # notice special syntax
True
>>> 4 + 5 < 2 + 7
False
>>> 4 + (5 < 2) + 7   # this surprised me!
11
```

Most of the time, the precedence follows what you would expect.

Precedence

Operators on the same line have equal precedence.

Operator	Meaning
`+, -`	Binary plus, minus
`*, /, //, %`	Multiplication, division, integer division, remainder

Evaluate them left to right.

All binary operators are *left associative*. Example: `x + y - z + w` means `((x + y) - z) + w`.

Note that assignment is *right associative*.

```
x = y = z = 1     # assign z first
```

Use Parentheses to Override Precedence

Use parenthesis to override precedence or to make the evaluation clearer.

```
>>> 10 - 8 + 5              # an expression
7
>>> (10 - 8) + 5            # what precedence will do
7
>>> 10 - (8 + 5)            # override precedence
-3
>>> 5 - 3 * 4 / 2           # not particularly clear
-1.0
>>> 5 - ((3 * 4) / 2)       # better
-1.0
```

Work to make your code easy to read!

CS303E: Elements of Computers and Programming

Repitition with Loops

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

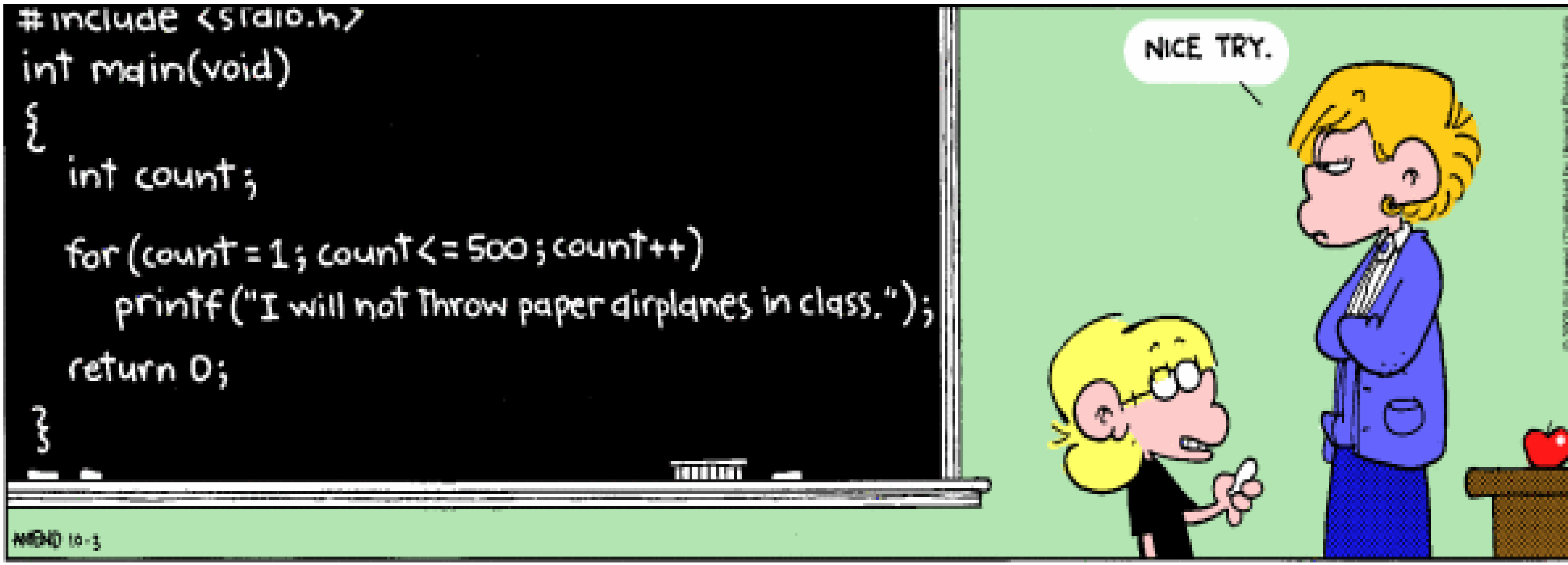
Last updated: May 30, 2024

Repetitive Activity

Often we need to do some (program) activity numerous times:

Using Loops

So we might as well use cleverness to do it.
That's what loops are for.



It doesn't have to be the exact same thing over and over.

And this is how we really harness the power of a computer that can perform tens of billions (or more) computations per second!

While Loop

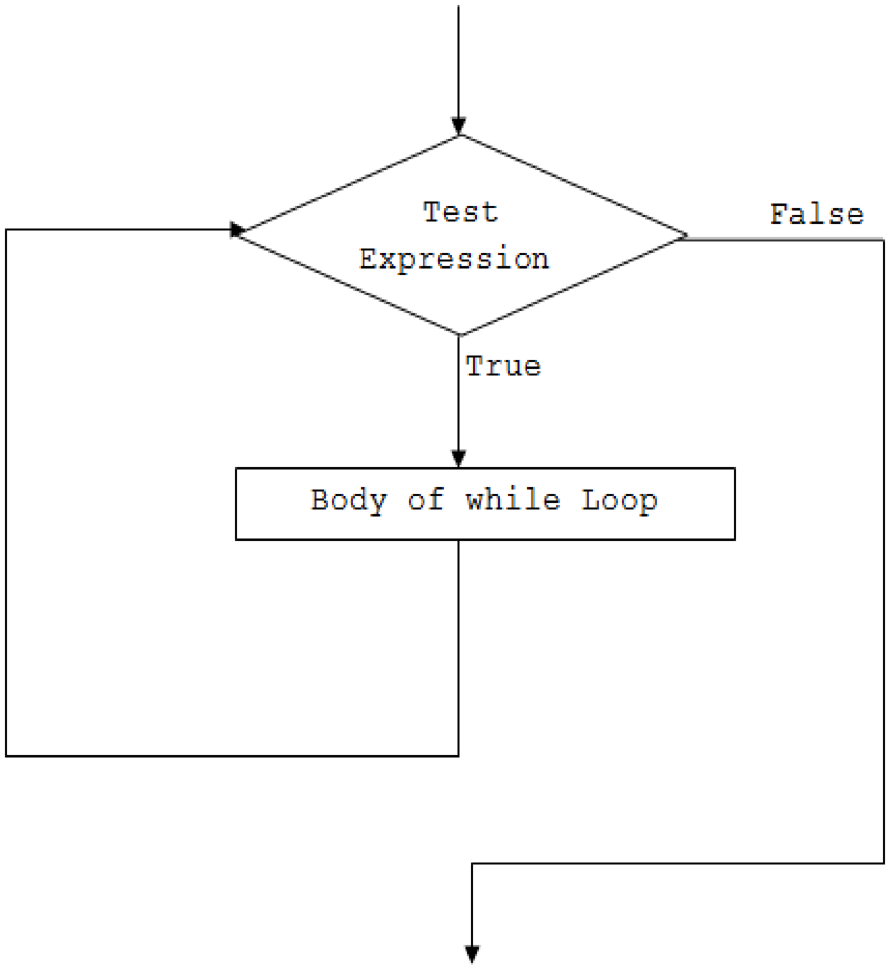
The majority of programming languages include syntax to **repeat** operations.

while loop is one option. General form:

```
while condition:
  statement(s)
```

Meaning: as long as the condition is true when checked, execute the statements.

As with conditionals (if/elif/else), all of the statements in the body of the loop must be indented the same amount.



While Loop

In file not_throw_airplanes.py:

```python
# Print out I will not throw paper airplanes in class
# 500 times.
def main():
    COUNT = 500
    MESSAGE = "I will not throw paper airplanes in class."
    i = 0
    while i < COUNT:
        print(i, MESSAGE)
        i += 1


main()
```

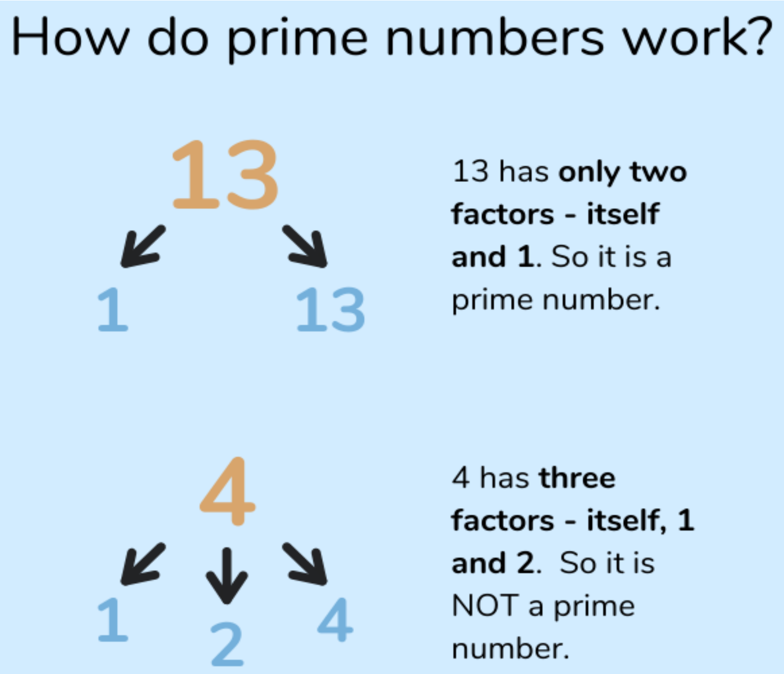
What would happen if we forgot the i += 1?

```
0 I will not throw paper airplanes in class.
1 I will not throw paper airplanes in class.
2 I will not throw paper airplanes in class.
3 I will not throw paper airplanes in class.
4 I will not throw paper airplanes in class.
```

While Loop Example: Test Primality

An integer is prime if it is greater than 1 and has no positive integer divisors except 1 and itself.

To test whether an arbitrary integer n is prime, see if any number in [2 ... n-1], divides it with no remainder.



You couldn't do that in *straight line* code without knowing n in advance. Why not?

Even then it would be *really* tedious if n is very large.

is_prime_1 Loop Example

is_prime_1.py

```python
def main():
    number = int(input("Please enter a number greater than"
                       + " or equal to 2: "))
    prime = True
    divisor = 2
    while divisor < number and prime:
        prime = number % divisor != 0
        divisor += 1
    if prime:
        print(number, "is prime.")
    else:
        print(number, "is not prime.")
    # OR print(number, " is",
    #       "not" if not prime else "", " prime", sep="")


main()
```

is_prime_1 Loop

```
Please enter a number greater than or equal to 2: 37
37 is prime.
```

```
Please enter a number greater than or equal to 2: 176970203
176970203 is prime.
```

The second example took ~24 seconds to complete on my laptop.

It works, though it's pretty inefficient. If a number is prime, we test every possible divisor in `[2 ... n-1]`.

-
- If n is not prime, it will have a divisor less than or equal to $\sqrt{n}$.
- There's no need to test any even divisor except 2.

A Better Version: is_prime_2.py

```python
import math


def main():
    """Determine if a number entered by the user is prime or not."""
    number = int(input("Please enter a number greater than"
                       + " or equal to 2: "))

    # Special case for 2, the only even prime.
    prime = number == 2 or number % 2 != 0
    # If number is not even then we only need to divide
    # by odd numbers.
    divisor = 3

    limit = math.sqrt(number)
    while divisor <= limit and prime:
        prime = number % divisor != 0
        divisor += 1
    if prime:
        print(number, "is prime.")
    else:
        print(number, "is not prime.")
    # OR print(number, " is",
    #        "not" if not prime else "", " prime", sep="")


main()
```

The Better is_prime_2 Version

is_prime_1 does 176,970,202 divisions to discover that 176_970_203 is prime.

is_prime_2 does "only” 13,302.

Took much less than a second to complete.

Computer scientists and software developers spend a lot of time trying to improve the efficiency of their programs and algorithms.

Measurably reduce the number of computations.

Example While Loop: Approximate Square Root

You could approximate the square root of
a positive integer as follows: square_root.py

```
# Approximate the square root of a positive
# integer VERY SLOWLY by increments of 0.1
def main():
    number = int(input("Enter a positive integer: "))
    while number < 0:
        print(number, 'isn\'t a positive int')
        number = int(input("Enter a positive integer: "))
    guess = 0.1
    while guess ** 2 < number:
        guess += 0.1
    print('The square root of', number,
          'is approximately equal to ', guess)


main()
```

Running the Example

```
Enter a positive integer: -37
-37 isn't a positive int
Enter a positive integer: -12
-12 isn't a positive int
Enter a positive integer: -891273
-891273 isn't a positive int
Enter a positive integer: 1_024_237
The square root of 1024237 is approximately equal to  1012.100000001616
```

```
Enter a positive integer: 100
The square root of 100 is approximately equal to  10.09999999999998
```

Notice that the last one isn't quite right. The square root of 100 is exactly 10.0. Foiled again by the approximate nature of floating point numbers and floating point arithmetic.

More efficient way of calculating square root?

Newton's method for approximating square roots adapted from the Dr. Math website

The goal is to find the square root of a number. Let's call it num

1. Choose a rough approximation of the square root of num, call it approx.

   How to choose?

2. Divide num by approx and then average the quotient with approx, in other words we want to evaluate the expression ((num/approx) + approx) / 2

3. How close are we? In programming we would store the result of the expression back into the variable approx.

4. How do you know if you have the right answer?

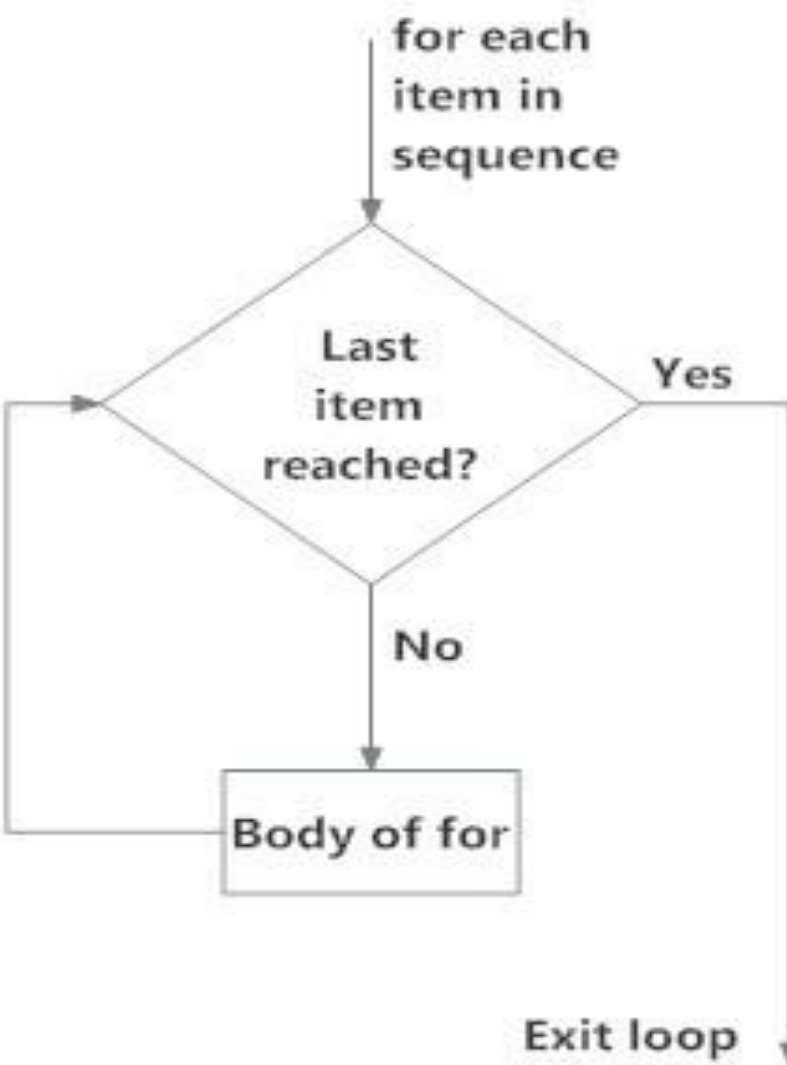
In a `for` loop, you typically know how many times you'll execute.

General form:

```
for <var> in <sequence>:
    <statement(s)>
```

Meaning: assign each element of sequence in turn to var and execute the statements.

As usual, all of the statements in the body must be indented the same amount.



What's a Sequence?

A Python sequence holds multiple items stored one after another.

```
>>> seq = [2, 3, 5, 7, 11, 13]    # a list
```

The range function is a good way to generate a sequence.

range(a, b) : denotes the sequence a, a+1, ..., b-1.

range(b) : is the same as range(0, b).

range(a, b, c) : generates a, a+c, a+2c, ..., b′, where b′ is the last value $<$ b.

Range Examples

```
>>> for i in range(3, 6): print(i, end=" ")
...
3 4 5
>>> for i in range(3): print(i, end=" ")
...
0 1 2
>>> for i in range(0, 11, 3): print(i, end=" ")
...
0 3 6 9
>>> for i in range(11, 0, -3): print(i, end=" ")
...
11 8 5 2
>>>
```

For Loop Example

Suppose you want to print a table of the powers of a given base up to base^n. In file powers_of.py:

```
# Print the powers of a base entered by the user up to
# the nth power, also entered by the user.
def main():
    base = int(input('Enter the base: '))
    max_power = int(input('Enter the maximum power: '))
    for power in range(0, max_power + 1):
        print(base, 'to the', power, 'is',
              base ** power)


main()
```

For Loop Example

```
Enter the base: 2
Enter the maximum power: 42
2 to the 0 is 1
2 to the 1 is 2
2 to the 2 is 4
2 to the 3 is 8
2 to the 4 is 16
2 to the 5 is 32
2 to the 6 is 64
2 to the 7 is 128
2 to the 8 is 256
2 to the 9 is 512
2 to the 10 is 1024
2 to the 11 is 2048
2 to the 12 is 4096
2 to the 13 is 8192
2 to the 14 is 16384
2 to the 15 is 32768
2 to the 16 is 65536
2 to the 17 is 131072
2 to the 18 is 262144
2 to the 19 is 524288
```

```
Enter the base: 1037
Enter the maximum power: 12
1037 to the 0 is 1
1037 to the 1 is 1037
1037 to the 2 is 1075369
1037 to the 3 is 1115157653
1037 to the 4 is 1156418486161
1037 to the 5 is 1199205970148957
1037 to the 6 is 1243576591044468409
1037 to the 7 is 1289588924913113740133
1037 to the 8 is 1337303715134898948517921
1037 to the 9 is 1386783952594890209613084077
1037 to the 10 is 1438094958840901147368768187849
1037 to the 11 is 1491304472318014489821412610799413
1037 to the 12 is 154648273779378102594480487739899128
```

Nested Loops

The body of `while` loops and `for` loops contain any kind of statements, **including other loops.**

Suppose we want to compute and print out the BMI value for heights from 4' 6" (4 feet, 6 inches = 54 inches) to 6' 10" (82 inches) going up by 2 inches each time
AND weights from 85 to 350 pounds, going up by 5 pounds?

We could then take that data and create a visual graph for quick look up.

It is arbitrary whether the *outer loop* is height or weight

Print BMI for various heights and weights

```python
# Print out BMI (Body Mass Index) values for heights from for
# heights from 4' 6" (4 feet, 6 inches = 54 inches)
# to 6' 10" (82 inches) going up by 2 inches each time
# AND weights from 85 to 350 pounds, going up by 5 pounds.
def main():
    english_units_conversion = 703
    for height in range(54, 83, 2):
        print('current height =', height)
        for weight in range(85, 351, 5):
            bmi = english_units_conversion * weight / (height ** 2)

            # Below is an example of the format function.
            # < means left justify
            # 4 means 4 total spots
            # .1 means 1 digit after the decimal
            # f means a floating point number
            print('height =', height, 'weight =', weight,
                  'bmi =', format(bmi, '<4.1f'))
```

CS303E: Elements of Computers and Programming

Functions

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: June 21, 2023

Functions

- We have used several built in functions already:
  - print(), input(), int(), float(), range()
- List of Python built in functions

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Modules - More Functions

- In addition to the standard built in functions.
standard Python includes many modules
  - Modules are Python scripts (programs) that contain, typically, related functions that we can reuse in many Python programs and scripts
- When you download Python, you download the standard modules.
- Most of these modules are beyond the scope of this course.
- Two that we will use are the math module mathematical operations which don't have defined operators and the random module, with functions to generate *pseudo* random numbers.

Function	Description	Example
fabs(x)	Returns the absolute value of the argument.	fabs(-2) is 2
ceil(x)	Rounds x up to its nearest integer and returns this integer.	ceil(2.1) is 3 ceil(-2.1) is -2
floor(x)	Rounds x down to its nearest integer and returns this integer.	floor(2.1) is 2 floor(-2.1) is -3
exp(x)	Returns the exponential function of x (e^x).	exp(1) is 2.71828
log(x)	Returns the natural logarithm of x.	log(2.71828) is 1.0
log(x, base)	Returns the logarithm of x for the specified base.	log10(10, 10) is 1
sqrt(x)	Returns the square root of x.	sqrt(4.0) is 2
sin(x)	Returns the sine of x. x represents an angle in radians.	sin(3.14159 / 2) is 1 sin(3.14159) is 0
asin(x)	Returns the angle in radians for the inverse of sine.	asin(1.0) is 1.57 asin(0.5) is 0.523599
cos(x)	Returns the cosine of x. x represents an angle in radians.	cos(3.14159 / 2) is 0 cos(3.14159) is -1
acos(x)	Returns the angle in radians for the inverse of cosine.	acos(1.0) is 0 acos(0.5) is 1.0472
tan(x)	Returns the tangent of x. x represents an angle in radians.	tan(3.14159 / 4) is 1 tan(0.0) is 0
fmod(x, y)	Returns the remainder of x/y as double.	fmod(2.4, 1.3) is 1.1
degrees(x)	Converts angle x from radians to degrees	degrees(1.57) is 90
radians(x)	Converts angle x from degrees to radians	radians(90) is 1.57

Importing Modules

- To use non standard functions, ones that are part of a module, we call the function with the name of the module, a period *spoken "dot"*, and the name of the function. math.sqrt(1000)

```
>>> math.sqrt(1000)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'math' is not defined
```

- must also import the module

```
>>> import math
>>> math.sqrt(1000)
31.622776601683793
```

- In a program or script, imports at the top of the file.

The random Module

- Several useful functions are defined in the random module:
- **`randint(a, b)`**: generate a random integer between a and b, inclusively.
- **`randrange(a, b)`**: generate a random integer between a and b-1, inclusively.
- **`random()`**: generate a float in the range [0 . . . 1).
- How would we simulate flipping a coin with two sides?

Examples of Calls to random Functions

```
>>> import random
>>> random.randint(1, 2)
2
>>> random.randint(1, 2)
2
>>> random.randint(1, 2)
2
>>> random.randint(1, 2)
1
>>> random.randint(1, 2)
1
>>> random.randint(1, 6)
6
>>> random.randint(1, 6)
4
>>> random.randint(1, 6)
3
>>> random.randrange(1, 2)
1
>>> random.randrange(1, 2)
1
>>> random.randrange(1, 2)
1
>>> random.randrange(1, 2)
>>>
```

```
>>> random.randrange(1, 3)
1
>>> random.randrange(1, 3)
2
>>> random.random()
0.877326549191274 5
>>> random.random()
0.616574268416400 1
>>> random.random()
0.927352470189636 5
>>> random.random()
0.1385262793329998 8
>>> random.random()
0.66413228194997 3
>>> for i in range(0, 10):
...     print(random.randint(1, 100))
...
63
51
43
87
60
51
33
26
```

Importing Modules

- Typing the name of the module every time can be tedious
  - A lot of programming languages and IDEs have features to reduce the amount of typing we have to do
- Can import specific or all functions from a module:

```
>>> from random import randint
>>> randint(1, 100)
78
>>> randint(1, 10)
8
>>> random()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'module' object is not callable
>>> from random import *
>>> random()
0.0699097275883659
```

The * is a wildcard, meaning all.

- **Any downside to always importing all?**

Three Common Data Types

Three data types we will use in many of our early Python programs are:

`int`: signed integers (whole numbers)

- Computations are exact and *of unlimited size*
- Examples: 4, -17, 0

`float`: signed real numbers (numbers with decimal points) Large

- range, but fixed precision
- Computations are approximate, not exact Examples:
- 3.2, -9.0, 3.5e7

`str`: represents text (a string)

- We use it for input and output We'll see
- more uses later Examples: "Hello, World!",
- 'abc'

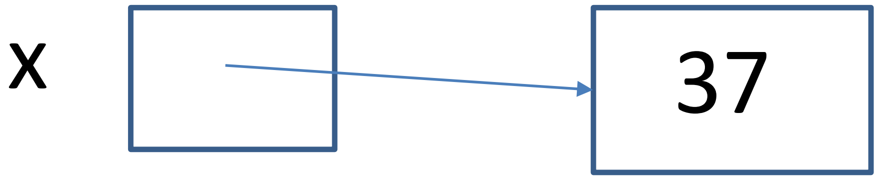
These are all *immutable.* The value cannot be altered.

Immutable

- It may appear some values are mutable
  - they are not
  - rather variables are mutable and can be bound (refer to) different values
- Note, how the id of x (*similar to its address)* has changed

```
>>> x = 37
>>> x
37
>>> id(x)
140711339416352
>>> x = x + 10
>>> x
47
>>> id(x)
140711339416672
```

x = 37



x = x + 10

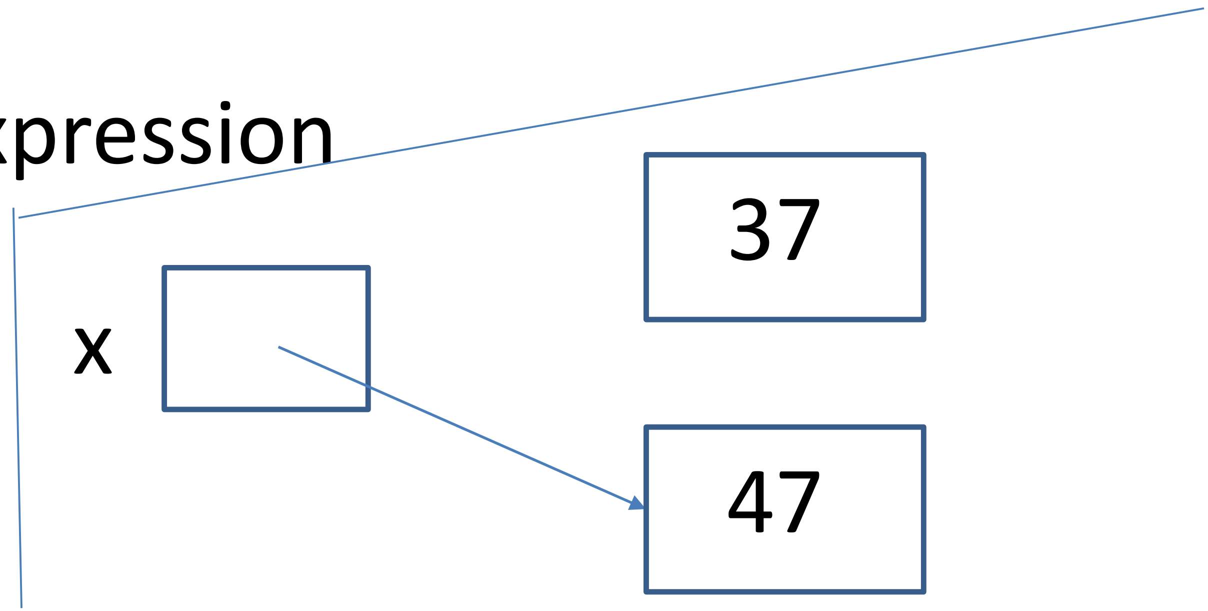
substitute in the value x is referring to

x = 37 + 10

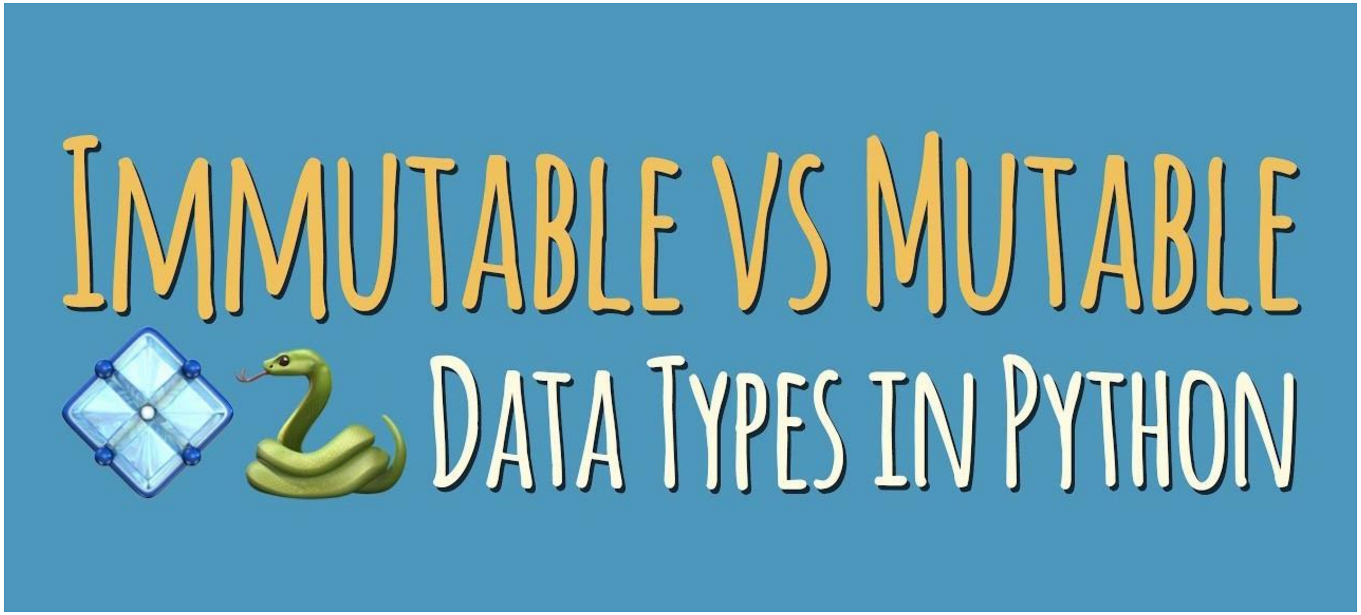
evaluate the expression

x = 47

so now …



Mutable vs. Immutable



An **immutable** value is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.

A **mutable** values is one that can be changed; e.g., sets, lists, etc.

What Immutable Means

- An **immutable** object is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.
- It also means that *there is typically only one copy of the object in memory.*
- Whenever the system encounters a new reference to 17, say, it creates a pointer (references) to the already stored value 17.
- Every reference to 17 is actually a pointer to the *only* copy of 17 in memory. Ditto for "abc".
- If you do something to the object that yields a new value (e.g., uppercase a string), you're actually creating a new object, not changing the existing one.

Function

We've seen lots of system-defined functions;
now it's time to define our own., like main.

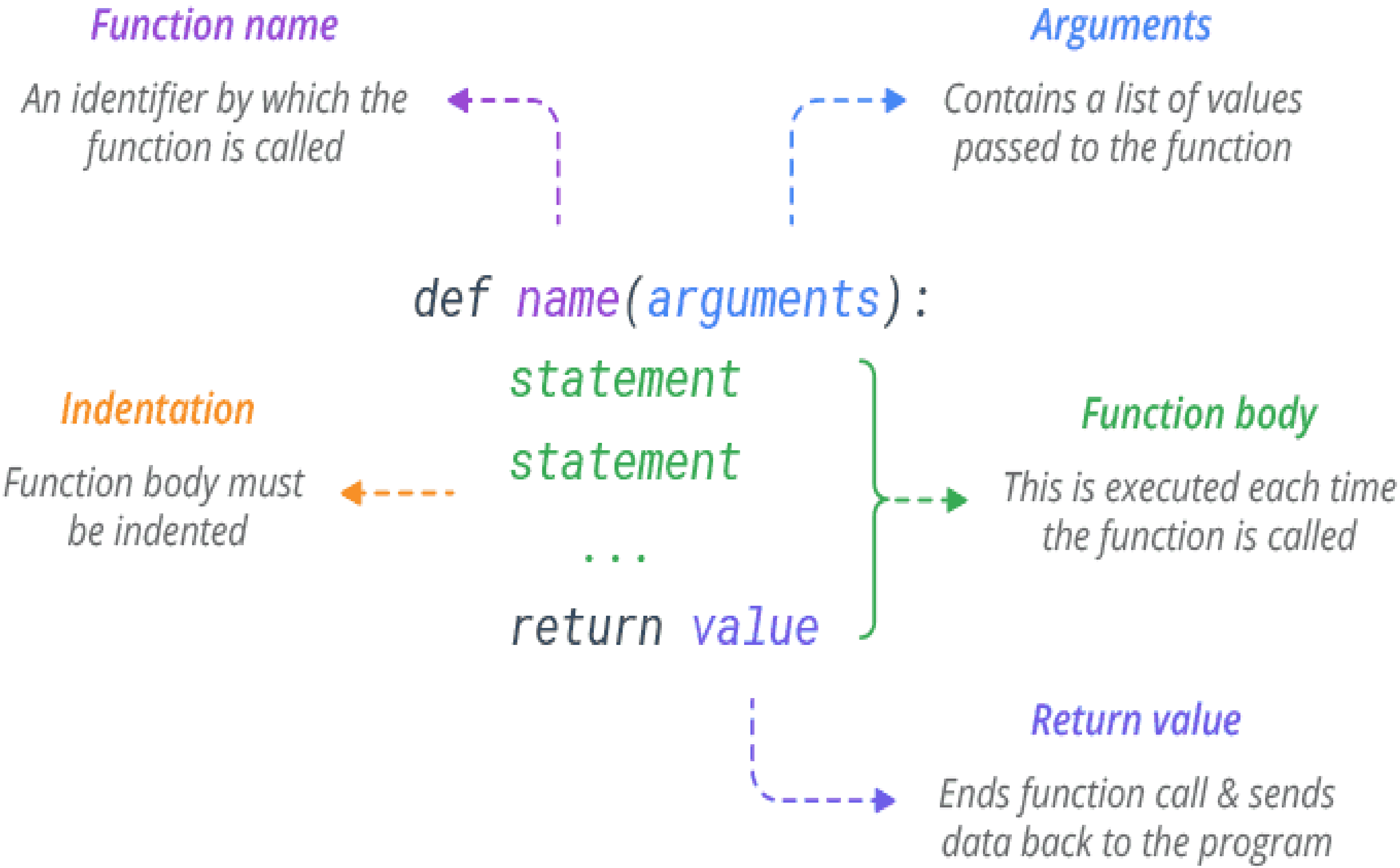
General form:

```
def functionName( list of parameters ):
                                          #
    header statement(s)                   # body
```

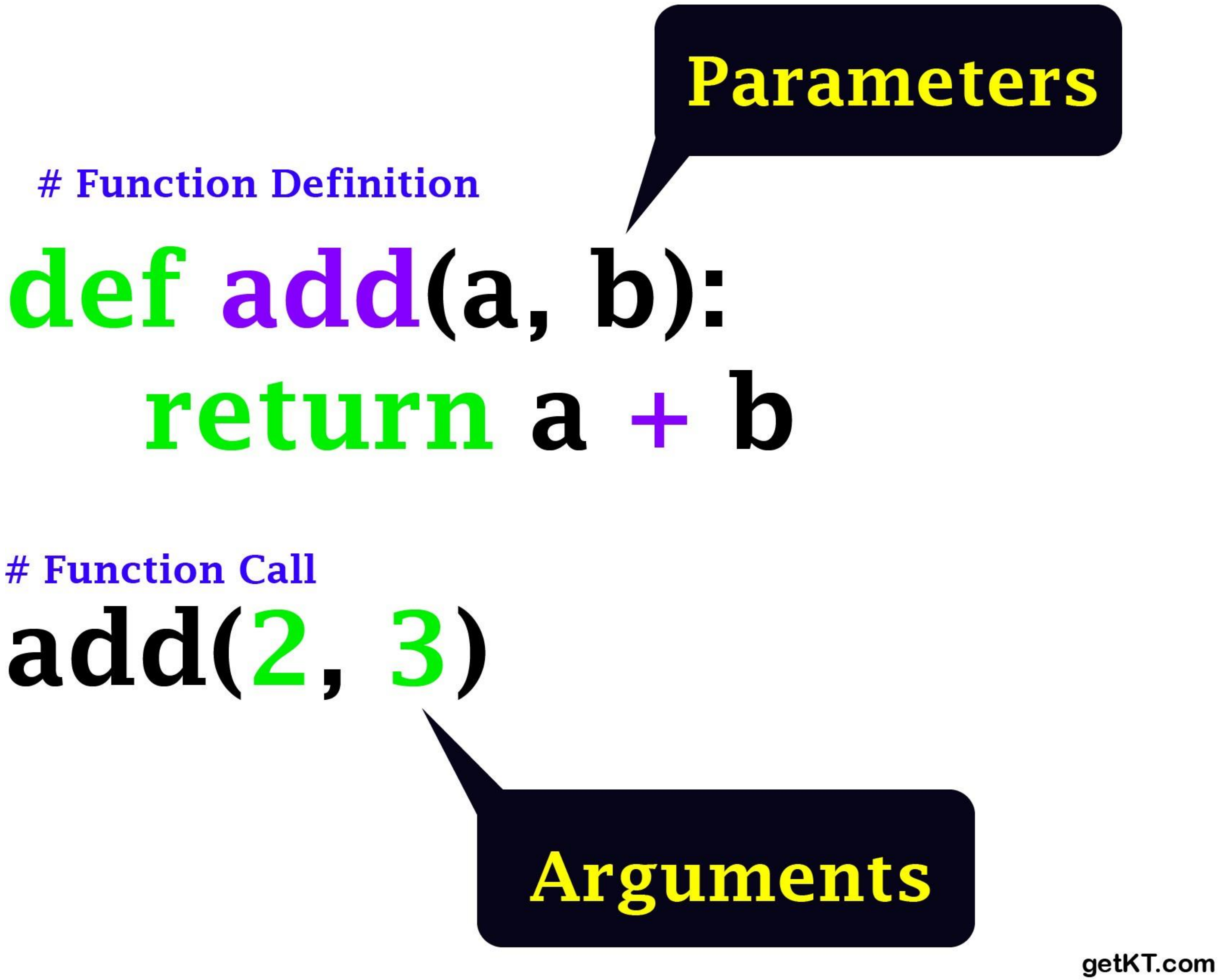
Meaning: a function definition defines a block of code that performs a specific task. It can reference any of the variables in the list of parameters. It may or may not return a value.

The parameters are **formal parameters**;
they hold arguments (refer to the same values) passed to the function later when the function is *called.*

Functions


Function name
An identifier by which the function is called
Arguments
Contains a list of values passed to the function
def name(arguments):
statement
statement
...
return value
Indentation
Function body must be indented
Function body
This is executed each time the function is called
Return value
Ends function call & sends data back to the program

Calling a Function



Function Example

Suppose you want to sum the integers 1 to n.

In file function_examples.py:

```
# Return the sum of values from 1 to n.
# This is an example of a cumulative sum algorithm.
def sum_to_n(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
```

Notice this defines a *function* to perform the task, but *won't perform the task until the function is called from else where.*
We still have to call/invoke the function with specific arguments.

```
def main():
    print(sum_to_n(1))
    print(sum_to_n(1000))
```

```
1
500500

Process finished with exit code
```

Some Observations

```
def sum_to_n(n)          # function header
    ....                 # function body
```

Here n is a *formal parameter*. It is used in the definition as a place holder for an *actual parameter* (e.g., 10 or 1000) in any specific call.

sum_to_n(n) *returns* an int value, meaning that a call to sum_to_n can be used anyplace an int expression can be used.

```
x = sum_to_n(30)
print(x)
print('Even' if sum_to_n(5) % 2 == 0 else 'Odd')
for i in range(1, 30):
    print(i, sum_to_n(i))
```

Note, with functions the argument is the input.
We occasionally ask the user for input in the function.

Once we've defined sum_to_n, we can use it almost as if were a primitive in the language without worry about the details of the definition.

We need to know what it does,
but don't care anymore how it does it!

This is called **information hiding** and / or **functional abstraction**.

And that is **POWERFUL!**

Another Way to Add Integers 1 to N

Suppose later we discover that we could have coded sumToN more efficiently (as discovered by the 8-year old C.F. Gauss in 1785):

```
# Efficient implementation of summing the values
# from 1 to n. We assume n >= 1
def sum_to_n(n):
    return (n + 1) * n // 2
```

Because we defined sum_to_n *as a function, we can just swap in this definition without changing any other code.* If we'd done the implementation in-line, we'd have had to go find every instance and change it.

When you execute a `return` statement, you return to the calling environment. Your functions may or may not explicitly return a value.

General forms:

```
return
return expression
```

A `return` that doesn't return a value actually returns the constant `None`. ***Use return without a value sparingly.***

Every function has an *implicit* return at the end.

```python
# Demonstrate the implicit return in functions even
# if no return written.
def print_x(x):
    print(x)
```

```python
print(print_x(73))
```

→

```
73
None
```

Some More Function Examples

Suppose we want to multiply the integers from 1 to n:

```
# Return the result of multiply the values from
# 1 to n. This is the factorial function. We assume n >= 0
def multiply_to_n(n):
    result = 1
    for i in range(2, n + 1):
        result *= result
```

Convert Fahrenheit to Celsius AND Celsius to Fahrenheit :

```
# Convert degrees fahrenheit to degrees celsius.
def fahrenheit_to_celsius(degrees_f):
    return 5 / 9 * (degrees_f - 32)


# Convert degrees celsius to degrees fahrenheit.
def celsius_to_fahrenheit(degrees_c):
    return 1.8 * degrees_c + 32
```

Fahr to Celsius Table

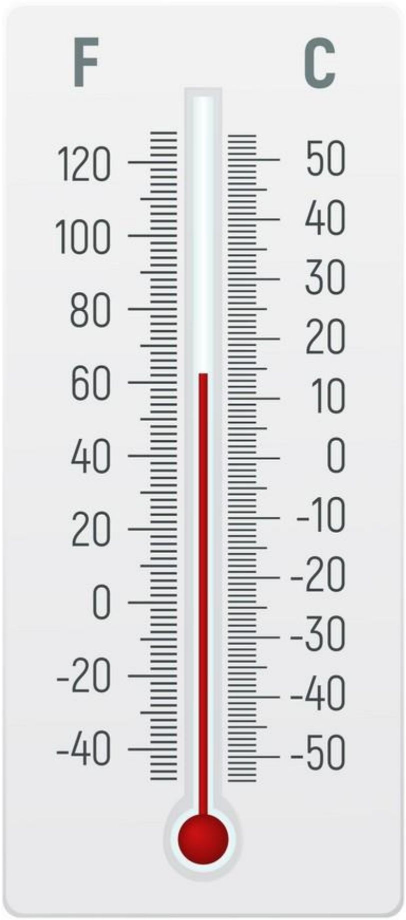
In slideset 1, we showed the C version of a program to print a table of Fahrenheit to Celsius values. Here's a Python version:

In file fahr_to_celsius_table.py:

```
from function_examples import fahrenheit_to_celsius


# Print the table.
def main():
    lower_temp = -50
    upper_temp = 250
    step = 10
    # If the loop variable has meaning beyond a simple
    # counter, okay to name it something other than i, k, j.
    for degrees_f in range(lower_temp, upper_temp + 1, step):
        degrees_c = fahrenheit_to_celsius(degrees_f)
        print(format(degrees_f, "3d"), '\t',
              format(degrees_c, "5.1f"))


main()
```

Running the Temperature Program

```
-50       -45.6
-40       -40.0
-30       -34.4
-20       -28.9
-10       -23.3
  0       -17.8
 10       -12.2
 20        -6.7
 30        -1.1
 40         4.4
 50        10.0
 60        15.6
 70        21.1
 80        26.7
 90        32.2
100        37.8
110        43.3
120        48.9
```



Exercise: Do a similar problem converting Celsius to Fahrenheit.

A Bigger Example: Print First 100 Primes

Suppose you want to print out a table of the first 100 primes, 10 per line.

You could sit down and write this program from scratch, without using functions. But it would be a complicated mess (see section 5.8).

Better to use **functional abstraction**: find parts of the algorithm that can be coded separately and "packaged" as functions.

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541

Print First 100 Primes: Algorithm

Here's some Python-like pseudocode to print 100 primes:

```
def print100Primes():
    primeCount = 0
    num = 0
    while (primeCount < 100):
        if (we've already printed 10 on the current line):
            go to a new line
        nextPrime = ( the next prime > num)
        print nextPrime on the current line
        num = nextPrime
        primeCount += 1
```

Note that most of this is just straightforward Python programming! The only "new" part is how to find the next prime. So we'll make that a *function*.

So let's *assume* we can define a function:

```
# Return the first prime larger than n.
def get_next_prime(n):
```

in such a way that it returns the first prime larger than num.

Is that even possible?

Is there always a "next" prime larger than num?

Yes! There are an infinite number of primes. So if we keep testing successive numbers starting at num+ 1, we'll eventually find the next prime. *That may not be the most efficient way!*

Value of Functional Abstraction

Notice we're following a "divide and conquer" approach: Reduce the solution of our bigger problem into one or more subproblems which we can tackle independently.

It's also an instance of "information hiding." We don't want to think about how to find the next prime, while we're worrying about printing 100 primes. Put that off! Think about one thing at a time. ***Structural decomposition.***

Next Step

Now solve the original problem, *assuming* we can write get_next_prime(n)

In file function_examples.py:

```
# Print a table of the first n primes
# 10 per line. We expect n >= 1
def print_prime_table(n):
    current_num = 1
    for i in range(1, n + 1):
        current_num = get_next_prime(current_num)
        print(format(current_num, '5d'), end=' ')
        # go to next line after every ten primes
        if i % 10 == 0:
            print()
    print()
```

Looking Ahead

Here's what the output should look like.

```
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229
  233  239  241  251  257  263  269  271  277  281
  283  293  307  311  313  317  331  337  347  349
  353  359  367  373  379  383  389  397  401  409
  419  421  431  433  439  443  449  457  461  463
  467  479  487  491  499  503  509  521  523  541
```

Of course, we couldn't do this if we really hadn't defined get_next_prime. So let's see what that looks like.

How to Find the Next Prime

The next prime (> num) can be found as indicated in the following pseudocode:

```
def get_next_prime(num):
    if num< 2:
        return 2 as the answer
    else:
        guess = num+ 1
        while (guess is not prime)
            guess += 1
        return guess as the answer
```

Again we solved one problem by assuming the solution to another problem: deciding whether a number is prime.

Can you think of ways to improve this algorithm?

Here's the Implementation

Note that we're assuming we can write:

```
# We assume n >= 2. Return True if n is prime,
# False otherwise.
def is_prime(n):
```

```
# Return the first prime larger than n.
def get_next_prime(n):
    if n < 2:
        return 2
    guess = n + 1
    while not is_prime(guess):
        guess += 1
    return guess
```

This works (assuming we have defined is_prime), but it's got an inefficiency. How can we make it more efficient?

Find Next Prime: A Better Version

When looking for the next prime, we don't have to test every number, just the odd numbers (after 2).

```python
# Return the first prime larger than n.
def get_next_prime(n):
    if n < 2:
        return 2
    # We know n >= 2 and that no even integers
    # greater than 2 are prime. So go to the next
    # odd number and only check odd numbers.
    guess = n + 1 if n % 2 == 0 else n + 2

    # OR maybe more clearly
    # guess = n + 1
    # if guess % 2 == 0:
    #       guess = guess + 1

    while not is_prime(guess):
        guess += 2
    return guess
```

Now all that remains is to write is_prime.

Is a Number Prime?

We already solved a version of this in a previous lecture.
Let's rewrite that code as a Boolean-valued function:

```python
# We assume n >= 2. Return True if n is prime,
# False otherwise.
def is_prime(n):
    # Special case for 2, the only even prime.
    if n == 2:
        return True
    # Check if there are any odd divisors
    # up to the square root of the number.
    prime = n % 2 != 0
    divisor = 3
    limit = math.sqrt(n)
    while divisor <= limit and prime:
        prime = n % divisor != 0
        divisor += 2
    return prime
```

Sidetrack - Boolean "Zen"

- Did you notice this line of code in the is_prime method?

```
return prime
```

- prime is a boolean that holds the value True of False, so we simply return than value in that variable

- avoid the following: it is unnecessarily verbose

```
# YUCK!!!!
if prime == True:
    return True
else:
    return False
```

One More Example

Suppose we want to find and print k primes, starting from a given number:

In file function_examples.py:

```
# Print the first num primes after the
# value start. One prime per line.
def print_num_primes_staring_from(num, start):
    if num == 0:
        print("Request was for 0 primes")
    else:
        print('First', num, 'primes after', start, '.')
        current = start
        for i in range(num):
            current = get_next_prime(current)
            print((i + 1), current)
```

Notice that we can use functions we've defined such as get_next_prime and is_prime (almost) *as if* they were Python primitives.

Positional Arguments

This function has four formal parameters:

```
# Demo of positional arguments.
def some_function(x1, x2, x3, x4):
```

Any call to this function should have exactly four actual arguments, which are matched to the corresponding formal parameters:

```
some_function(5, 12, 5, 13)
x = 3
y = -5
some_function(x, y + 2, x * y, 12)
```

This is called using **positional** arguments.

Keyword Arguments

It is also possible to use the formal parameters as **keywords**.

```
# Demo of positional arguments.
def some_function(x1, x2, x3, x4):
    print('In some_function')
    print(x1, x2, x3, x4)
```

These two calls are equivalent:

```
some_function(5, 12, -7, 13)
some_function(x3=-7, x1=5, x4=13, x2=12)
```

```
In some_function
5 12 -7 13
In some_function
5 12 -7 13
```

Keyword Arguments

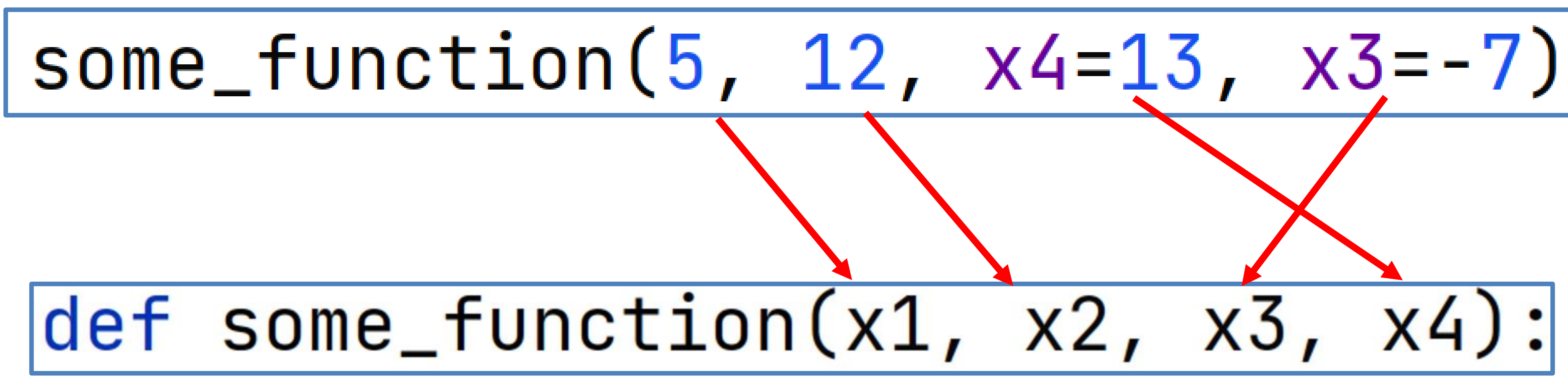
You can list the keyword arguments in any order, but all must still be specified.

```
some_function(x3=12, x1=12)
```

```
Traceback (most recent call last):
  File "C:/Users/scottm/PycharmProjects/AssignnmentSolutions/SlidesCode/function_
    main()
  File "C:/Users/scottm/PycharmProjects/AssignnmentSolutions/SlidesCode/function_
    some_function(x3=12, x1=12)
TypeError: some_function() missing 2 required positional arguments: 'x2' and 'x4'
```

And even possible to mix keyword arguments with positional arguments.

The positional arguments must come first followed by the keyword.



Default Parameters

You can also specify **default arguments** for a function. If you don't specify a corresponding actual argument, the default is used.

```
# Demonstrate a default argument for a parameter.
def print_rectangle_area(width=1.0, height=2.0):
    area = width * height
    print('A rectangle with a width of', width,
          'and a height of', height,
          'has an area equal to', area)
```

```
print_rectangle_area()  # uses default arguments
print_rectangle_area(4.5, 7.6)  # uses positional arguments
print_rectangle_area(height=20.5, width=5.2)  # uses keyword arguments
print_rectangle_area(4.5)  # default height
print_rectangle_area(height=10.0)  # default width
print_rectangle_area(width=5.25)  # default height
```

Do any of the built in functions we have been using have default arguments?

Using Defaults

```
A rectangle with a width of 1.0 and a height of 2.0 has an area equal to 2.0
A rectangle with a width of 4.5 and a height of 7.6 has an area equal to 34.199
A rectangle with a width of 5.2 and a height of 20.5 has an area equal to 106.6
A rectangle with a width of 4.5 and a height of 2.0 has an area equal to 9.0
A rectangle with a width of 1.0 and a height of 10.0 has an area equal to 10.0
A rectangle with a width of 5.25 and a height of 2.0 has an area equal to 10.5
```

You can mix default and non-default arguments, but must define the non-default arguments first.

```
def email(address, message=''):
```

Passing by Reference

All values in Python are objects, including numbers, strings, etc.

When you pass an argument to a function, you're actually passing a **reference** to the object, not the object itself.

There are two kinds of objects in Python:

mutable: you can change them in your program.

immutable: you can't change them in your program.

If you pass a reference to a mutable object, it can be changed by your function. If you pass a reference to an immutable object, it can't be changed by your function.

What is a Data Type?

A **data type** is a categorization of values.

Data Type	Description	Example
int	integer. An immutable number of unlimited magnitude	42
float	A real number. An immutable floating point number, system defined precision	3.1415927
str	string. An immutable sequence of characters	'Wikipedia'
bool	boolean. An immutable truth value	True, False
tuple	Immutable sequence of mixed types.	(4.0, 'UT', True)
list	Mutable sequence of mixed types.	[12, 3, 12, 7, 6]
set	Mutable, unordered collection, no duplicates	[12, 6, 3]
dict	dictionary a.k.a. maps, A mutable group of (key, value pairs)	{'k1': 2.5, 'k2': 5}

Others we likely won't use in 303e:
complex, bytes, frozenset

Passing an Immutable Object

Consider the following code:

```
def increment_x(x):
    x += 1
    print('Value of x in the function increment_x =', x)


def reverse_list(lst):
    lst.reverse()
    print('list in the function reverse_list =', lst)
```

```
print()
x = 3
print('x before function call:', x)
increment_x(x)
print('x after function call: ', x)
print()

lst = [2, 3, 5, 7, 11]
print('list before function call:', lst)
reverse_list(lst)
print('list after function call: ', lst)
```

Passing Immutable and Mutable Objects - Output

```
x before function call: 3
Value of x in the function increment_x = 4
x after function call:  3

list before function call: [2, 3, 5, 7, 11]
list in the function reverse_list = [11, 7, 5, 3, 2]
list after function call:  [11, 7, 5, 3, 2]
```

Notice that the immutable integer parameter to increment_x was unchanged, while the mutable list parameter to reverse_list was changed.

Variables are mutable. They can be made to refer to different objects (values), but some objects (values) such as ints, floats, and Strings in Python are immutable.

Scope of Variables

Variables defined in a Python program have an associated **scope**, meaning the portion of the program in which they are defined.

A **global variable** is defined outside of a function and is visible after it is defined. *Use of global variables is generally considered bad programming practice.*
Not allowed per our 303e program hygiene guidelines.

A **local variable** is defined within a function and is visible from the definition until the end of the function.

A local definition overrides a global definition.

Overriding

A local definition (locally) overrides the global definition.

```
x = 1                        # x is global

def func():
    x = 2                    # this x is local
    print(x)                 # will print 2

func()
print(x)                     # will print 1
```

Running the program:

```
> python funcy.py
2
1
```

Returning Multiple Values - Useful

The Python `return` statement can also return multiple values. In fact it returns a *tuple* of values.

```
def multipleValues( x, y ):
    return x + 1, y + 1

print( "Values returned are: ", multipleValues( 4, 5.2 ))

x1, x2 = multipleValues( 4, 5.2 )
print( "x1: ", x1, "\tx2: ", x2 )
```

```
Values returned are: (5, 6.2)
x1: 5   x2: 6.2
```

You can operate on this using tuple functions, which we'll cover later in the semester, or assign them to variables.

CS303E: Elements of Computers and Programming

Files

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: June 23, 2022

Value of Files

Files are a persistent way to store programs, input data, and output data.

Files are stored in the memory of your computer in an area allocated to the *file system*, which is typically arranged into a hierarchy of *directories (aka folders)*.

The ***path*** to a particular file details where the file is stored within this hierarchy.

Relative Pathnames

A path to a file may be *absolute* or *relative*.

If you just use the name of the file, you're assuming that it is in the current working directory.

```
plato% pwd
/u/scottm/314
plato% ls -l
total 8
drwx------ 17 scottm prof 4096 Sep 14  2020 grade
-rw-r--r--  1 scottm prof   42 Nov 25  2019 nums
-rw-r--r--  1 scottm prof   42 May  4 11:28 nums_sorted
-rw-r--r--  1 scottm prof   58 Nov 25  2019 simple.txt
drwx------  2 scottm prof 4096 Aug 19  2020 src
```

pwd -> print working directory

ls -l -> list the contents of the current directory in long form (with details)

Relative Pathnames

```
drwx------ 17 scottm prof 4096 Sep 14  2020 grade
-rw-r--r--  1 scottm prof   42 Nov 25  2019 nums
-rw-r--r--  1 scottm prof   42 May  4 11:28 nums_sorted
-rw-r--r--  1 scottm prof   58 Nov 25  2019 simple.txt
drwx------  2 scottm prof 4096 Aug 19  2020 src
plato% cat calculate_taxes.py
cat: calculate_taxes.py: No such file or directory
plato% cat src/calculate_taxes.py
def main():
    """Calculate US Federal Income Tax.

    Ask user for income and calculate US
    Federal income tax for 2021.
    Assumes user is filing single.
```

cat -> from con**cat**enate, synonym for append (in this case to standard output)

src/ means look for the file in the directory named src

File Paths

On Windows, a file path might be:

C:\Users\scottm\314\src\calculate_texas.py

On Linux or MacOS, it might be:

/home/scottm/314/src/calculate_texas.py

Python passes filenames around as strings, which causes some problems for Windows systems, partly because Windows uses the '\' in filepaths.

Recall that backslash is an escape character, and including it in a string may require escaping it.

Raw Strings

There is a way in Python to treat a string as a **raw string**, meaning that escaped characters are treated just as any other characters.

```
>>> print('abc\ndef')
abc
def
>>> print(r'abc\ndef')
abc\ndef
```

Prefix the string with an 'r'. You may or may not need to do the for Windows pathnames including '\'

In CS303e when we open a file we will generally assume it is in the same directory as the running Python program.

When doing homework, how do you know what that is so you can put your data files in the same directory?

```
import os
print(os.getcwd())
```

```
print(os.getcwd()) # os already imported above
```

```
C:\Users\scottm\Documents\303e\_Su21\lecture_code\examples
```

Of course your output will be different.

Python provides a simple, elegant interface to storing and retrieving data in files.

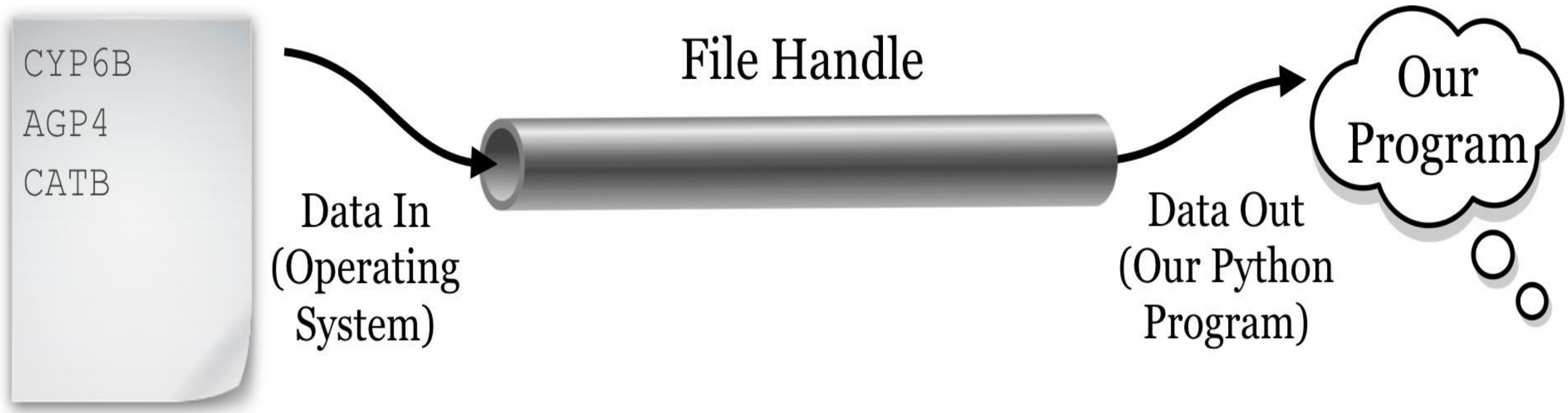
Functions for dealing with files:

open : establish a connection to the file and associate a local file *handle* with a physical file.

close : terminate the connection to the file.

Opening a File

Before your program can access the data in a file, it is necessary to *open* it. This returns a *file object*, also called a 'handle,' that you can use within your program to access the file.



It also informs the system how you intend for your program to interact with the file, the 'mode.'

General Form:

fileVariable = open(filename, mode)

```
>>> outfile = open('test_file.txt', 'w')
>>> outfile.write('Testing can show the presence of bugs ...\n')
42
>>> outfile.write('but not prove their absence.\n')
29
>>> outfile.close()
```

What do you think the 42 and 29 (an int returned by the write function) represent above?

Notice we are calling a function (method) on a variable.
outfile.write

```
(lecture_code) C:\Users\scottm\Documents\303e\_Su21\lecture_code>type test_file.txt
Testing can show the presence of bugs ...
but not prove their absence.
```

Opening a File: Modes

Permissible modes for files:

Mode	Description
'r'	Open for reading.
'w'	Open for writing. **If the file already exists the old contents are overwritten.**
'a'	Open for appending data to the end of the file.
'rb'	Open for reading binary data.
'wb'	Open for writing binary data.

You also have to have necessary permissions from the operating system to access the files.

This semester we won't be using the binary modes.

In other words we are going to read from files assuming it is encoded as text. In binary we would read the raw 0s and 1s.

Closing the File

General form:

```
file_variable.close()
```

All files are closed by the OS when your program terminates. Still, it is very important to close any file you open in Python.

- the file will be locked from access by any other program while you have it open;
- items you write to the file may be held in internal buffers rather than written to the physical file;
- if you have a file open for writing, you can't read it until you close it, and re-open for reading;
- *it's just good programming practice*.

Using the with statement

Although not in the textbook, the preferred way of opening a file is with the **with** statement. (Another Python keyword)

```
def demo_with(file_name):
    """Demonstrate creating file objects with the with keyword."""
    with open(file_name, 'r') as in_file:
        # Simply print the lines in the file
        for line in in_file:
            print(line, end='')
        print('Still in with. Is file closed? ',
              in_file.closed)
    # Is the file closed?
    print('After with block. Is file closed? ', in_file.closed)
```

```
Still in with. Is file closed?  False
After with block. Is file closed?  True
```

There are various Python functions for reading data from or writing data to a file, given the file object in variable fn.

Function	Description
fn.read()	Return entire remaining contents of file as a string.
fn.read(k)	Return next k characters from the file as a string.
fn.readline()	Returns the next line as a string.
fn.readlines()	Returns all remaining lines in the file as a list of strings.
fn.write(str)	Writes the string to the file.

These functions advance an internal *file pointer (like a cursor in a word processing document or a program editor)* that indicates where in the file you're reading/writing.

open sets the file pointer or cursor at the beginning of the file.

Testing File Existence

Sometimes you need to know whether a file exists, otherwise you may overwrite an existing file.

Use the `isfile` function from the `os.path` module.

```
>>> isfile('foo.txt')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'isfile' is not defined
>>> import os.path
>>> os.path.isfile('foo.txt')
False
>>> os.path.isfile('test_file.txt')
True
```

Here the filepath given is *relative* to the current directory.

Example: Read Lines from File

```
import os.path


def main():
    """Open the file. Print out all lines with a line number."""
    file_name = input('Enter file name: ')
    if not os.path.isfile(file_name):
        print(file_name, 'does not exist in the current directory.')
    else:
        file = open(file_name, 'r')
        line = file.readline()
        line_number = 0

        # Print out lines of file with line numbers.
        while line:
            line_number += 1
            print(format(line_number, '4d'), ': ',
                  line.strip(), sep='')
            line = file.readline()
        print('Found', line_number, 'lines.')
        print('Value of line that caused loop to stop:', line)
        file.close()
```

Example: Read Lines from File

```
Enter file name: lyrics.txt
   1: That's great, it starts with an earthquake
   2: Birds and snakes, and aeroplanes
   3: And Lenny Bruce is not afraid
   4:
   5: Eye of a hurricane, listen to yourself churn
```

...

```
  66: It's the end of the world as we know it (tim
  67: It's the end of the world as we know it (tim
  68: It's the end of the world as we know it and
Found 68 lines.
Value of line that caused loop to stop:
```

Example: Write File

Let's write out the flip of 10,000 coins to a file, H for heads, T for tails. 50 results per line separated by a space.

One major difference is that print inserts a newline at the end of each line, unless you ask it not to. write does not do that.

```
# Write out the results of coin flips to a file.
import random


def main():
    num_flips = 10_000
    flips_per_line = 50
    out_file = open('flip_results.txt', 'w')
    for i in range(1, num_flips + 1):
        side = 'H' if random.random() < 0.5 else 'T'
        out_file.write(side)
        if i % flips_per_line == 0:
            out_file.write('\n')
    out_file.close()
```

Part of Resulting File - Coin Flip Results

```
THTHHTHHHHTTHHHHHHHHHHTHTHTTHHTTTHHTTTTTTTTTTTTTHH
HHTHTTHTTHTTTHHTTTTHTHTHHTHTHHHHTHHTHHTTTHHHHTTHTH
HHTHTTHHTTHTTTTTHTTTHTHTHHHTHHHHTHTTHHHTTHHHTTHHH
HHTHTHTHTHHHHTHTHTTHTHHHTTHTTHTTHTTTHTHTHTHTTHHHHH
THTTTTTHTTHHTHHHHHTHHHHTTTHHHTTHHHTTHHHTHHTTTHTHTH
HTHHTHTHHTHHTTHTHTTHHTTHHTHTTHTTHTHTTTHTHHHTTHTTTT
HTTHHHTHTHTTHTHTTTTHHHHHTTTHTTHHTTTHHTTHHHTHHHHHH
THHTTHHTTTTTHHTHHTHHHTHTTHTTTHHTTTHHTTHTHTHTTTTHHT
TTTHTHHTHHHTTHTTTTTHTHHTTHTHTTTTTTTTTHHHHHTHHHHTTHT
TTTTTTTTHHHTHHHHTTHTHTHHHTHTTTTTHTHTHTHHHHHHHTHTTT
THTHTHTTHHHHHHHHHTHTTHTTHHTHTHHTHTTHTHHTTHHHHHHTHH
HHHHTHHTHHHHTTHHHHTHTTHHHHTHTHHTTHTHTTTHHTHHHTTTT
THTTHHHTTHTTTHHTTHTTTHHHTHHHTTHHHTTHTTTTTTHTTTTHHT
THTTHTHHTTHTTTTHHHHTTHTHHHHTHTTTTHHHHHTTHHTHHHTHTH
TTTHHHTHTHHTTTHHTTHTHTHHTTTHTTHTHTHHTHHTTHTTTHHHHH
HHHTHTHTHHHHHTHTHHTHHTHTHTHTTHTHTHHHTTTTHTTHTTTHHH
HHTTHHTHHTHTTTHHHTHHHTTTHHHHHHTHTHTTHHTHHTHTTHHHTT
HTTHTHTTTTTHHHHTHHHHHHTTTTTHTTTTHHHTHTTHHHTHHTHTT
TTHTHHHTTHTHTTHTTTHTHTTTTHTHTHTHTTHHHTHTTHTHTHTTHT
HHHTHHTHHTTHHTHTHHHTTTHHTHTHHTTHTTHTTHHHTTTHTHHHTH
```

Note, the line numbers are NOT part of the file. They are shown by the text editor I used.

Aside: Redirecting Output

There's another way to get the output of a program into a file.

When your file does a print, it sends the output to standard out, which is typically the terminal.

You can *redirect* the output to a file, using > filename on Linux systems. Anything that would have been printed on the screen goes into a file instead.

Notice that this happens at the OS level, not at the Python level. *Programmers know how to do things multiple ways!*

Can even redirect standard output inside of a Python program. This is part of how the auto grader works. Redirecting your program's standard output so we can compare it to what we expect the output to be.

Aside: Redirecting Output

```
plato% ls -l
total 36
drwxrwxr-x   3 scottm usl   4096 Feb  2   2001 00Fall
drwxr-x---   2 scottm usl   4096 Dec 18   2000 00Spring
drwx------   2 scottm prof  4096 Jun  2   2020 assignmn_solutions
drwxr-sr-x   2 scottm usl   4096 Feb  7   2001 Fall2000
drwx------  16 scottm prof  4096 Jul 28   2020 grading
-rw-r--r--   1 scottm prof   422 Jun  3  10:54 hello_world.py
drwxr-sr-x   2 scottm usl   4096 May 20   2001 Quilt
drwxr-sr-x   2 scottm usl   4096 Feb 16   2001 Rock
drwxr-sr-x   2 scottm usl   4096 Feb  7   2001 Spring2000
plato% python hello world.py > output.txt
plato% ls
00Fall    assignmn_solutions  grading         output.txt  Rock
00Spring  Fall2000            hello_world.py  Quilt       Spring2000
plato% cat output.txt
3
Hello World!
Hook 'em Horns!
362880
plato%
```

Example: Reading and Writing File

```
import os.path

def copy_file():
    ''' Copy contents from file1 to file2. '''
    # Ask user for filenames
    f1 = input('Source filename: ').strip()
    f2 = input('Target filename: ').strip()
    # Check if target file exists.
    if os.path.isfile( f2 ):
        print( f2 + ' already exists' )
        return
    # Open files for input and output
    infile = open( f1, 'r' )
    outfile = open( f2, 'w' )
    # Copy from input to output a line at a time
    for line in infile:
        outfile.write( line )
    # Close both files
    infile.close()
    outfile.close()

copy_file( )
```

Notice the use of a for loop to read all the lines in the file.

Example: Reading and Writing File

One cannot simultaneously read and write a file in Python. However, you can write a file, close it, and re-open it for reading.

```python
import random


def main():
    """Write out 100 random integers to a file, then read the file."""
    outfile = open('random_nums.txt', 'w')
    for i in range(100):
        outfile.write(str(random.randint(0, 99)) + ' ')
    outfile.close()

    # Now read in the numbers and print 10 per line
    infile = open('random_nums.txt', 'r')
    nums = infile.read()
    print(nums, '\n')

    numbers = [int(x) for x in nums.split()]
    num_printed = 0
    for x in numbers:
        num_printed += 1
        print(format(x, '3d'), end='')
        if num_printed == 10:
            print()
            num_printed = 0
        else:
            print(' ', end='')
```

```
 19  52  13  78  48  67  56  10   8  26
 34  54  75  80  16  85  83  97  40  70
 55   3  30  67  70  85   6  11  80   0
 56  56  28  57  67   2  57  52  90  52
 79  85  87  74  24  50  67  74  64  32
 71  42  97  22  75  57   7  18  77   1
 29  74  43  62  53  28  35  21  23  18
 48  82  22  71  62  23  84  98  53  36
 11  79  72  32  57  93   1  59  37  18
 42  27  43  54  11  50  12  77  80  43
```

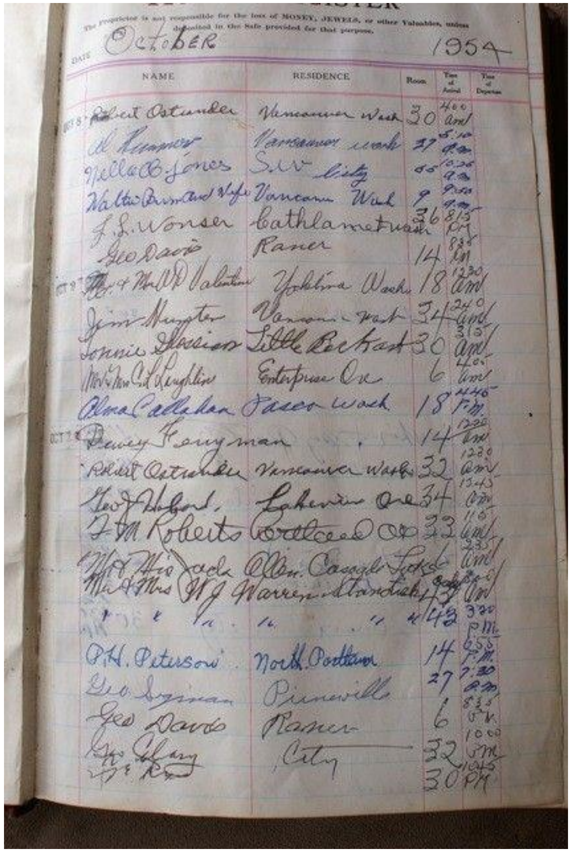
Append Mode

Opening a file in append mode `'a'`, means that writing a value to the file appends it at the end of the file.

It *does not* overwrite the previous content of the file.

You might use this to maintain a log file of transactions on an account.

New transactions are added at the end, but all transactions are recorded.



CS303E: Elements of Computers and Programming

Lists

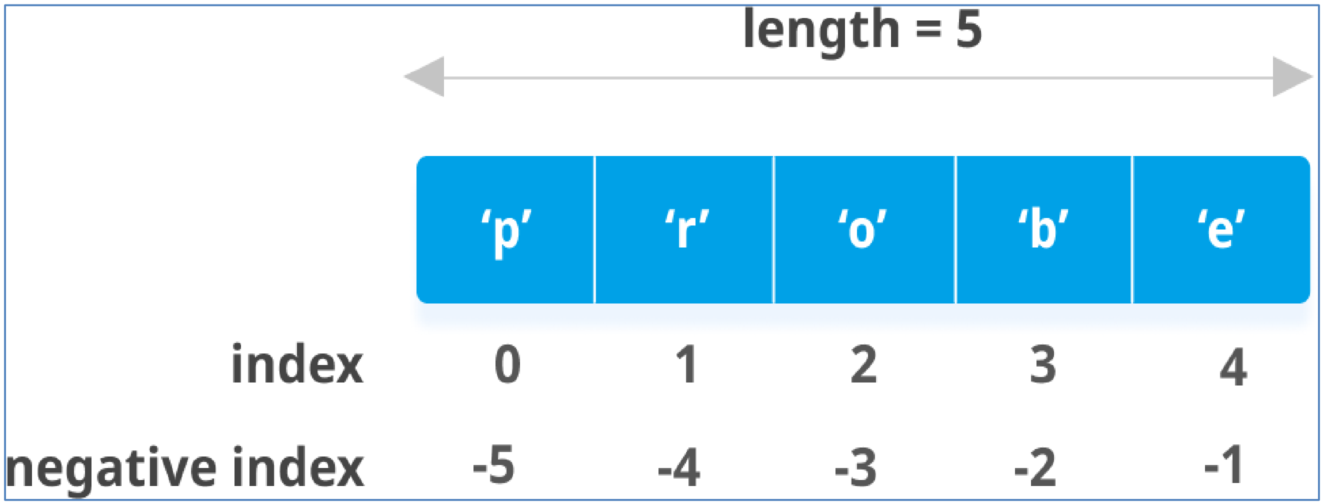
Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: June 28, 2023

Lists

The `list` class is a very useful tool in Python.



Both lists and strings are sequence types in Python, so share many similar methods. Unlike strings, lists are *mutable*.

If you change a list, it doesn't create a new copy; *it changes the actual contents of the list.*

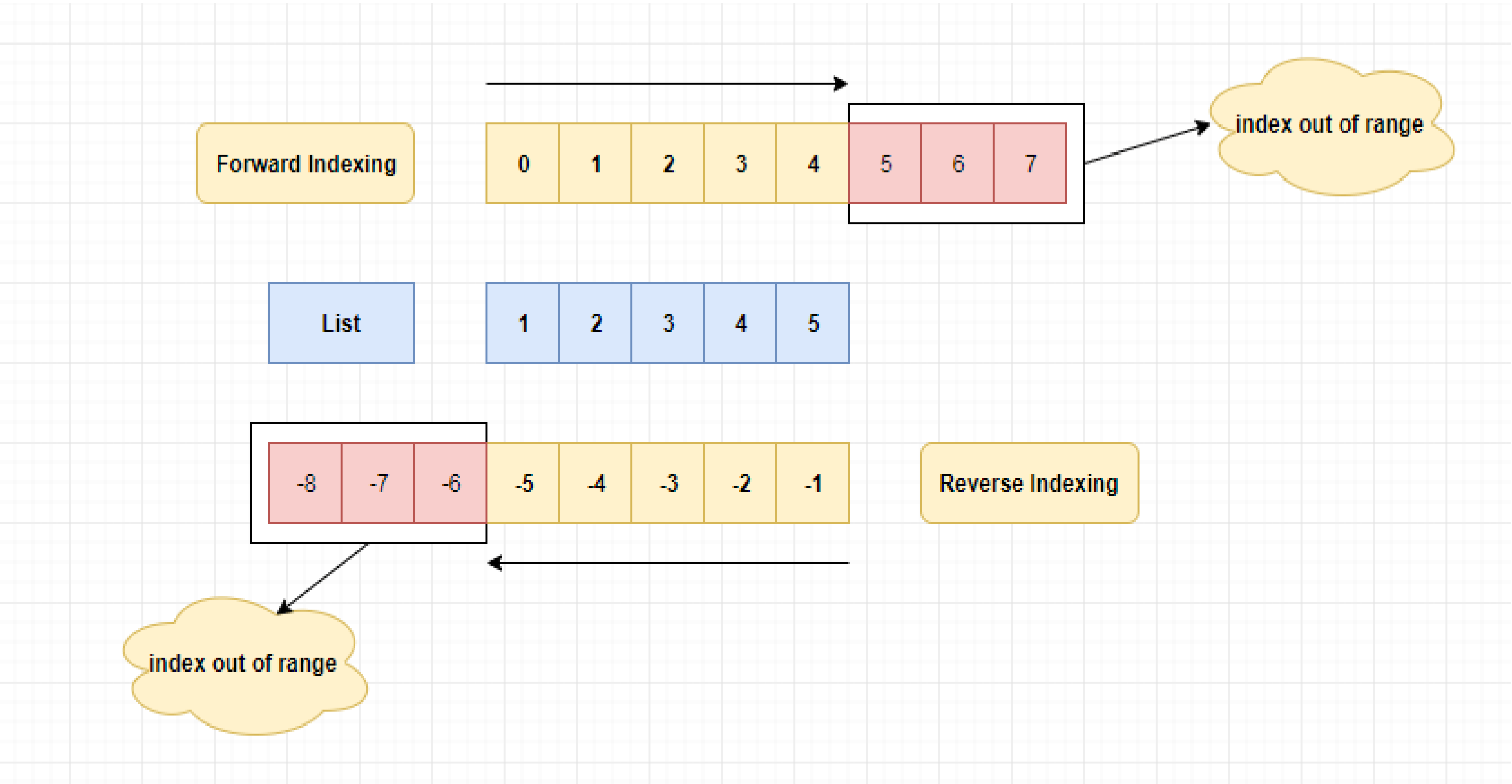
Value of Lists

Suppose you have 30 different test grades to average. You could use 30 variables: grade1, grade2, ..., grade30. Or you could use one list with 30 elements: grades[0], grades[1], ..., grades[29].

```
def grades_example():
    """Shows creation of a list and determining
    average of elements."""
    grades = [67, 82, 56, 84, 66, 77, 64, 64, 85, 67,
              73, 63, 98, 74, 81, 67, 93, 77, 97, 65,
              77, 91, 91, 74, 93, 56, 96, 90, 91, 99]
    total = 0
    for score in grades:
        total += score
    average = total / len(grades)
    print("Class average =", format(average, '.2f'))
```

Indexing and Slicing

With Lists you can get sublists using **slicing**



List Slicing

- List slicing format: `list[start : end]`

- Span is a list containing copies of elements from `start` up to, but not including, `end`
  - If `start` not specified, `0` is used for start index
  - If `end` not specified, `len(list)` is used for end index

- Slicing expressions can include a step value and negative indexes relative to end of list

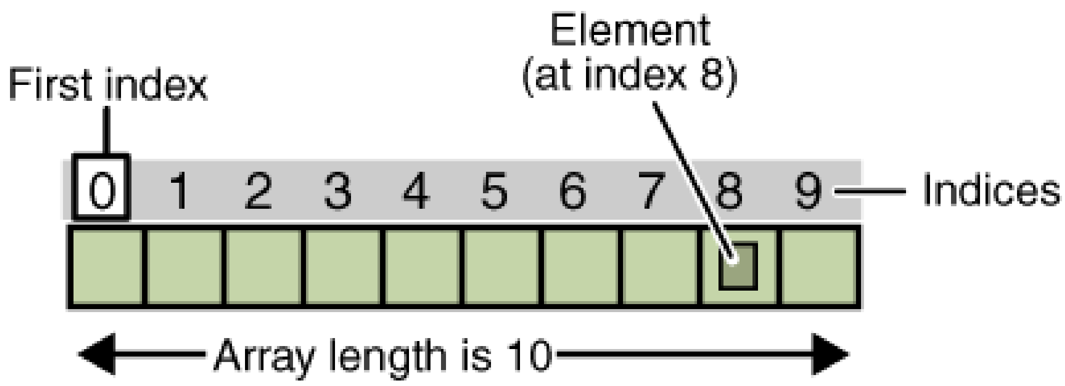
Creating Lists

Lists can be created with the `list` class constructor or using special syntax.

```
>>> list()                # create empty list, with constructor
[]
>>> list([1, 2, 3])       # create list [1, 2, 3]
[1, 2, 3]
>>> list(["red", 3, 2.5])    # create heterogeneous list
['red', 3, 2.5]
>>> ["red", 3, 2.5]       # create list, no explicit constructor
['red', 3, 2.5]
>>> range(4)              # not an actual list
range(0, 4)
>>> list(range(4))        # create list using range
[0, 1, 2, 3]
>>> list("abcd")          # create character list from string
['a', 'b', 'c', 'd']
```

Lists vs. Arrays

Many programming languages have an **array** type.



Arrays are:

- homogeneous (all elements are of the same type)
- fixed size
- permit very fast access time

Python lists are:

- heterogeneous (can contain elements of different types)
- variable size
- permit fast access time

Lists and arrays are examples of ***data structures***. A **very** simple definition of a data structure is **a variable that stores other variables.**
CS313e explores many standard data structures.

Sequence Operations

Lists are sequences and inherit various functions from sequences.

Function	Description
x in s	x is in sequence s
x not in s	x is not in sequence s
s1 + s2	concatenates two sequences
s * n	repeat sequence s n times
s[i]	ith element of sequence (0-based)
s[i:j]	slice of sequence s from i to j-1
len(s)	number of elements in s
min(s)	minimum element of s
max(s)	maximum element of s
sum(s)	sum of elements in s
for loop	traverse elements of sequence
<, <=, >, >=	compares two sequences
==, !=	compares two sequences

Calling Functions on Lists

```
>>> l1 = [1, 2, 3, 4, 5]
>>> len(l1)
5
>>> min(l1)        # assumes elements are comparable
1
                   # assumes elements are comparable
>>> max(l1)
5
                   # assumes summing makes sense
>>> sum(l1)
15
>>> l2 = [1, 2, "red"]
>>> sum(l2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str
    '
>>> min(l2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and
    'int'
>>>
```

We could rewrite the grades_examples function as follows:

```
def grades_example_2():
    """Shows creation of a list and determining
    average of elements. This version takes advantage
    of the sum function for sequences."""
    grades = [67, 82, 56, 84, 66, 77, 64, 64, 85, 67,
              73, 63, 98, 74, 81, 67, 93, 77, 97, 65,
              77, 91, 91, 74, 93, 56, 96, 90, 91, 99]
    average = sum(grades) / len(grades)
    print("Class average =", format(average, '.2f'))
```

Traversing Elements with a For Loop

General Form:

```
for u in list:
    body
```

In file `test.py`:

```
for u in range(3):                # not really a list
    print(u, end=" ")
print()

for u in [2, 3, 5, 7]:
    print(u, end=" ")
print()

for u in range(15, 1, -3):    # not really a list
    print(u, end=" ")
print()
```

```
> python test.py
0 1 2
2 3 5 7
15 12 9 6 3
```

Comparing Lists

Compare lists using the operators: `>`, `>=`, `<`, `<=`, `==`, `!=`. Uses *lexicographic* ordering: Compare the first elements of the two lists; if they match, compare the second elements, and so on. The elements must be of *comparable* classes.

```
>>> list1 = ["red", 3, "green"]
>>> list2 = ["red", 3, "grey"]
>>> list1 < list2
True
>>> list3 = ["red", 5, "green"]
>>> list3 > list1
True
>>> list4 = [5, "red", "green"]
>>> list3 < list4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and
    'int'
>>> ["red", 5, "green"] == [5, "red", "green"]
False
```

List Comprehension

List comprehension gives a compact syntax for building lists.

```
>>> range(4)                                # not actually a list
range(0, 4)
>>> [ x for x in range(4) ]                 # create list from range
[0, 1, 2, 3]
>>> [ x ** 2 for x in range(4) ]
[0, 1, 4, 9]
>>> lst = [ 2, 3, 5, 7, 11, 13 ]
>>> [ x ** 3 for x in lst ]
[8, 27, 125, 343, 1331, 2197]
>>> [ x for x in lst if x > 2 ]
[3, 5, 7, 11, 13]
>>> [s[0] for s in ["red", "green", "blue"] if s <= "green"]
['g', 'b']
>>> from IsPrime3 import *
>>> [ x for x in range(100) if isPrime(x) ]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
     59, 61, 67, 71, 73, 79, 83, 89, 97]
```

List Comprehension with Files

List comprehension gives a compact syntax for building lists, even from files.

mples.py × | sample.txt ×

```
Team by team, reporters baffled, trumped, tethered, cropped
Look at that low plane, fine, then
Uh oh, overflow, population, common group
But it'll do, save yourself, serve yourself
World serves its own needs, listen to your heart bleed
Tell me with the Rapture and the reverent in the right, right
You vitriolic, patriotic, slam fight, bright light
Feeling pretty psyched
```

List Comprehension with Files

List comprehension gives a compact syntax for building lists, even from files.

```
def list_from_file(file_path):
    """Read the lines from the given file and print them out."""
    with open(file_path, 'r') as infile:
        lines = [line.strip() for line in infile]
    print('number of lines:', len(lines))
    line_num = 1
    for line in lines:
        print(line_num, ': ', line, sep='')
        line_num += 1
```

List Comprehension with Files

List comprehension gives a compact syntax for building lists, even from files.

```
1: Team by team, reporters baffled, trumped, tethered, cropped
2: Look at that low plane, fine, then
3: Uh oh, overflow, population, common group
4: But it'll do, save yourself, serve yourself
5: World serves its own needs, listen to your heart bleed
6: Tell me with the Rapture and the reverent in the right, right
7: You vitriolic, patriotic, slam fight, bright light
8: Feeling pretty psyched
```

Let’s Take a Break

More List Methods

These are methods from class `list`.

Since lists are mutable, these actually change `t`.

Method	Description
`t.append(x)`	add x to the end of t
`t.count(x)`	number of times x appears in t
`t.extend(l1)`	append elements of l1 to t
`t.index(x)`	index of first occurence of x in t
`t.insert(i, x)`	insert x into t at position i
`t.pop()`	remove and return the last element of t
`t.pop(i)`	remove and return the ith element of t
`t.remove(x)`	remove the first occurence of x from t
`t.reverse()`	reverse the elements of t
`t.sort()`	order the elements of t

List Examples

```
>>> l1 = [1, 2, 3]
>>> l1.append(4) # add 4 to the end of l1
>>> l1              # note: changes l1
[1, 2, 3, 4]
>>> l1.count(4)   # count occurrences of 4 in          l1
1
>>> l2 = [5, 6, 7]
>>> l1.extend(l2)    # add elements of l2 to l1
>>> l1
[1, 2, 3, 4, 5, 6, 7]
>>> l1.index(5)    # where does 5 occur in l1?
4
>>> l1.insert(0, 0)  # add 0 at the start of l1
>>> l1              # note new value of l1
[0, 1, 2, 3, 4, 5, 6, 7]
>>> l1.insert(3, 'a')          # lists are heterogenous
>>> l1
[0, 1, 2, 'a', 3, 4, 5, 6, 7]
>>> l1.remove('a')   # what goes in can come out
>>> l1
[0, 1, 2, 3, 4, 5, 6, 7]
```

List Examples

```
>>> l1.pop()                # remove and return last element
7
>>> l1
[0, 1, 2, 3, 4, 5, 6]
>>> l1.reverse()            # reverse order of elements
>>> l1
[6, 5, 4, 3, 2, 1, 0]
>>> l1.sort()               # elements must be comparable
>>> l1
[0, 1, 2, 3, 4, 5, 6]
>>> l2 = [4, 1.3, "dog"]
>>> l2.sort()               # elements must be comparable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and
    'float'
>>> l2.pop()                # remove 'dog'
'dog'
>>> l2
[4, 1.3]
>>> l2.sort()               # int and float are comparable
>>> l2
[1.3, 4]
```

Random Shuffle

A useful method on lists is random.shuffle() from the random module.

```
>>> list1 = [ x for x in range(9) ]
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> random.shuffle(list1)
>>> list1
[7, 4, 0, 8, 1, 6, 5, 2, 3]
>>> random.shuffle(list1)
>>> list1
[4, 1, 5, 0, 7, 8, 3, 2, 6]
>>> random.shuffle( list1 )
>>> list1
[7, 5, 2, 6, 0, 4, 3, 1, 8]
```

Processing CSV Lines

Suppose grades for a class were stored in a list of csv strings, such as:

```
student_data = ['Alice,90,75',
                'Robert,8,77',
                'Charlie,60,80']
```

Here the fields are: Name, Midterm grade, Final Exam grade.

Compute the average for each student and print a table of results.

Processing CSV Lines from List

```python
def print_test_scores(student_data):
    """Print the test scores for the elements of student_data.

    student_data is a list of Strings. Each String is of the form:
    '<Name>, <Midterm Score>, <Final Score>'
    Course score is based on 1/3 of midterm score and 2/3s of
    final score.
    """
    print('Name          MT    FN   Course')
    print('----------------------------')
    for student in student_data:
        data = student.split(",")
        if len(data) != 3:
            print('Bad student data:', student)
        else:
            name = data[0].strip()
            midterm = int(data[1].strip())
            final = int(data[2].strip())
            course_score = midterm / 3 + final * 2 / 3
            print(format(name, '10s'), format(midterm, '4d'),
                  format(final, '4d'), format(course_score, '6.2f'))
```

Processing CSV Lines

```
students = ['Alice, 90 ,98', ' Robert ,58 ,77',
            'Michael, 80', 'Charlie, 60 ,80']
print_test_scores(students)
```

```
Name            MT    FN   Course
---------------------------------
Alice           90    98   95.33
Robert          58    77   70.67
Bad student data: Michael, 80
Charlie         60    80   73.33
```

Copying Lists

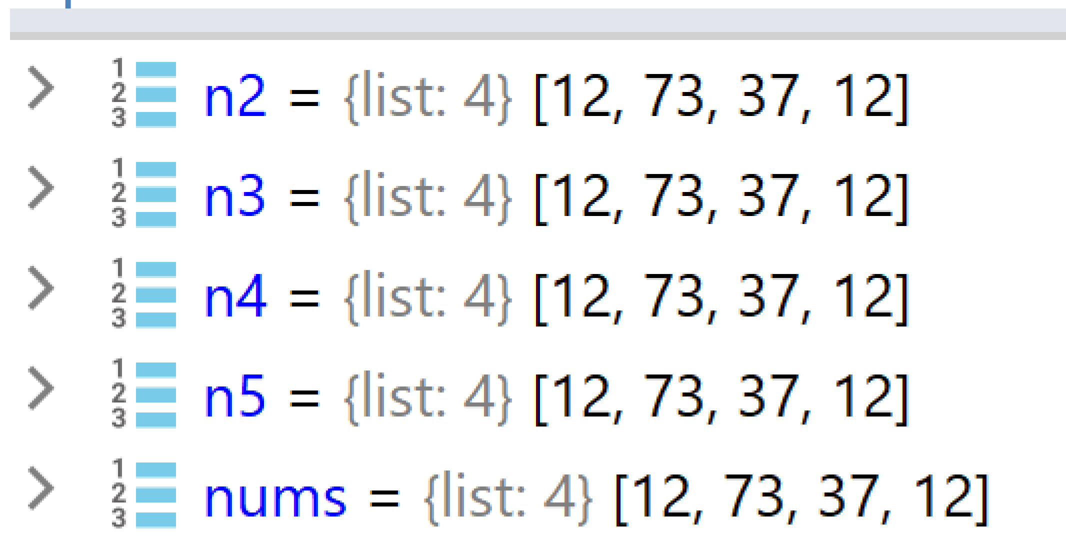
Suppose you want to make a copy of a list. *The following won't work!*

```
>>> nums = [12, 56, 37, 12]
>>> n2 = nums
>>> n2 is nums
True
>>> n2 == nums
True
>>> n2[1] = 73
>>> n2
[12, 73, 37, 12]
>>> nums
[12, 73, 37, 12]
```

Copying Lists

But, many ways of making a copy of a list.

```
>>> nums
[12, 73, 37, 12]
>>> n2 = nums.copy()
>>> n2 is nums
False
>>> n3 = list(nums)
>>> n3 is nums
False
>>> n3 is n2
False
>>> n4 = nums[0:]
>>> n4 is nums
False
>>> n5 = [i for i in nums]
>>> n5 is nums
False
```



Passing Lists to Functions

Like any other *mutable* object, when you pass a list to a function, you're really passing a reference (pointer) to the object in memory.

```
def alter( lst ):
    lst.pop()

def main():
    lst = [1, 2, 3, 4]
    print( "Before call: ", lst )
    alter( lst )
    print( "After call: ", lst )

main()
```

```
> python ListArg.py
Before call:  [1, 2, 3, 4]
After call:  [1, 2, 3]
```


TIME FOR A
BREAK

Example Problems

To get good at working with lists, we must practice!

- CodingBat: https://codingbat.com/python
  - List1: first_last6, same_first_last, max_end3
  - List2: count_even, big_diff, has_22
- given list of ints or floats, is it sorted in descending order?
- get **last** index of a given value in list
- given two lists of ints, return a list that contains the difference between corresponding elements
  - change to be the max
- are all the elements of a given list unique? In other words, no duplicate values in the list
- given a list of ints place all even values before all odd values

CS303E: Elements of Computers and Programming

Lists of Lists

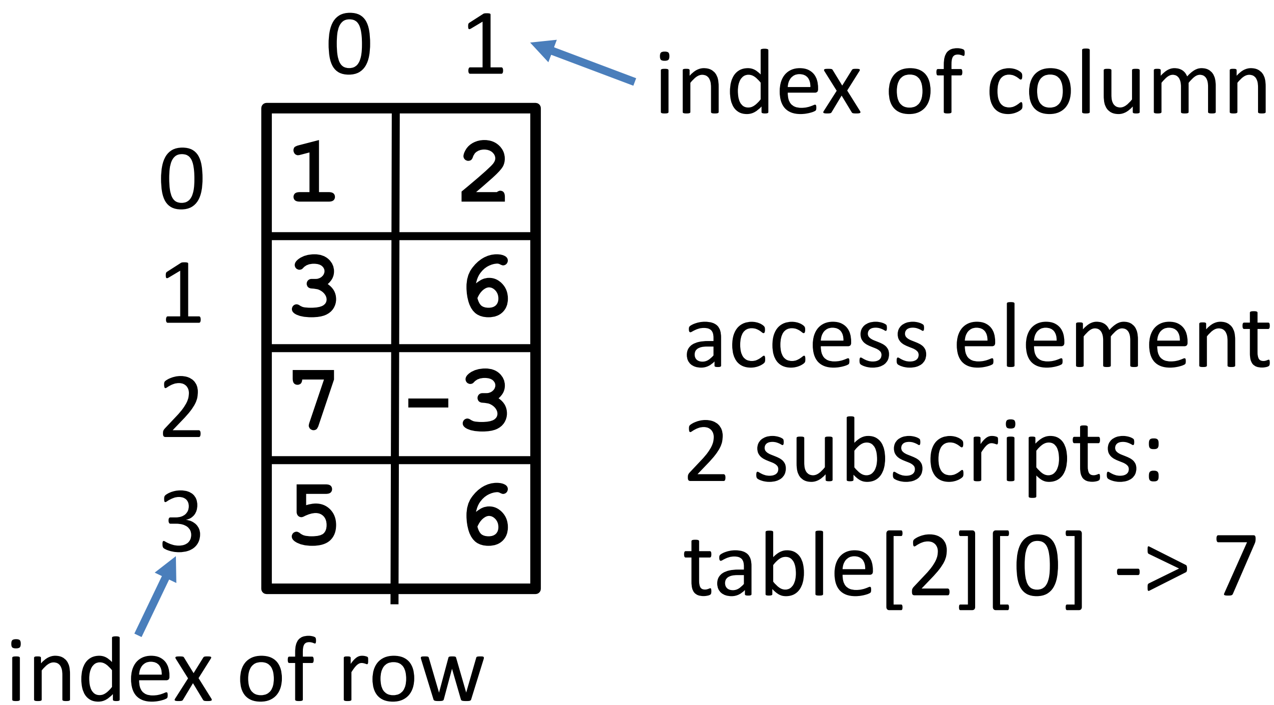
Mike Scott
Department of Computer Science
University of Texas at Austin

Last updated: May 30, 2024

Can create list of lists in Python

table = [[1, 2], [3, 6], [7,-3], [5, 6]]

- Access an element with 2 subscripts.
- By convention first subscript is row and the second is the column



access element with
2 subscripts:
table[2][0] -> 7

Can also use list comprehension

table2 = [[0] * 12] * 10

A list of lists with 10 rows and 12 columns per row.

flips = [['H' if random.random() <= 0.5 else 'T' for x in range(12)] for x in range(10)]

A table with 10 rows and 12 columns per row. Each elements is a random coin flip.

Write a function that returns the index of the row of a list of lists of ints has the largest sum. In the case of a tie return the index closest to 0.

Write a function that returns the index of the **column** of a list of lists of ints has the largest sum. In the case of a tie return the index closest to 0.

Conway's Game of Life

- a cellular automaton designed by John Conway, a mathematician
- not really a game
- a simulation
- takes place on a 2d grid
- each element of the grid is occupied or empty by a simple organism, but not any known organism

Simulation

http://www.cuug.ab.ca/dewara/life/life.html

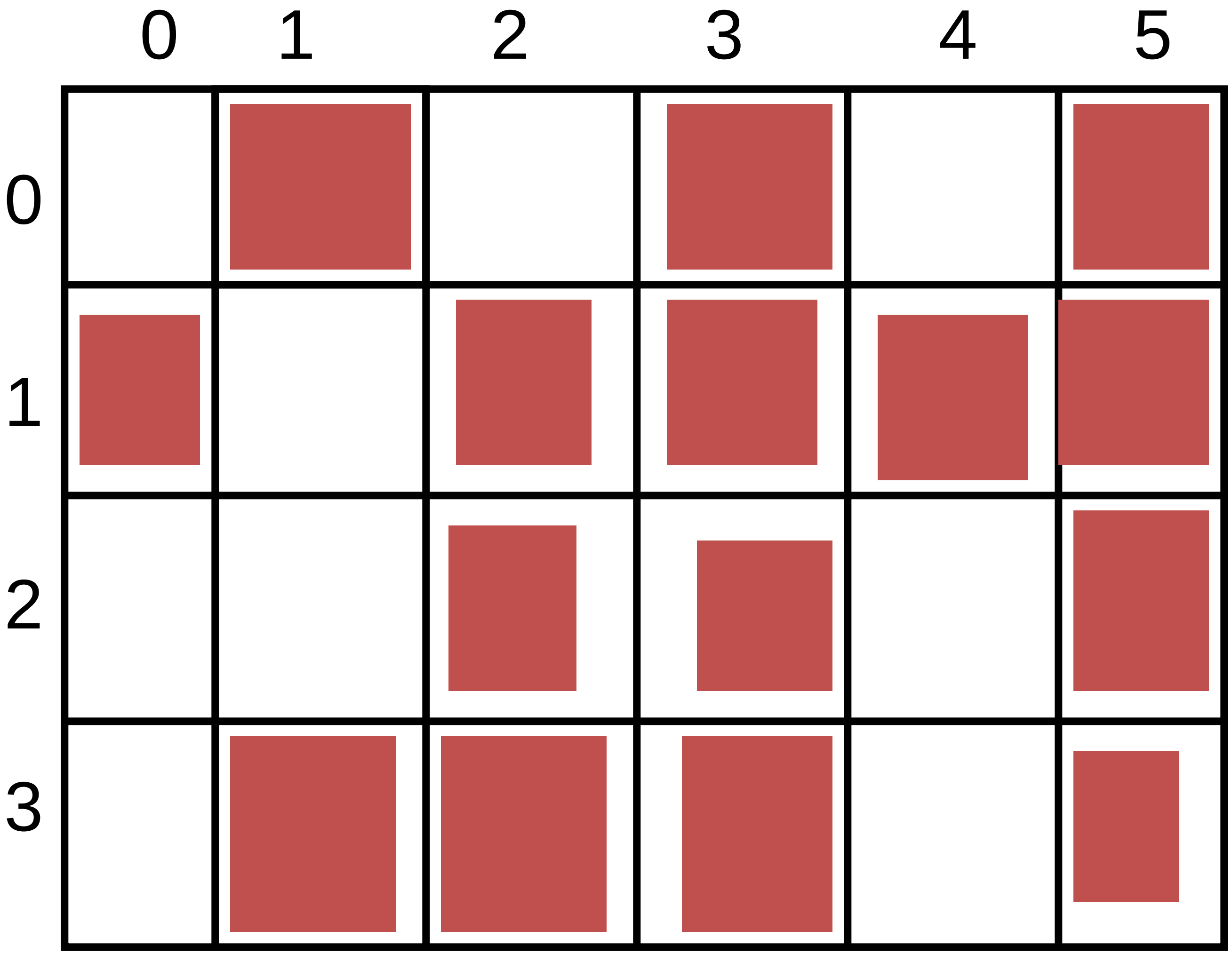
- Select pattern from menu
- Select region in large area with mouse by pressing the control key and left click at the same time
- Select the paste button

Generation 0

	0	1	2	3	4	5
0	.	*	.	*	.	*
1	*	.	*	*	*	*
2	.	.	*	*	.	*
3	.	*	*	*	.	*

* indicates occupied, . indicates empty

Or

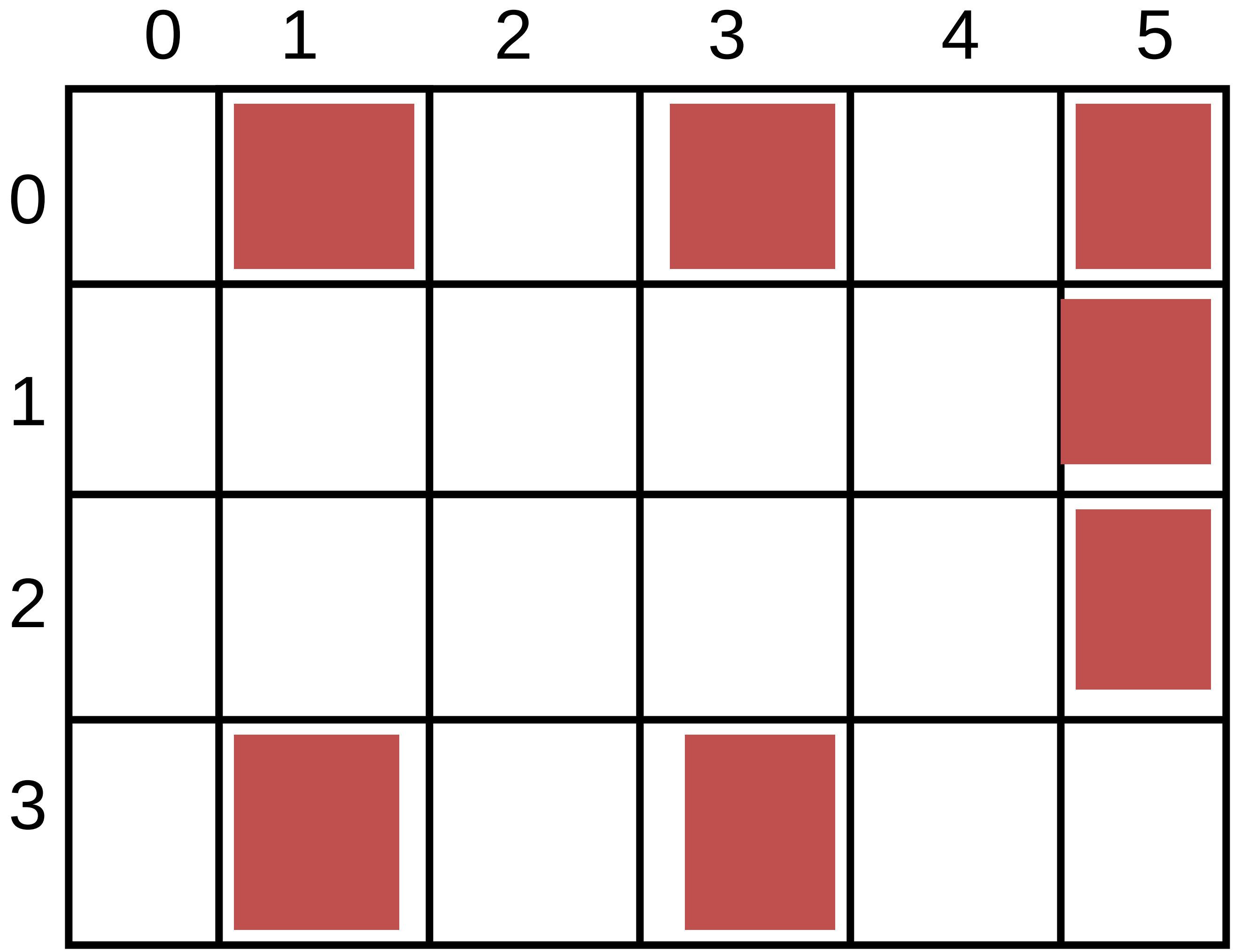


Generation 1

	0	1	2	3	4	5
0	.	*	.	*	.	*
1	.	.	.	.	.	*
2	.	.	.	.	.	*
3	.	*	.	*	.	.

* indicates occupied, . indicates empty

Or , Generation 1



Rules of the "Game"

If a cell is occupied in this generation.

- it survives if it has 2 or 3 neighbors in this generation
- it dies if it has 0 or 1 neighbors in this generation
- it dies if it has 4 or more neighbors in this generation

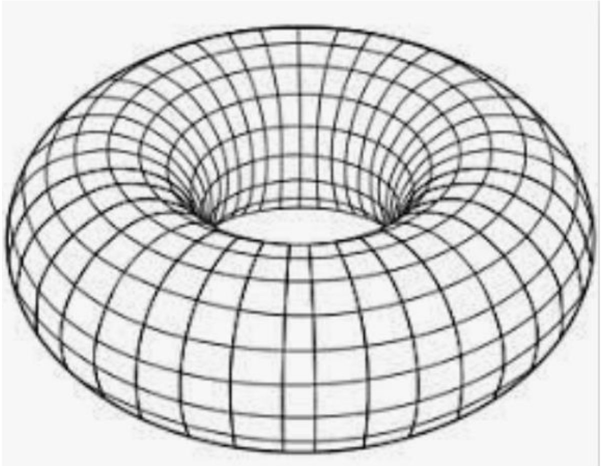
If a cell is unoccupied in this generation.

there is a birth if it has exactly 3 neighboring cells that are occupied in this generation

Neighboring cells are up, down, left, right, and diagonal. In general a cell has 8 neighboring cells

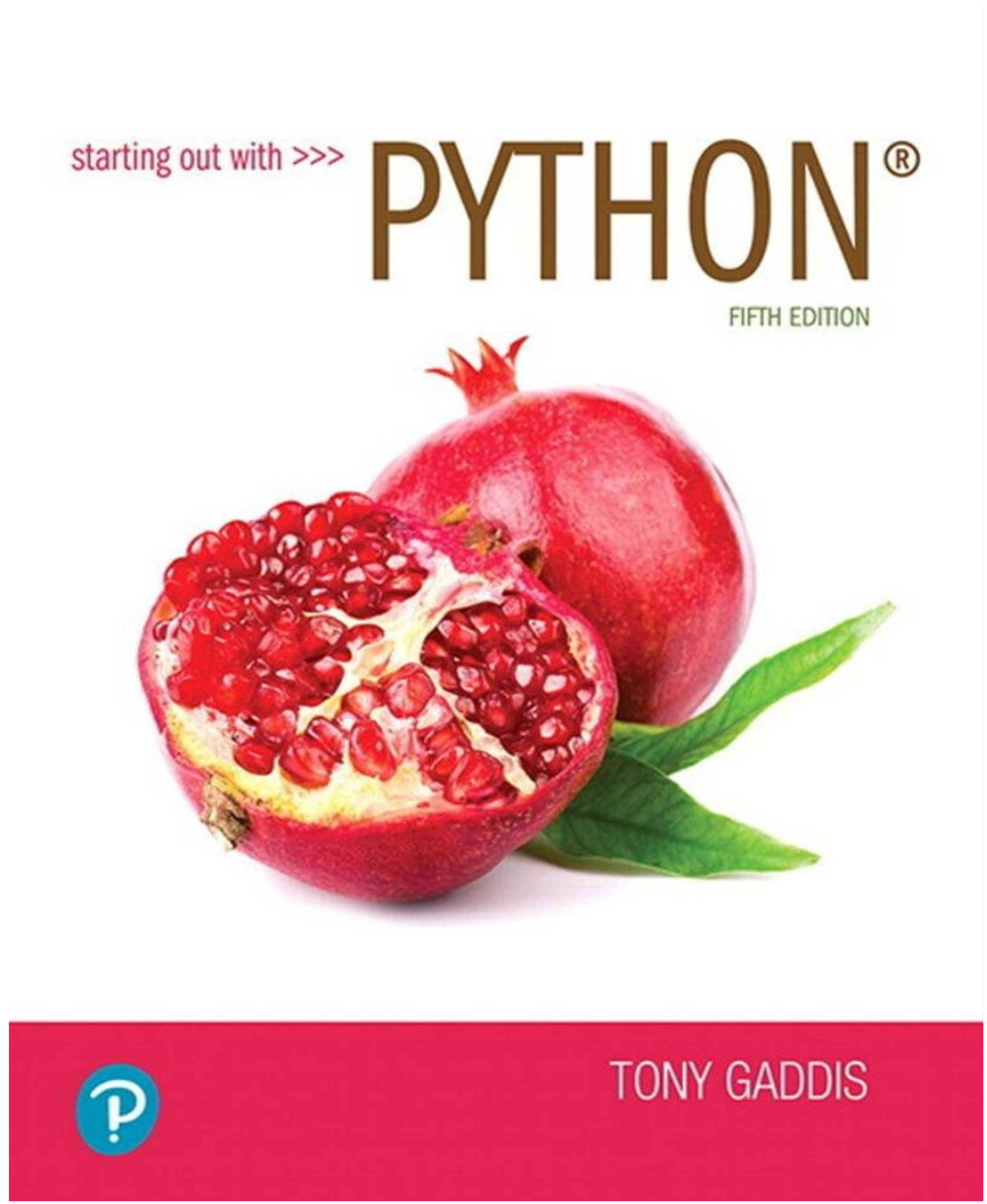
Design and implement a complete Python program to automate Conway's Game of Life

- text based
- user input for size of world
- wrapped or bounded?
- border or not?
- high level design first,
  then implement solution
- test, test, test, test



7.9 and Chapter 8

Tuples and More About Strings



Tuples

- **Tuple**: an **immutable** sequence
  - similar to a list, but ….
  - Once it is created it cannot be changed
  - Format: `tuple_name = (item1, item2)`
  - Notice the use of ( ) instead of [ ]
  - Tuples have operations similar to lists
    - Subscript indexing for retrieving elements
    - Methods such as `index`
    - Built in functions such as `len, min, max`
    - Slicing expressions
    - The `in`, `+`, and `*` operators

Tuples (cont'd.)

- **Tuples do not support the methods:**
  - `append`
  - `remove`
  - `insert`
  - `reverse`
  - `sort`
  - Why not? They are immutable.

Tuples (cont'd.)

- **Advantages for using tuples over lists:**
  - Processing tuples is faster than processing lists
  - Tuples can be safer (immutable)
  - Some operations in Python require use of tuples
- **<u>`list()` function</u>: converts tuple to list**
- **<u>`tuple()` function</u>: converts list to tuple**
- **Fun fact, a function that returns 2 or more values returns them in a tuple**

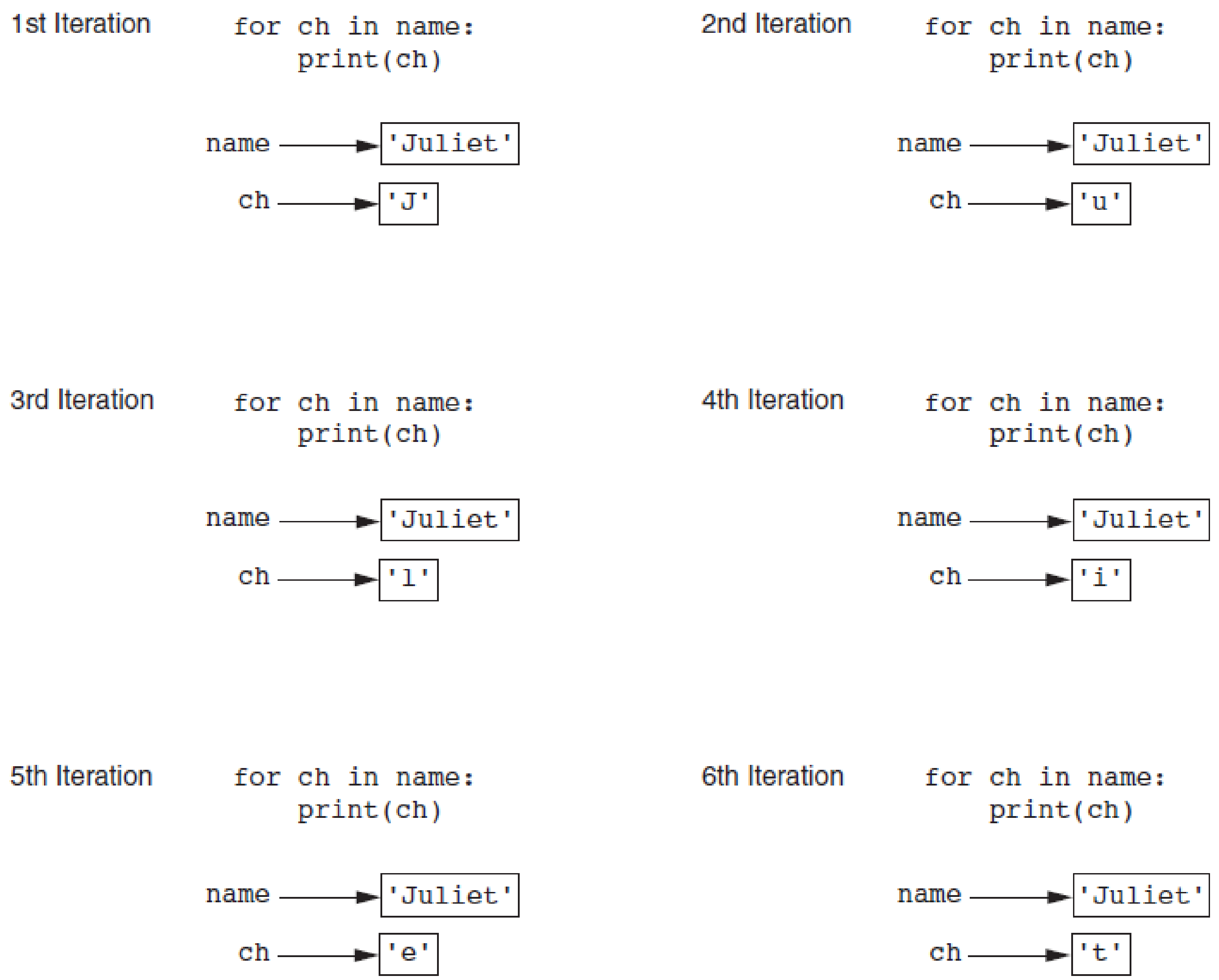
Basic String Operations

- **Many types of programs perform operations on strings**
- **In Python, many tools for examining and manipulating strings**
  - Strings are sequences, so many of the tools that work with sequences (such as ranges, lists, and tuples) also can be used with strings

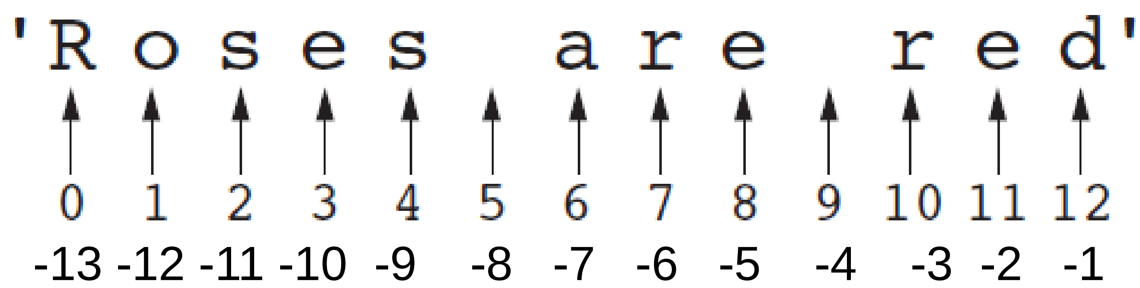
Accessing the Individual Characters in a String

- **To access an individual character in a string:**
  - Use a `for` loop
    - Format: `for character in string:`
    - Useful when need to iterate over the whole string, such as to count the occurrences of a specific character
    - Each ‘character’ is simply a string of length 1
  - Use indexing
    - Each character has an index specifying its position in the string, starting at 0
    - Format: `character = my_string[i]`

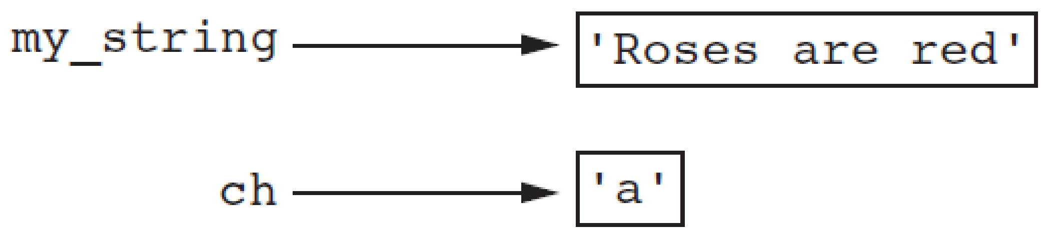
Figure 8-1 Iterating over the string `'Juliet'`



Accessing the Individual Characters in a String (cont'd.)



Getting a copy of a character from a string




```
ch = my_string[6]
```

Accessing the Individual Characters in a String (cont'd.)

- **`IndexError` exception will occur if:**
  - You try to use an index that is out of range for the string
    - Likely to happen when loop iterates beyond the end of the string
- **use the `len(`*`string`*`)` function to obtain the length of a string**
  - Useful to prevent loops from iterating beyond the end of a string

Accessing the Individual Characters in a String

- **How to access the individual elements of the string using a for loop and the range function?**

```
name = 'Olivia A.'
for in range(len(name)):
    print(name[i],
          type(name[i]))
```

```
O <class 'str'>
l <class 'str'>
i <class 'str'>
v <class 'str'>
i <class 'str'>
a <class 'str'>
  <class 'str'>
A <class 'str'>
. <class 'str'>
```

- **Or**

  ```
  for ch in string_var:
  ```

  if we don't care about position

String Concatenation

- **<u>Concatenation</u>: appending one string to the end of another string**
  - Use the + operator to produce a string that is a combination of its operands
  - The augmented assignment operator += can also be used to concatenate strings
    - The operand on the left side of the += operator must be an existing variable; otherwise, an exception is raised

Strings Are Immutable

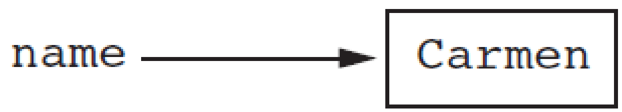
- **Strings are immutable**
  - Once they are created, they cannot be changed
    - Concatenation doesn't actually change the existing string, but rather creates a new string and assigns the new string to the previously used variable
  - Cannot use an expression of the form
  - *string[index] = new_character*
    - Statement of this type will raise an exception

```
>>> name
'Olivia A.'
>>> name[7] = 'R'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Strings Are Immutable, Variables Are Not

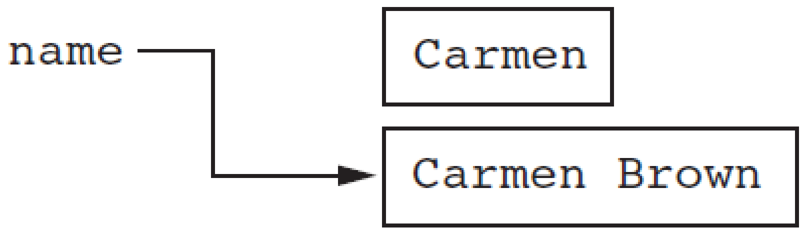
The string ‘Carmen’ assigned to `name`

```
name = 'Carmen'
```



The string ‘Carmen Brown’ assigned to `name`

```
name = name + ' Brown'
```



String Slicing

- **<u>Slice</u>: span of items taken from a sequence, known as *substring***
  - Slicing format: `string[start : end]`
    - Expression will return a string containing a copy of the characters from `start` up to, but not including, `end`
    - If `start` not specified, `0` is used for start index
    - If `end` not specified, `len(string)` is used for end index
  - Slicing expressions can include a step value and negative indexes relative to end of string

Testing, Searching, and Manipulating Strings

- **You can use the `in` operator to determine whether one string is contained in another string**
  - General format: *`string1 in string2`*
    - *`string1`* and *`string2`* can be string literals or variables referencing strings
- **Similarly you can use the `not in` operator to determine whether one string is not contained in another string**

String Methods

- **Strings in Python have many types of methods, divided into different types of operations**
  - General format:

    `mystring.method(arguments)`

- **Some methods test a string for specific characteristics**
  - Generally Boolean methods, that return `True` if a condition exists, and `False` otherwise

String Methods (cont'd.)

Table 8-1 Some string testing methods

Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

Implement a function that prompts the user for an int and error checks it. Keep prompting until they enter an int

String Methods (cont'd.)

- **Some methods create and return a modified version of the string**
  - Simulate strings as mutable objects
- **String comparisons are case-sensitive**
  - Uppercase characters are distinguished from lowercase characters
  - `lower` and `upper` methods can be used for making case-insensitive string comparisons

String Methods (cont'd.)

Table 8-2 String Modification Methods

Method	Description
`lower()`	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`lstrip(char)`	The *char* argument is a string containing a character. Returns a copy of the string with all instances of *char* that appear at the beginning of the string removed.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`rstrip(char)`	The *char* argument is a string containing a character. The method returns a copy of the string with all instances of *char* that appear at the end of the string removed.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`strip(char)`	Returns a copy of the string with all instances of *char* that appear at the beginning and the end of the string removed.
`upper()`	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

String Methods (cont'd.)

- **Programs commonly need to search for substrings**
- **Several methods to accomplish this:**
  - `endswith(`*`substring`*`)`: checks if the string ends with *`substring`*
    - Returns `True` or `False`
  - `startswith(`*`substring`*`)`: checks if the string starts with *`substring`*
    - Returns `True` or `False`

String Methods (cont'd.)

- **Several methods to accomplish this (cont'd):**
  - `find(`*`substring`*`)`: searches for *`substring`* within the string
    - Returns lowest index of the substring, or if the substring is not contained in the string, returns -1
  - `replace(`*`substring, new_string`*`)`:
    - Returns a copy of the string where every occurrence of *`substring`* is replaced with *`new_string`*

String Methods (cont'd.)

Table 8-3 Search and replace methods

Method	Description
`endswith(`*`substring`*`)`	The *substring* argument is a string. The method returns true if the string ends with *substring*.
`find(`*`substring`*`)`	The *substring* argument is a string. The method returns the lowest index in the string where *substring* is found. If *substring* is not found, the method returns −1.
`replace(`*`old, new`*`)`	The *old* and *new* arguments are both strings. The method returns a copy of the string with all instances of *old* replaced by *new*.
`startswith(`*`substring`*`)`	The *substring* argument is a string. The method returns true if the string starts with *substring*.

The Repetition Operator

- **Repetition operator: makes multiple copies of a string and joins them together**
  - The * symbol is a repetition operator when applied to a string and an integer
    - String is left operand; number is right
  - General format: `string_to_copy * n`
  - Variable references a new string which contains multiple copies of the original string

Splitting a String

- **`split` method: returns a list containing the words in the string**
  - By default, uses space as separator
  - Can specify a different separator by passing it as an argument to the `split` method
  - Also referred to as *parsing* a string.

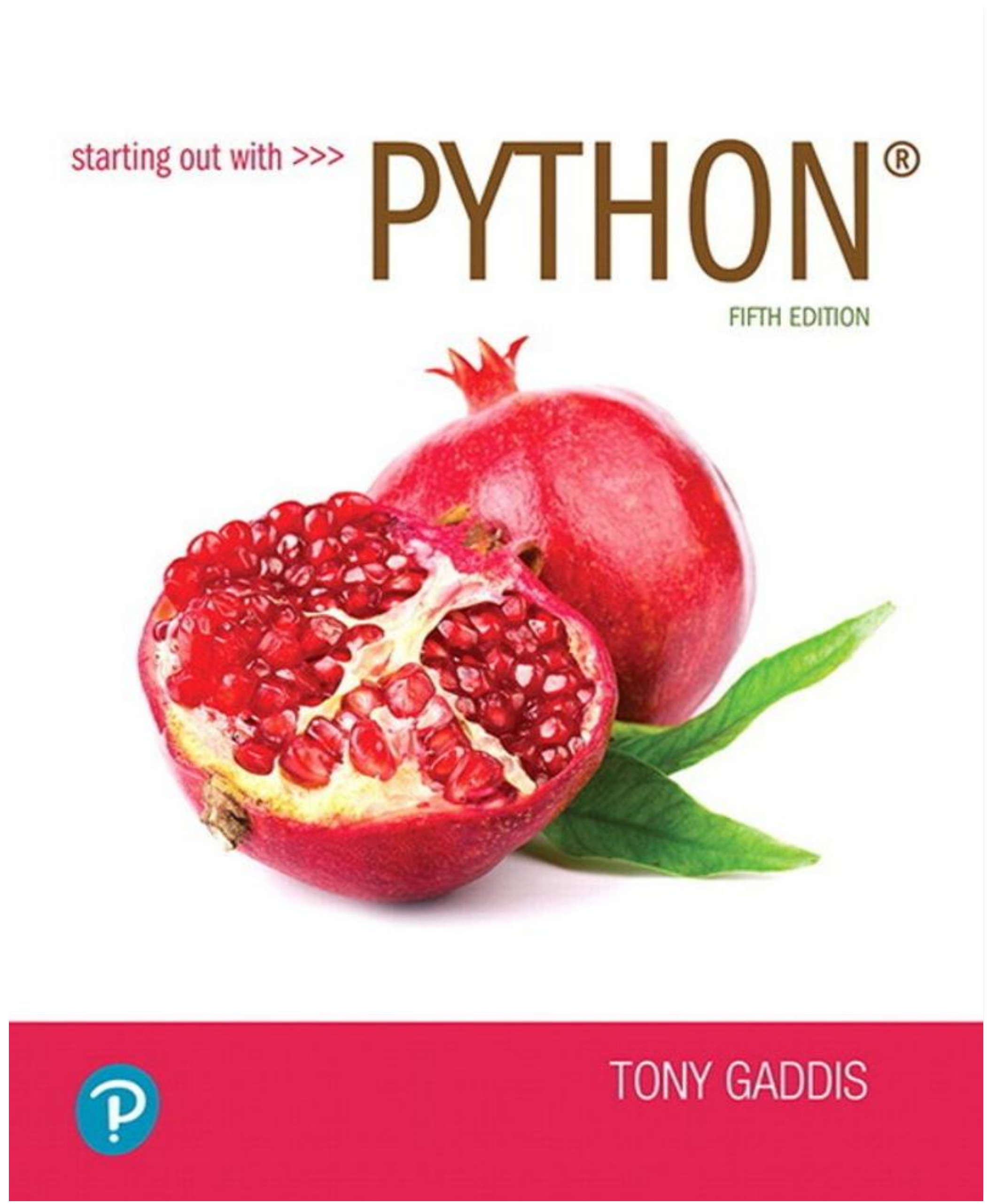
chr and ord Functions

- Recall, the vast majority of computer systems store data in a binary form, 0's and 1's
- We have *encoding schemes* to specify what a given sequence of 0's and 1's represents, such as characters, colors, sound
- In Python, the built in chr and ord functions can be used to see the encoding for strings of length 1

```
>>> ord('A')
65
>>> ord(' ')
32
>>> ord('a')
97
>>> chr(101)
'e'
>>> chr(66)
'B'
```

CHAPTER 9

Dictionaries and Sets



Topics

- Dictionaries
- Sets
- Serializing Objects

DNA Count

- **DNA <u>D</u>eoxyribo<u>n</u>ucleic <u>a</u>cid**
  - "The polymer carries genetic instructions for the development, functioning, growth and reproduction of all known organisms and many viruses. "
- **Part of the building blocks of DNA are 4 nitrogen containing nucleobases**
  - cytosine [C], guanine [G], adenine [A] or thymine [T]

DNA Data

- **Massive amounts of work to catalog and decode DNA in organisms has been done.**
- **https://www.kaggle.com/datasets/nageshsingh/dna-sequence-dataset?select=dog.txt**
- **ATGCCACAGCTAGATACATCCACCTGATTTATTATA ATCTTTTCAATATTTCTCACCCTCTTCATCCTATTTC AACTAAAAATTTCAAATCACTACTACCCAGAAAAC CCGATAACCAAATCTGCTAAAATTGCTGGTCAACA TAATCCTTGAGAAAACAAATGAACGAAAATCTATTC GCTTCTTTCGCTGCCCCCTCAATAA**

DNA Counts

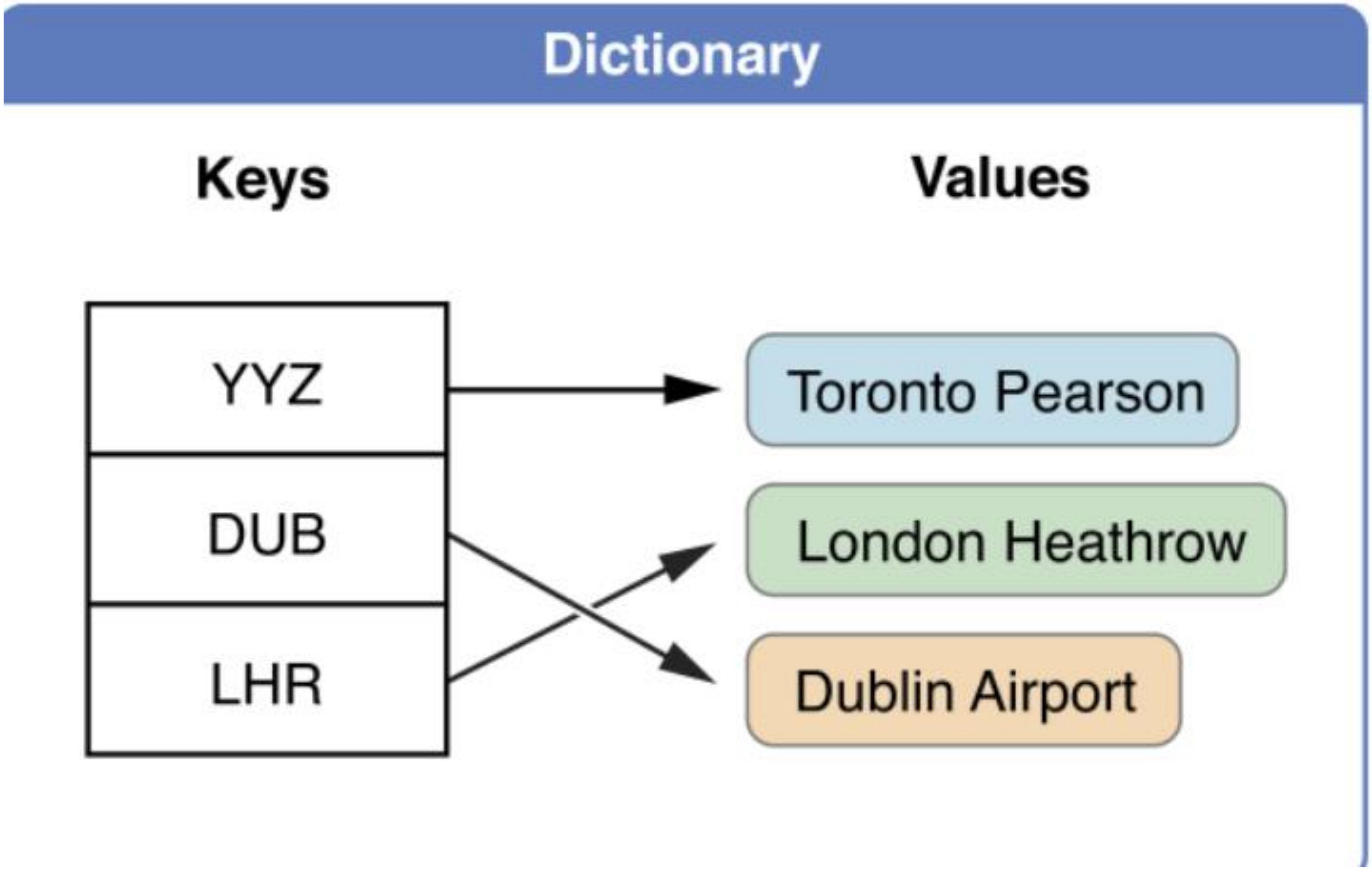
- **Write a function that given a string that represents a portion of DNA returns the frequency of the four nucleobases**
- **cytosine [C], guanine [G], adenine [A] or thymine [T]**

Dictionaries

- **<u>Dictionary</u>: data structure that stores a collection of *key-value pairs***
  - Each element consists of a *key* and a *value*
    - Often referred to as *mapping* of key to value
    - Key must be an immutable object
    - A real world dictionary, the words are the keys and the definitions are the values
    - Given the word you can find the value ***quickly***
  - To retrieve a specific value, use the key associated with it
  - Format for creating a dictionary with given values

```
dictionary = {key1:val1, key2:val2}
```

Visualization of Dictionary



- **https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html**

Retrieving a Value from a Dictionary

- **Prior to Python 3.7 the keys in a dictionary are in no discernible order from the client's perspective**
- **Python 3.7 and later, dictionaries maintain keys in *insertion order***
- **General format for retrieving value from dictionary: `dictionary[key]`**
  - If `key` in the dictionary, associated value is returned, otherwise, `KeyError` exception is raised
- **Test whether a key is in a dictionary using the `in` and `not in` operators**
  - Helps prevent `KeyError` exceptions

Adding Elements to an Existing Dictionary

- **Dictionaries are mutable objects**
- **To add a new key-value pair:**

```
dictionary[key] = value
```

  - If key exists in the dictionary, the value associated with it will be changed
  - if the key doesn't exist this adds the *key-value* pair to the dictionary

Deleting Elements From an Existing Dictionary

- **To remove a key-value pair:**

  d.pop(key)

  - If key is not in the dictionary, `KeyError` exception is raised
  - OR `del` *`dictionary`*`[`*`key`*`]`

Getting the Number of Elements and Mixing Data Types

- `len` function: used to obtain number of key-value pairs in a dictionary
- Keys must be immutable objects, but associated values can be any type of object
  - One dictionary can include keys of several different immutable types. Heterogeneous.
- Values stored in a single dictionary can be of different types

Creating an Empty Dictionary and Using `for` Loop to Iterate Over a Dictionary

- **To create an empty dictionary:**
  - Use `{}`
  - Use built-in function `dict()`
  - Elements can be added to the dictionary as program executes
- **Use a `for` loop to iterate over a dictionary**
  - General format: `for key in dictionary:`

Some Dictionary Methods

- **`clear` method: deletes all the elements in a dictionary, leaving it empty**
  - Format: `dictionary.clear()`
- **`get` method: gets a value associated with specified key from the dictionary**
  - Format: `dictionary.get(key, default)`
    - *`default`* is returned if *`key`* is not found
  - Alternative to `[]` operator
    - Cannot raise `KeyError` exception

Some Dictionary Methods (cont'd.)

- **`items` method: returns all the dictionaries keys and associated values**
  - Format: *`dictionary`*`.items()`
  - Returned as a *dictionary view*
    - Each element in dictionary view is a tuple which contains a key and its associated value
    - Use a `for` loop to iterate over the tuples in the sequence
    - Can use a variable which receives a tuple, or can use two variables which receive key and value

Some Dictionary Methods (cont'd.)

- **<u>`keys` method</u>: returns all the dictionaries keys as a sequence**
  - Format: *`dictionary`*`.keys()`
- **<u>`pop` method</u>: returns value associated with specified key and removes that key-value pair from the dictionary**
  - Format: *`dictionary`*`.pop(`*`key, default`*`)`
    - *`default`* is returned if *`key`* is not found

Some Dictionary Methods (cont'd.)

- `popitem` **method: returns a randomly selected key-value pair and removes that key-value pair from the dictionary**
  - Format: *`dictionary`*`.popitem()`
  - Key-value pair returned as a tuple
- `values` **method: returns all the dictionaries values as a sequence**
  - Format: *`dictionary`*`.values()`
  - Use a `for` loop to iterate over the values

Some Dictionary Methods (cont'd.)

Table 9-1 Some of the dictionary methods

Method	Description
`clear`	Clears the contents of a dictionary.
`get`	Gets the value associated with a specified key. If the key is not found, the method does not raise an exception. Instead, it returns a default value.
`items`	Returns all the keys in a dictionary and their associated values as a sequence of tuples.
`keys`	Returns all the keys in a dictionary as a sequence of tuples.
`pop`	Returns the value associated with a specified key and removes that key-value pair from the dictionary. If the key is not found, the method returns a default value.
`popitem`	Returns a randomly selected key-value pair as a tuple from the dictionary and removes that key-value pair from the dictionary.
`values`	Returns all the values in the dictionary as a sequence of tuples.

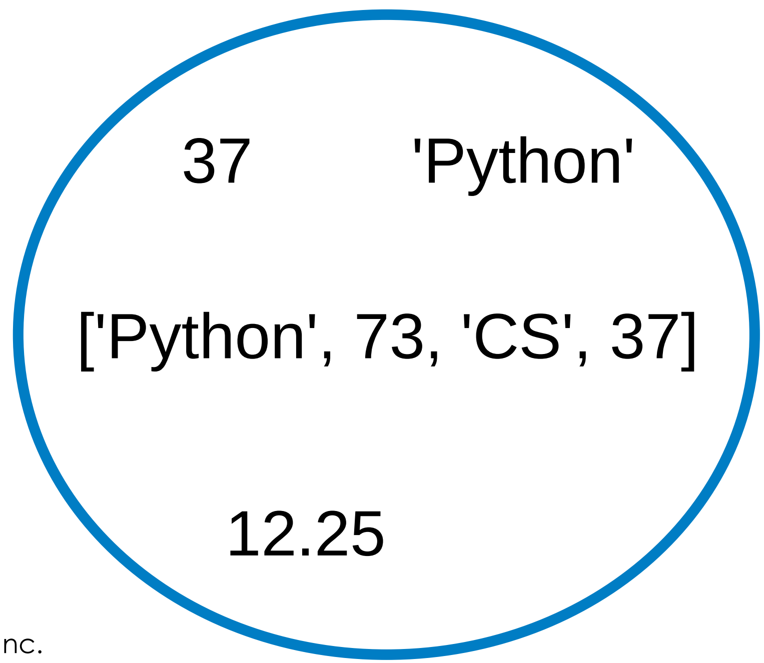
Dictionary Example

- Use a dictionary to determine which "word" occurs the most in a text.
- What will be the keys?
- What will be the values?

Sets

- **<u>Set</u>: object that stores a collection of data in same way as mathematical set**
  - Items are unique, duplicates don’t' exist in a set
  - Set is unordered, from the client's perspective
  - Elements can be of different data types

A Set



Creating a Set

- **`set` function: used to create a set**
  - Simple set creation
    - **`set1 = {12, 'Python', 37, 73}`**
  - For empty set, call `set()`
  - For non-empty set, call `set(`*`argument`*`)` where *`argument`* is an object that contains iterable elements
    - e.g., *`argument`* can be a list, string, or tuple
    - If *`argument`* is a string, each character becomes a set element
      - For set of strings, pass them to the function as a list
    - If *`argument`* contains duplicates, only one of the duplicates will appear in the set

Creating Data Types

- **List:**
  - `data = [7, 37, 5, 37, 12, 37.5]`
- **List of lists:**
  - `table = [[1, 2], [3, 7], [19, 73]]`
- **String:**
  - `name = 'Python Language'`
- **Tuple:**
  - `tup1 = (37, 'Python', 73, 12, 12)`
- **Dictionary:**
  - `freq_map = {'Python': 3, 'Java': 7}`
- **Set:**
  - `lang_set= {'Python', 'Java', 'C++'}`

Sets are Unordered

- Unlike the keys of a dictionary (which are a set, no duplicates), the elements in a Python set are unordered from the client's perspective.

```
>>> lang_set= {'Python', 'Java', 'C++',
...               12, 'Swift', 37, 12}
>>> lang_set
{'Java', 'Python', 37, 'C++', 'Swift', 12}
```

Getting the Number of and Adding Elements

- **`len` function: returns the number of elements in the set**
- **Sets are mutable objects**
- **`add` method: adds an element to a set**
  - What if set already contains that element?
- **`update` method: adds a group of elements to a set**
  - Argument must be a sequence containing iterable elements, and each of the elements is added to the set

Deleting Elements From a Set

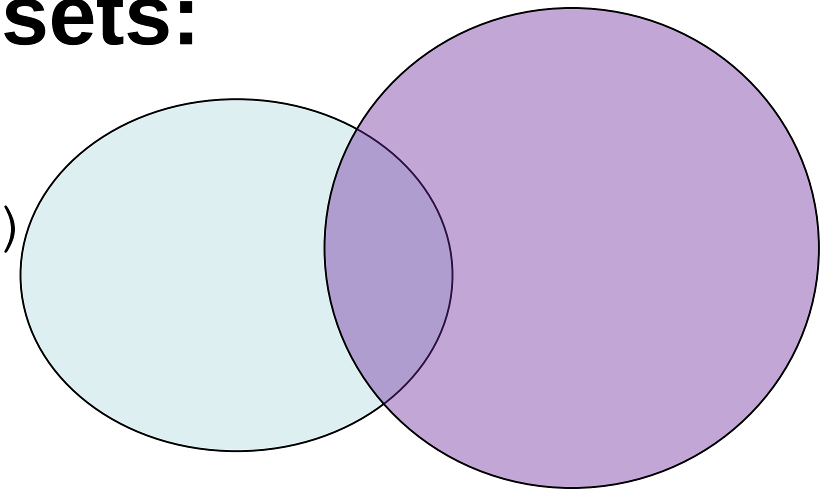
- **`remove` and `discard` methods: remove the specified item from the set**
  - The item that should be removed is passed to both methods as an argument
  - Behave differently when the specified item is not found in the set
    - `remove` method raises a `KeyError` exception
    - `discard` method does not raise an exception
- **`clear` method: clears all the elements of the set**

Using the `for` Loop, `in`, and `not in` Operators With a Set

- **A `for` loop can be used to iterate over elements in a set**
  - General format: `for` *`item`* `in` *`set`*`:`
  - The loop iterates once for each element in the set
- **The `in` operator can be used to test whether a value exists in a set**
  - Similarly, the `not in` operator can be used to test whether a value does not exist in a set

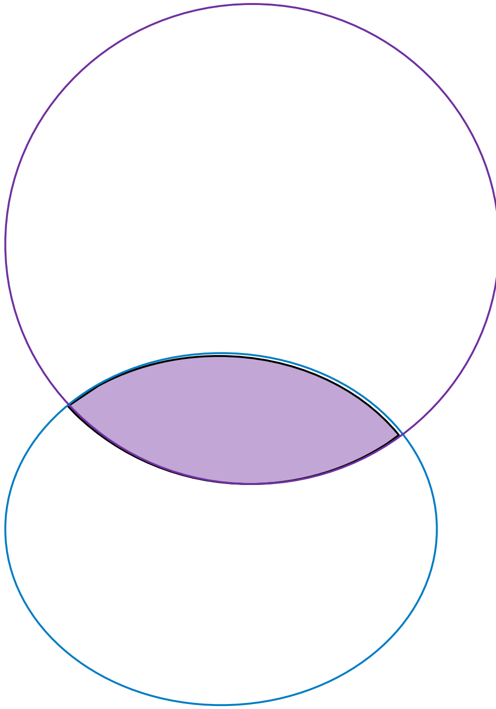
Finding the Union of Sets

- **<u>Union of two sets</u>: a set that contains all the elements of both sets**
- **To find the union of two sets:**
  - Use the `union` method
    - Format: *`set1`*`.union(`*`set2`*`)`
  - Use the `|` operator
    - Format: *`set1`*` | `*`set2`*
  - Both techniques return a new set which contains the union of both sets



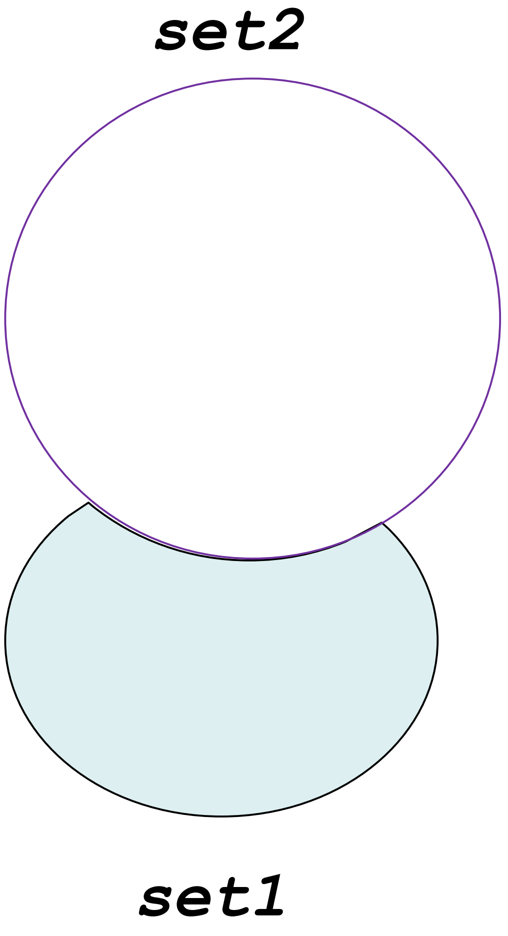
Finding the Intersection of Sets

- **<u>Intersection of two sets</u>: a set that contains only the elements found in both sets**
- **To find the intersection of two sets:**
  - Use the `intersection` method
    - Format: `set1.intersection(set2)`
  - Use the `&` operator
    - Format: `set1 & set2`
  - Both techniques return a new set which contains the intersection of both sets



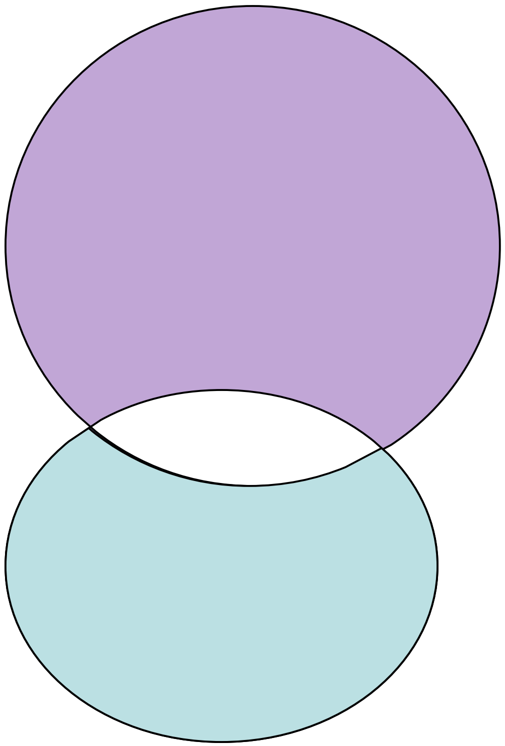
Finding the Difference of Sets

- **<u>Difference of two sets</u>: a set that contains the elements that appear in the first set but do not appear in the second set**
- **To find the difference of two sets:**
  - Use the `difference` method
    - Format: `set1.difference(set2)`
  - Use the – operator
    - Format: `set1 - set2`



Finding the Symmetric Difference of Sets

- **<u>Symmetric difference of two sets</u>: a set that contains the elements that are not shared by the two sets**
- **To find the symmetric difference of two sets:**
  - Use the `symmetric_difference` method
    - Format: `set1.symmetric_difference(set2)`
  - Use the `^` operator
    - Format: `set1 ^ set2`



Finding Subsets and Supersets

- **Set A is subset of set B if all the elements in set A are included in set B**
- **To determine whether set A is subset of set B**
  - Use the `issubset` method
    - Format: `setA.issubset(setB)`
  - Use the <= operator
    - Format: `setA <= setB`

Finding Subsets and Supersets (cont'd.)

- **Set A is superset of set B if it contains all the elements of set B**
- **To determine whether set A is superset of set B**
  - Use the `issuperset` method
    - Format: `setA.issuperset(setB)`
  - Use the `>=` operator
    - Format: `setA >= setB`

Serializing Objects

- Serialize an object: convert the object to a stream of bytes that can easily be stored in a file
- Pickling: serializing an object

Serializing Objects (cont'd.)

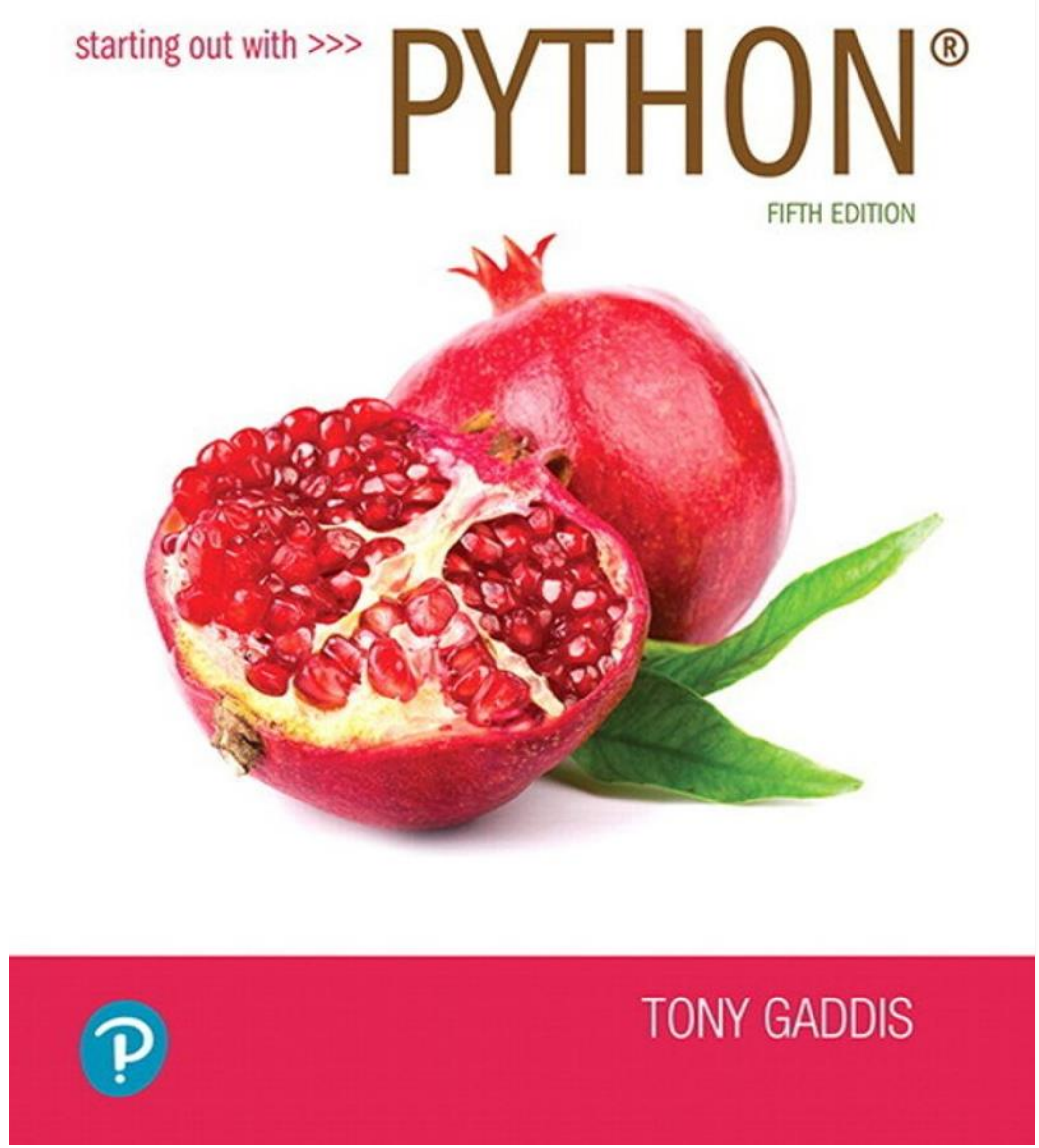
- **To pickle an object:**
  - Import the `pickle` module
  - Open a file for binary writing, 'wb' option
  - Call the `pickle.dump` function
    - Format: `pickle.dump(`*`object, file`*`)`
  - Close the file
- **You can pickle multiple objects to one file prior to closing the file**

Serializing Objects (cont'd.)

- **<u>Unpickling</u>: retrieving pickled object**
- **To unpickle an object:**
  - Import the `pickle` module
  - Open a file for binary writing, 'rb'
  - Call the `pickle.load` function
    - Format: `pickle.load(`*`file`*`)`
  - Close the file
- **You can unpickle multiple objects from the file**

CHAPTER 10

Classes and Object-Oriented Programming



Pearson

Procedural Programming

- Procedures: synonym for **functions** and sub-routines
- **Procedural programming: writing programs made of functions that perform specific tasks**
  - Functions typically operate on data items that are separate from the functions
  - Data items commonly passed from one function to another
  - Focus: On the algorithm and steps. Create functions that operate on the program’s data

Object-Oriented Programming

- **Object-oriented programming: focuses on creating classes and objects**
- **Model the problem on the data involved first, not the big steps.**
- **Class: A programmer defined data type**
- **Object: entity that contains data and functions**
  - Data is known as data attributes and functions are known as methods
    - Methods perform operations on the data attributes
- **Encapsulation: combining data and code into a single object**

Object Oriented Programming

- Recall a CPU only knows how to perform on the **order of 100 operations**
- High level languages such as Python allow us to, seemingly, **create new operations** by defining new functions
- Object oriented languages allow programmers to **create new data types** in addition to the ones built into the language
  - int, float, string, list, tuple, file, dictionary, set

Object Oriented Design Example - Monopoly

If we had to start from scratch what new data types would we need to create?

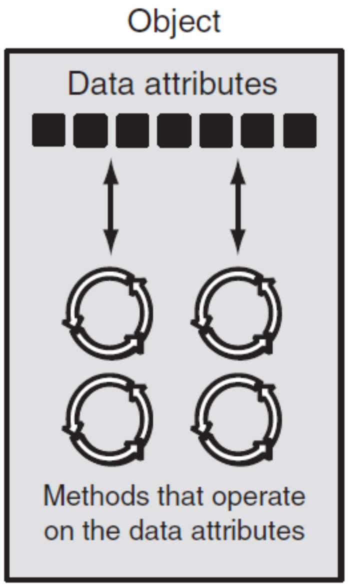
Data Types Needed:

Object Orientation

- **The basic idea of object oriented programming (OOP) is to view your problem as a *collection of objects*, each of which has certain state and can perform certain actions.**
- **Each object has:**
  - **some *data* that it maintains characterizing its current state;**
  - **a set of actions (*methods*) that it can perform.**
- **A programmer interacts with an object by calling its methods; this is called *method invocation*. That should be the *only way* that another programmer interacts with an object.**
- **Significant object-oriented languages include Python, Java, C++, C#, Perl, JavaScript, Objective C, and others.**

Object-Oriented Programming (cont'd.)

Figure 10-1 An object contains data attributes and methods

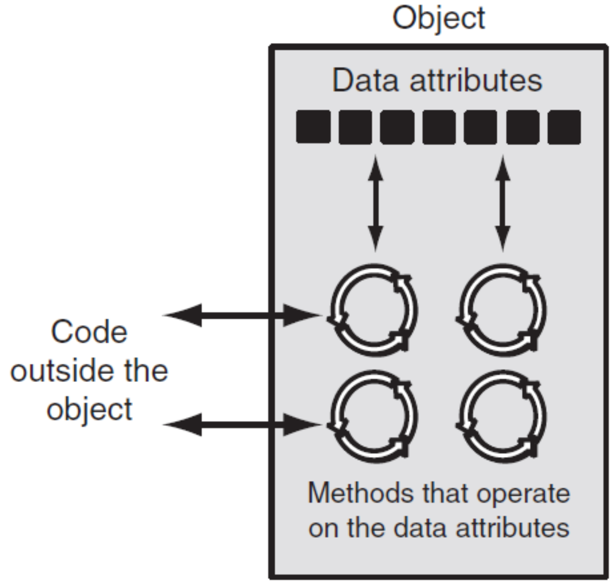


Object-Oriented Programming (cont'd.)

- **Data hiding: object's data attributes are hidden from code outside the object**
  - Access restricted to the object's methods
    - Protects from accidental corruption
    - Outside code does not need to know internal structure of the object
- **Object reusability: the same object can be used in different programs**
  - Example: 3D image object can be used for architecture and game programming

Object-Oriented Programming (cont'd.)

Figure 10-2 Code outside the object interacts with the object's methods



An Everyday Example of an Object

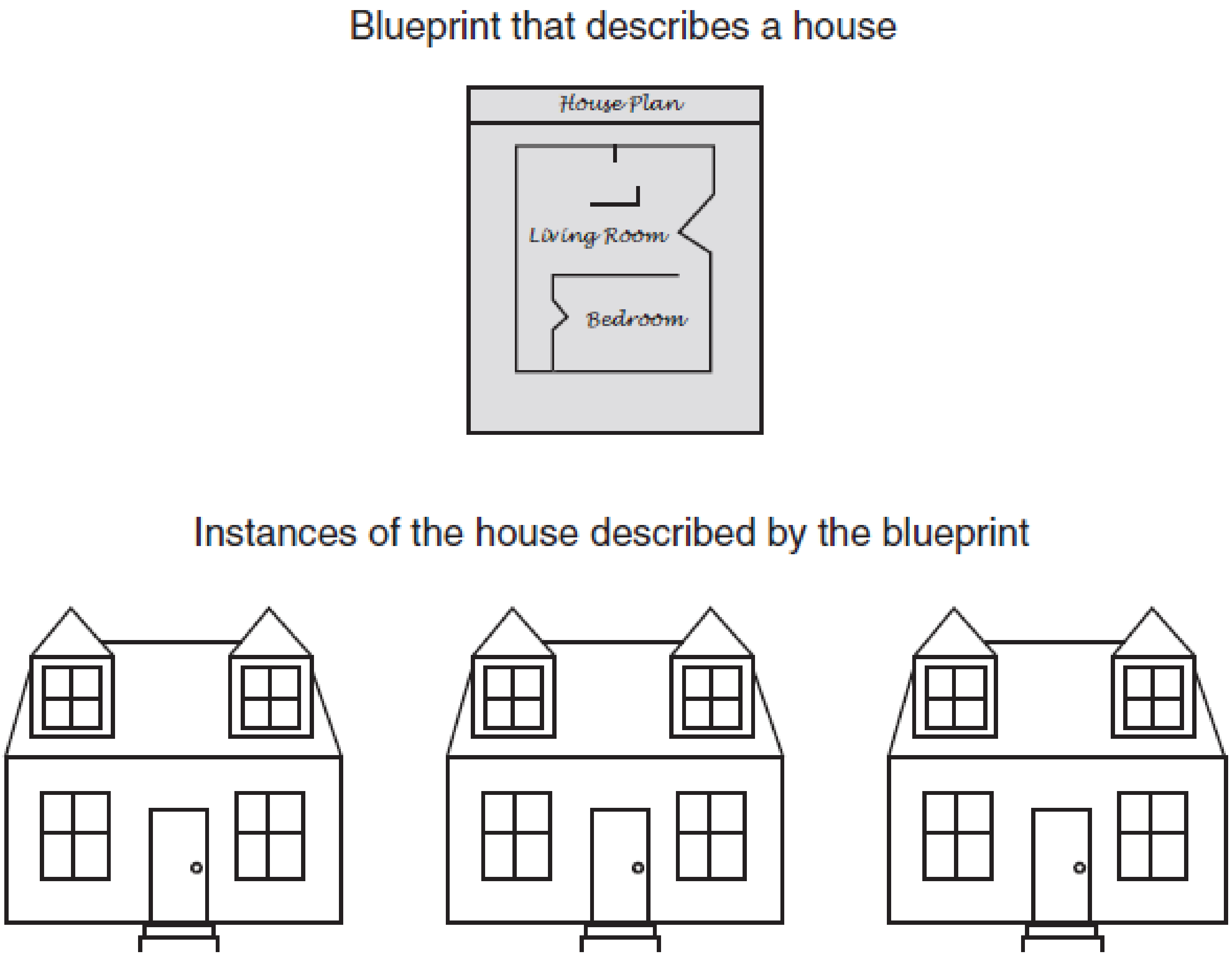
- **Data attributes: define the state of an object**
  - Example: clock object would have `second`, `minute`, and `hour` data attributes
- **Public methods: allow external code to manipulate the object**
  - Example: `set_time`, `set_alarm_time`
- **Private methods: used for object's inner workings**

Classes

- **<u>Class</u>: code that specifies the data attributes and methods of a particular type of object**
  - Similar to a blueprint of a house or a cookie cutter
- **<u>Instance</u>: an object created from a class**
  - Similar to a specific house built according to the blueprint or a specific cookie
  - There can be many instances of one class

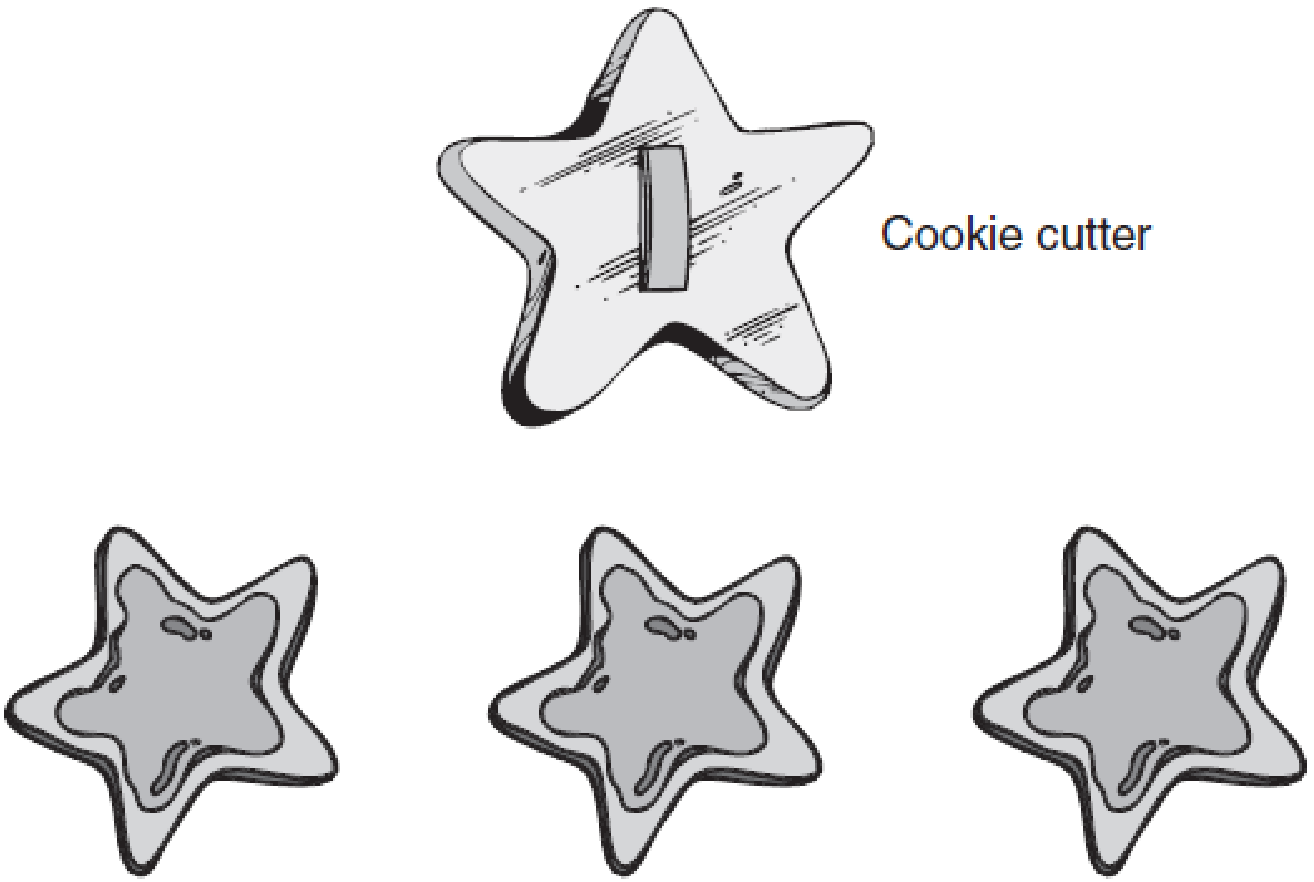
Classes

A blueprint and houses built from the blueprint



Classes

The cookie cutter metaphor



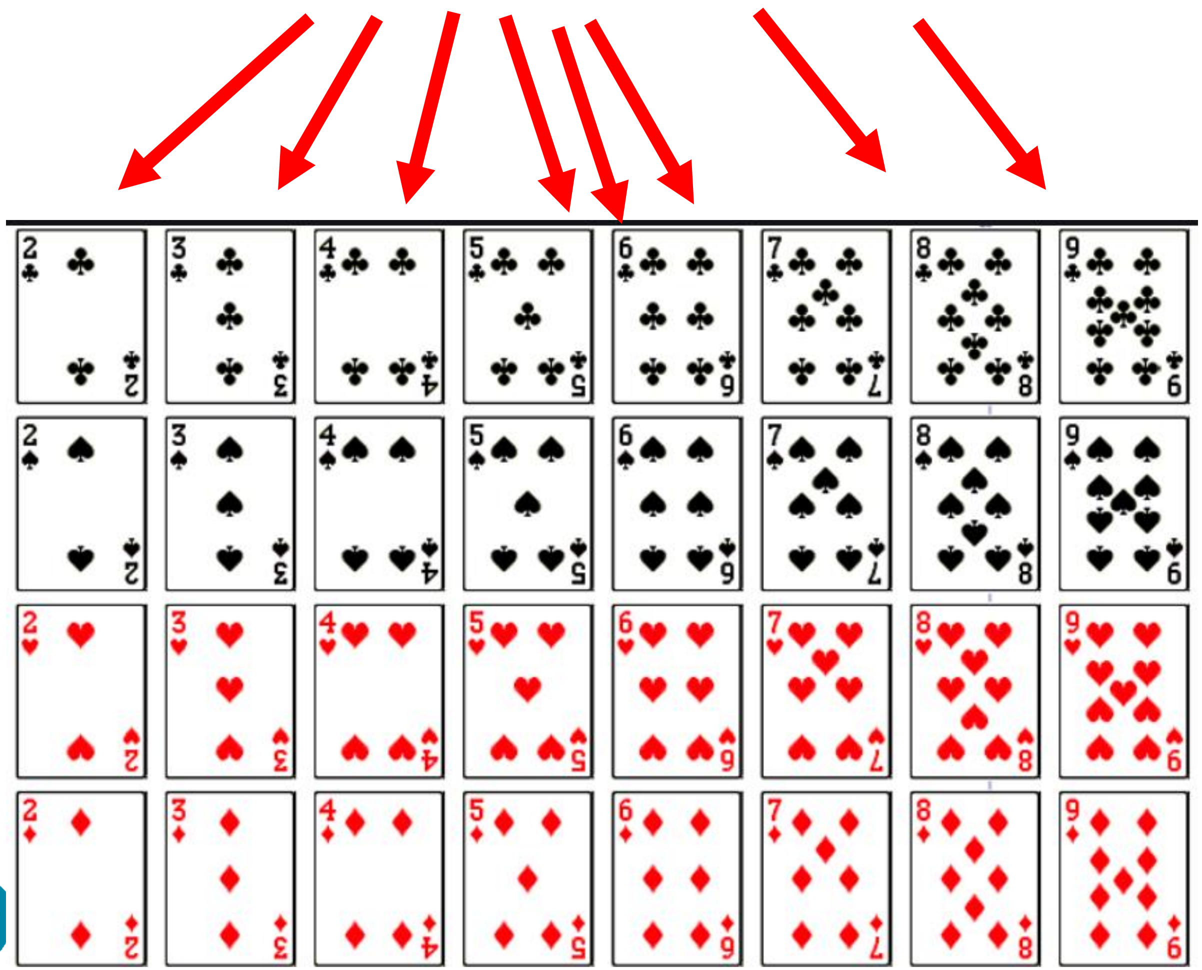
Simple Example

Class Definition for Playing cards

Playing cards have:

A Rank

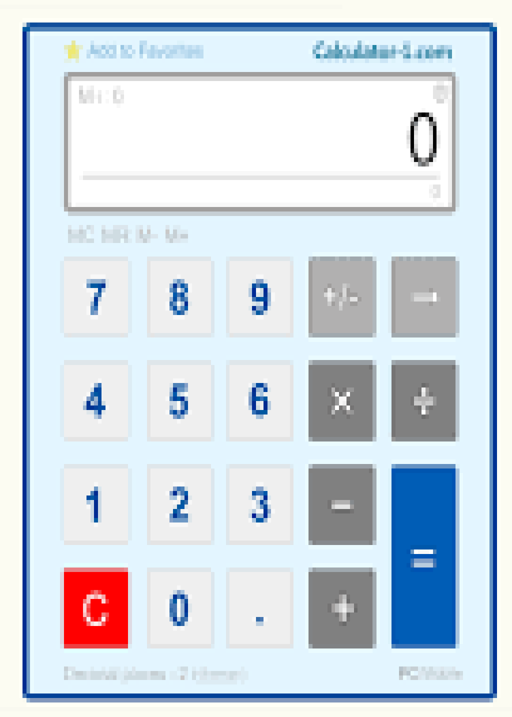
A Suit



Define a PlayingCard class and then create objects of type PlayingCard to form a deck or a hand of cards.

A Concrete Example

- **Imagine that you're trying to do some simple arithmetic. You need a Calculator application, programmed in an OO manner. It will have:**
- **Some data: the current value of its**
  - *accumulator* (the value stored and displayed on the screen).
  - History of ops?
  - Memory?
- **Some methods: things that you can ask of the calculator to do:**
  - add a number to the accumulator, subtract a number, multiply by a number, divide by a number, zero out the accumulator value, etc.



Calculator Specification

- In Python, you implement a particular type of object (soda machine, calculator, etc.) with a `class`.
- Let's define a class for our simple interactive calculator.
- Data: the current value of the accumulator.
  Maybe a history of operations? Memory spots, aka variables?
- Methods: any of the following.
  - clear: zero the accumulator
  - print: display the accumulator value
  - add k: add k to the accumulator
  - sub k: subtract k from the accumulator
  - mult k: multiply accumulator by k
  - div k: divide accumulator by k

Yet Another Example

- Example: A soda machine has:
  - **Data:** products inside, change available, amount previously deposited, etc.
- **Methods:** accept a coin, select a product, dispense a soda, provide change after purchase, return money deposited, etc.
- Assignment 13

Class Definitions

- **<u>Class definition</u>: set of statements that define a class's methods and data attributes**
  - Format: begin with `class` *`ClassName`*:
    - <u>Class names typically start with uppercase letter and internal words are capitalized, aka CamelCase</u>
  - Method definition like other Python function definitions
    - <u>`self` parameter</u>: required in every method in the class – references the specific object that the method is working on - **The object the method is working on. The object that called the method**
      name = 'Olivia'
      name.upper() # name is the argument to self

Class Definitions (cont'd.)

- **Initializer method: automatically executed when an instance of the class is created**
  - Initializes object's data attributes and assigns `self` parameter to the object that was just created.
  - Format: `def __init__ (self):`
  - That's two underscores before and after `init`.
  - Typically the first method in a class definition.

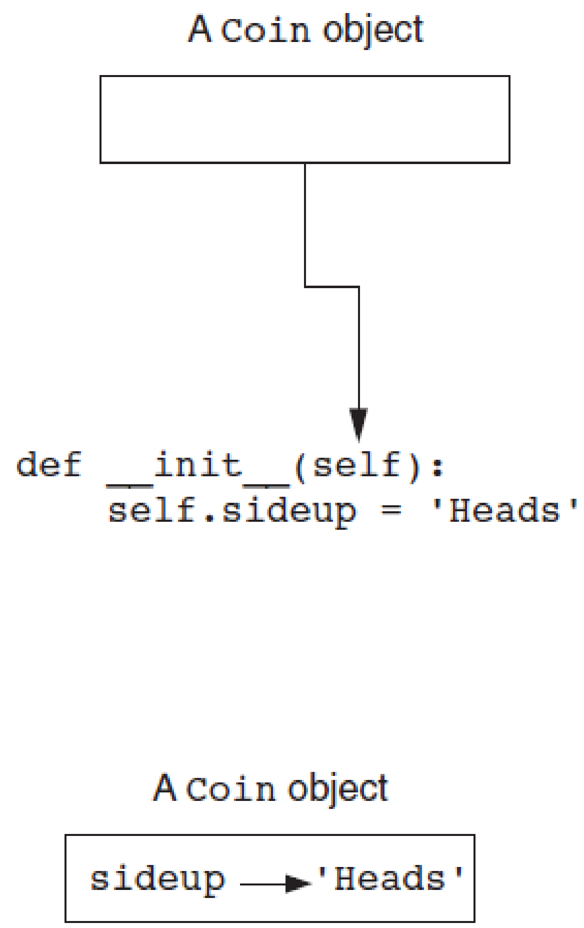
Class Definitions (cont'd.)

Actions caused by the `Coin()` expression

1. An object is created in memory from the `Coin` class.

2. The `Coin` class's `__init__` method is called, and the `self` parameter is set to the newly created object

After these steps take place, a `Coin` object will exist with its `sideup` attribute set to `'Heads'`.



Class Definitions (cont'd.)

- **To create a new instance of a class call the initializer method**
  - Format: `my_instance = ClassName()`
- **To call any of the class methods using the created instance, use dot notation**
  - Format: `my_instance.method()`
  - Because the `self` parameter references the specific instance of the object, the method will affect this instance
    - Reference to `self` is passed automatically

Hiding Attributes and Storing Classes in Modules

- **An object's data attributes (aka the internal variables) should be difficult to access**
  - To make sure of this, place two underscores (__) in front of attribute name
    - Example: `__current_minute`
- **Classes can be stored in modules**
  - Filename for module must end in .py
  - Module can be imported to programs that use the class

The Circle Class - in Circle.py

```
import math


class Circle:
    """Model a simple circle.

    Each circle has a center point expressed as x and y coordinates
    and a radius."""

    def __init__(self, x=0, y=0, radius=0):
        self.__x = x
        self.__y = y
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def get_x(self):
        return self.__x

    def get_y(self):
        return self.__y
```

The Circle Class - in Circle.py

```
def get_area(self):
    return self.__radius ** 2 * math.pi

def get_perimeter(self):
    return 2 * self.__radius * math.pi

def contains(self, other_circle):
    """Return if other_circle is contained wholly in this Circle."""
    distance = ((self.__x - other_circle.__x) ** 2
                + (self.__y - other_circle.__y) ** 2)
    distance = math.sqrt(distance)
    return distance + other_circle.__radius <= self.__radius

def __str__(self):
    return ('x: ' + str(self.__x) + ', y: ' + str(self.__y)
            + ', radius: ' + str(self.__radius))
```

Client Code of Circle Class

```
c1 = Circle(1, 2, 4)
print(c1.__radius)  # causes runtime error
print(c1.__x)  # causes runtime error
c1.__radius = 5
print(str(c1))
c2 = Circle(3, 1, 1)
print(c1.contains(c2))
print(c2.contains(c1))
```

- **Recall, variables prefixed with the double underscore (_ _) are hidden from clients.**
- **Careful, easy to create logic errors**

Logic Error in Client Code

- **Clients can add attributes (internal data, internal variables) to objects**
- **Flexible? Yes. Dangerous? You bet!**

```
c2 = Circle(3, 1, 1)  # x, y, radius
c2.__x = 12
print('c2.__x in client code', c2.__x)
print('c2.get_x(), in client code', c2.get_x())
print('Result of print(c2) in client code:')
print(c2)
```

```
c2.__x in client code 12
c2.get_x(), in client code 3
x: 3, y: 1, radius: 1
```

The `BankAccount` Class – More About Classes

- **Class methods can have multiple parameters in addition to `self`**
  - For `__init__`, parameters needed to create an instance of the class
    - Example: a `BankAccount` object is created with a balance
      - When called, the initializer method receives a value to be assigned to a `__balance` attribute
  - For other methods, parameters may be needed to perform required task
    - Example: `deposit` method amount to be deposited

The `__str__` method

- <u>Object's state</u>: the values of the object's attribute at a given moment
- `__str__` method: return a string version of the object, typically the state of its internal data
- Automatically called when the object is passed as an argument to the `print` function
- Automatically called when the object is passed as an argument to the `str` function

Working With Instances

- **<u>Instance attribute</u>: belongs to a specific instance of a class**
  - Created when a method uses the `self` parameter to create an attribute
  - Can be local to a method, but continues to exist after that method completes
- **If many instances of a class are created, each would has its own set of attributes**

Figure 10-8 The `coin1`, `coin2`, and `coin3` variables reference three `Coin` objects

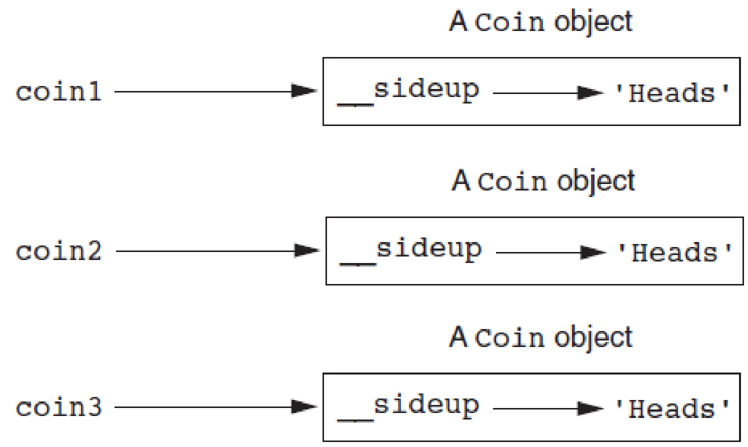
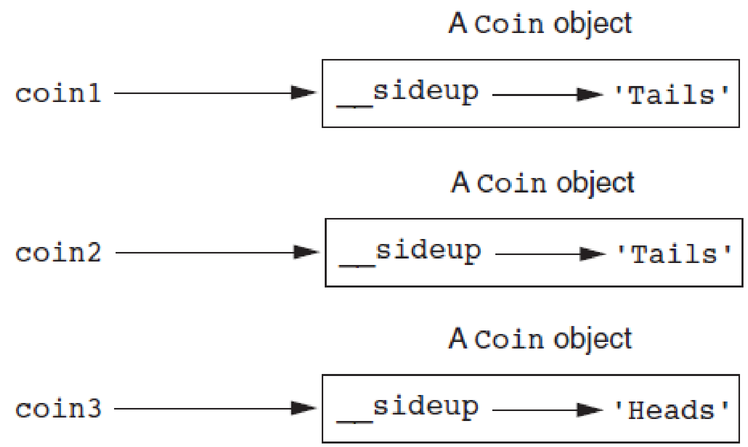


Figure 10-9 The objects after the `toss` method



Accessor and Mutator Methods

- **Typically, all of a class's data attributes are private and provide methods to access and change them**
- **<u>Accessor methods</u>: return a value from a class's attribute without changing it**
  - Safe way for code outside the class to retrieve the value of attributes
- **<u>Mutator methods</u>: store or change the value of a data attribute**
  - You **DO NOT** have to have mutator methods for all (or any) internal attributes

Passing Objects as Arguments

- **Methods and functions often need to accept objects as arguments**
- **When you pass an object as an argument, you are actually passing a reference to the object**
  - The receiving method or function has access to the actual object
    - Methods of the object can be called within the receiving function or method, and data attributes may be changed using mutator methods

Other methods

- generally methods with the _ _name_ _ format are not meant to be called directly
- Instead we define them and then the are called with other operators

_ _init_ _	ClassName()		
_ _len_ _	len()		
_ _str_ _	str		
_ _add_ _	+	_ _eq_ _	==
_ _lt_ _	<	_ _le_ _	<=
_ _gt_ _	>	_ _ge_ _	>=

Displaying New Classes in Data Structures

```
c1 = Circle(3, 1, 1)
c2 = Circle(5, 4, 3)
print(c1, c2)
data1 = [c1, c2]
print(data1)
```

Output of print. Great!

```
x: 3, y: 1, radius: 1 x: 5, y: 4, radius: 3
[<__main__.Circle object at 0x000001E56D308640>,
 <__main__.Circle object at 0x000001E56D308670>]
```

Output of print of list. Yuck!

_ _str_ _ and _ _ repr_ _

- print calls the _ _str_ _ method on objects sent to it
- a data structure calls the _ _repr_ _ method on the objects inside it to
- repr for representation
- Like _ _str_ _ but should display the object in a way that we could use to rebuild the object

__repr__ method for Circle

```python
def __repr__(self):
    result = ('Circle(x=' + str(self.__x) + ", y=" + str(self.__y)
              + ', radius=' + str(self.__radius) + ')')
    return result
```

```python
c1 = Circle(3, 1, 1)
c2 = Circle(5, 4, 3)
print(c1, c2)
data1 = [c1, c2]
print(data1)
```

```
x: 3, y: 1, radius: 1 x: 5, y: 4, radius: 3
[Circle(x=3, y=1, radius=1), Circle(x=5, y=4, radius=3)]
```

Techniques for Designing Classes

- **UML diagram: standard diagrams for graphically depicting object-oriented systems**
  - Stands for Unified Modeling Language
- **General layout: box divided into three sections:**
  - Top section: name of the class
  - Middle section: list of data attributes
  - Bottom section: list of class methods

Figure 10-10 General layout of a UML diagram for a class

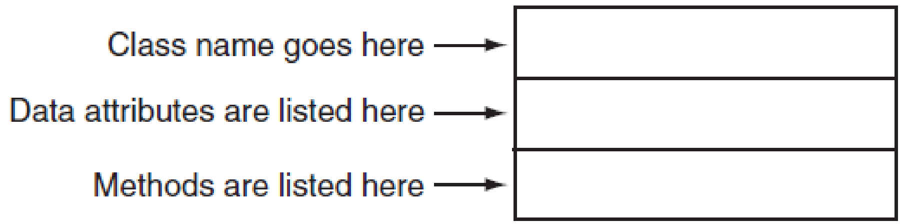
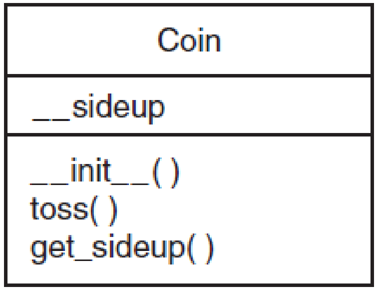


Figure 10-11 UML diagram for the `coin` class



Finding the Classes in a Problem

- **When developing object oriented program, first goal is to identify classes**
  - Typically involves identifying the real-world objects that are in the problem
  - Technique for identifying classes:
    1. Get written description of the problem domain
    2. Identify all nouns in the description, each of which is a potential class
    3. Refine the list to include only classes that are relevant to the problem

Finding the Classes in a Problem (cont'd.)

1. **Get written description of the problem domain**
   - May be written by you or by an expert
   - Should include any or all of the following:
     - Physical objects simulated by the program
     - The role played by a person
     - The result of a business event
     - Recordkeeping items

Finding the Classes in a Problem (cont'd.)

2. **Identify all nouns in the description, each of which is a potential class**
   - Should include noun phrases and pronouns
   - Some nouns may appear twice

Finding the Classes in a Problem (cont'd.)

3. **Refine the list to include only classes that are relevant to the problem**
   - Remove nouns that mean the same thing
   - Remove nouns that represent items that the program does not need to be concerned with
   - Remove nouns that represent objects, not classes
   - Remove nouns that represent simple values that can be assigned to a variable

Identifying a Class's Responsibilities

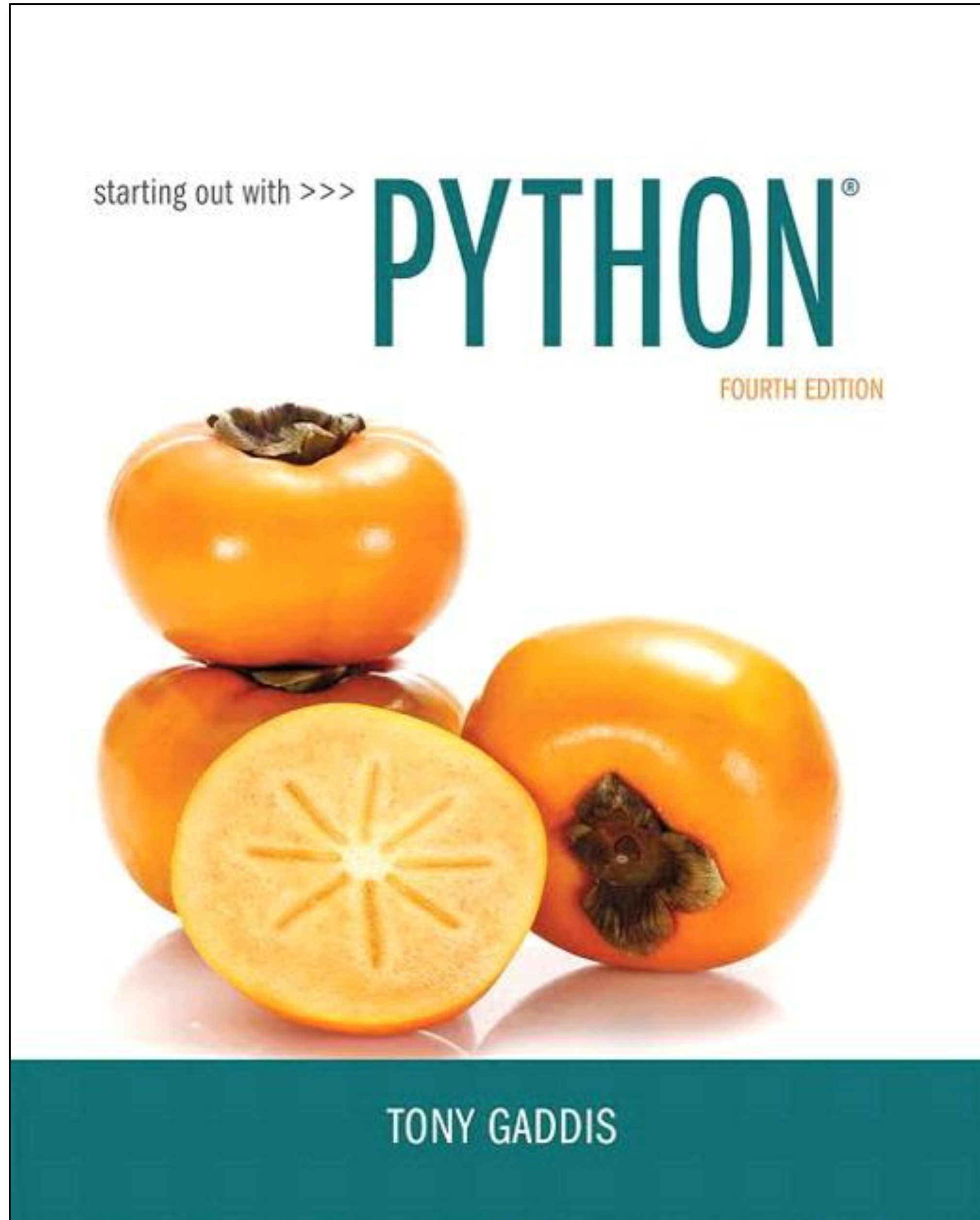
- **A classes responsibilities are:**
  - The things the class is responsible for knowing
    - Identifying these helps identify the class's data attributes
  - The actions the class is responsible for doing
    - Identifying these helps identify the class's methods
- **To find out a class's responsibilities look at the problem domain**
  - Deduce required information and actions

Summary

- **This chapter covered:**
  - Procedural vs. object-oriented programming
  - Classes and instances
  - Class definitions, including:
    - The `self` parameter
    - Data attributes and methods
    - `__init__` and `__str__` functions
    - Hiding attributes from code outside a class
  - Storing classes in modules
  - Designing classes

CHAPTER 12

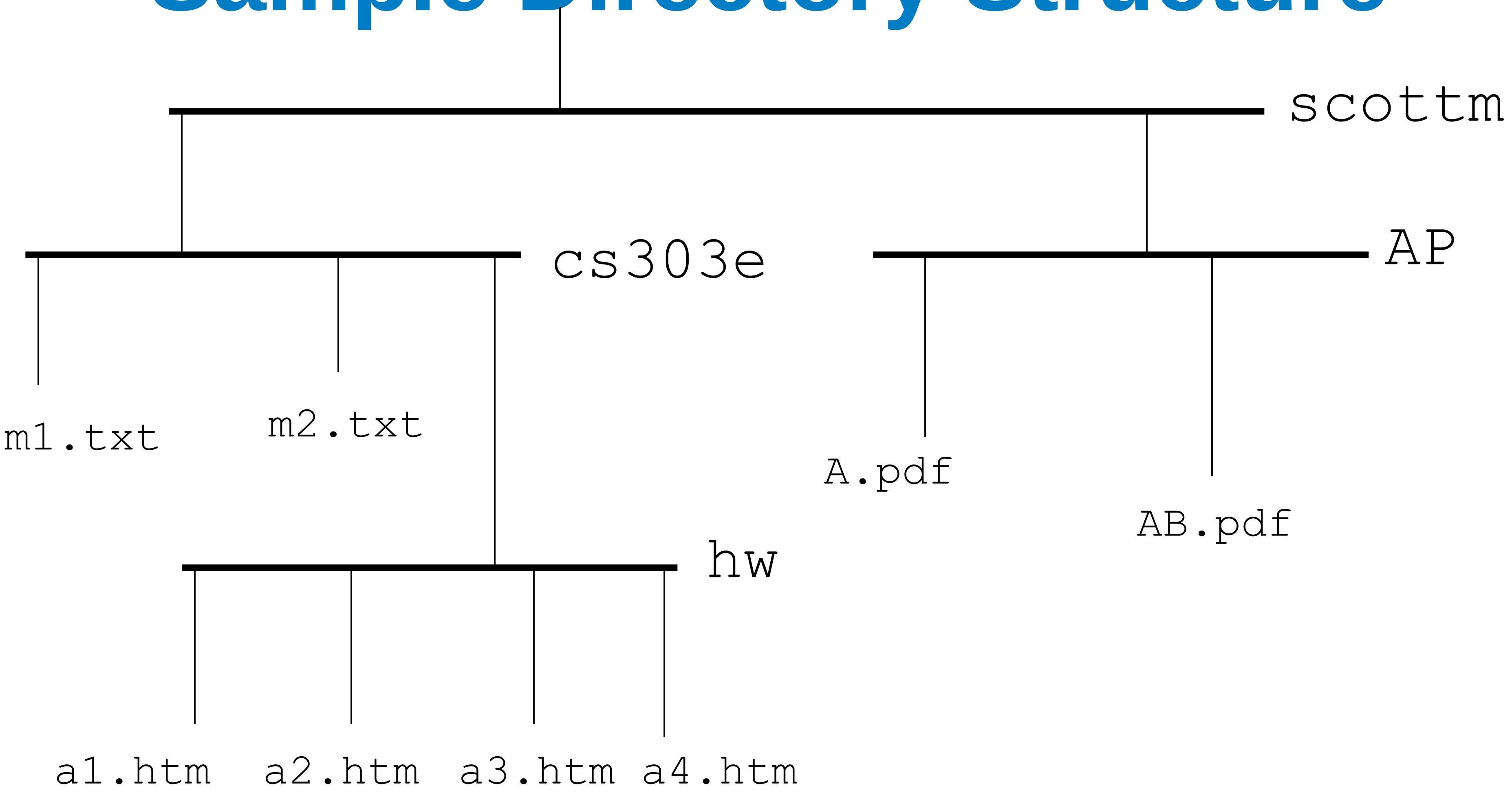
Recursion



An Interesting Problem

- **Write a method that determines how much space is take up by the files in a directory**
- **A directory can contain files and directories**
- **How many directories does our code have to examine?**
- **How would you add up the space taken up by the files in a single directory**
  - Hint: don't worry about any sub directories at first

Sample Directory Structure



os.path

- **We used os.path to check if a path (location of a file or directory) refers to a file that exists**
- **Lots of other useful methods:**
  - os.path.isfile(path)
  - os.path.isdir(path)
  - os.path.getsize(path)
    - Return the size, in bytes, of path. Raise OSError if the file does not exist or is inaccessible.
  - os.listdir(path='.')
    - Return a list containing the names of the entries in the directory given by path.

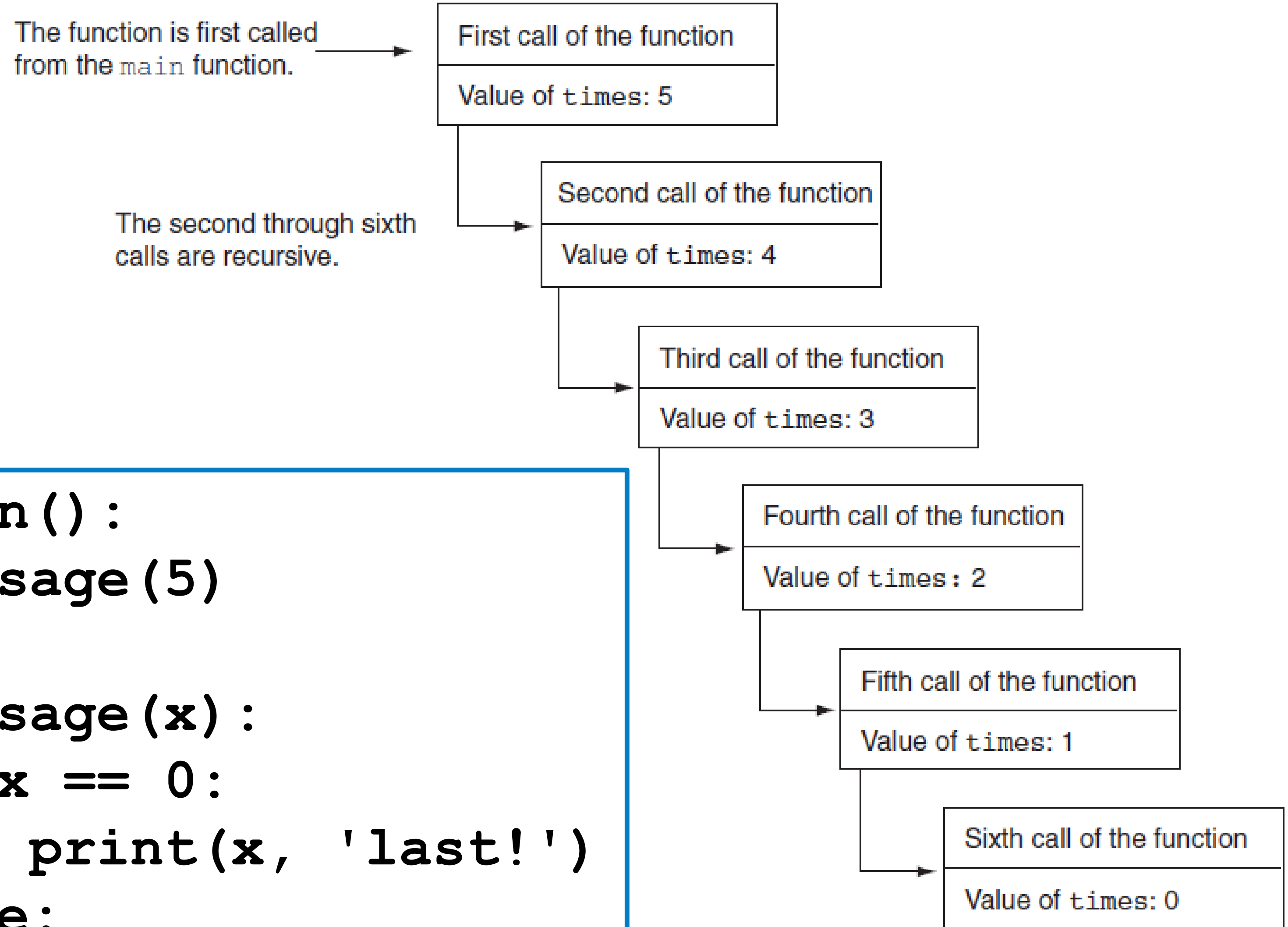
Implementation

- **Write a function that given the name of a directory returns the size of the files in that directory**
  - … and if the directory has directories in it (subdirectories) return the size of the files in those subdirectories
    - … and if those subdirectories have subdirectories…

Introduction to Recursion

- **Recursive function: a function that calls itself (with different arguments)**
- **Recursive function must have a way to control the number of times it repeats**
  - Usually involves an `if-else` statement which defines when the function should return a value and when it should call itself
- **Depth of recursion: the number of times a function calls itself**

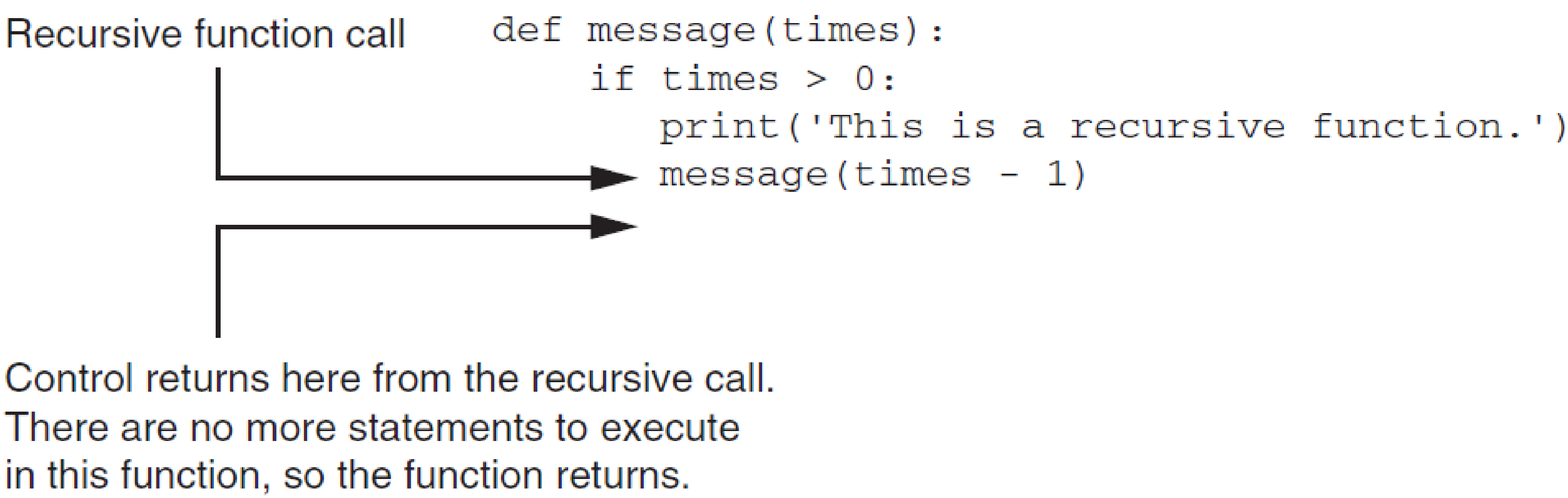
Figure 12-2 Six calls to the message function




```
def main():
    message(5)

def message(x):
    if x == 0:
        print(x, 'last!')
    else:
        print(x)
        message(x - 1)
```

Introduction to Recursion (cont'd.)

- Control returns to the point after the recursive function call



Problem Solving with Recursion

- **Recursion is a powerful tool for solving repetitive problems**
- **Recursion is never *required* to solve a problem**
  - Any problem that can be solved recursively can be solved with a loop
  - Recursive algorithms may be less efficient than iterative ones in the number of computations
    - Due to *overhead* of each function call

Problem Solving with Recursion (cont'd.)

- **Some repetitive problems are more easily solved with recursion**
- **General outline of recursive function:**
  - If the problem can be solved now without recursion, solve and return
    - Known as the *base case*
  - Otherwise, reduce problem to smaller problem of the same structure and call the function again to solve the smaller problem
    - Known as the *recursive case*

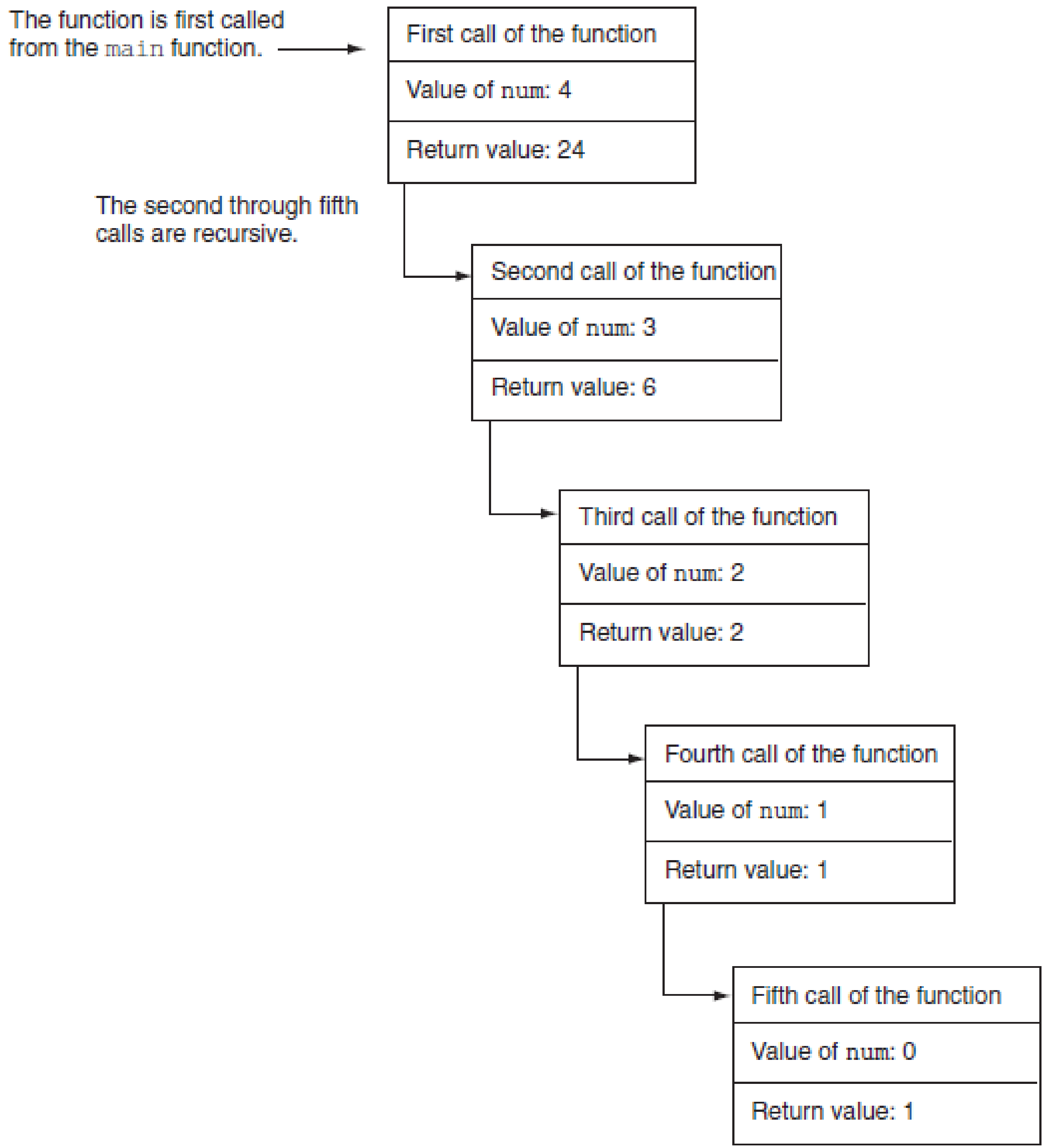
Using Recursion to Calculate the Factorial of a Number

- **In mathematics, the $n!$ notation represents the factorial of a number *n***
  - For $n = 0$, $n! = 1$
  - For $n > 0$, $n! = 1 \times 2 \times 3 \times \ldots \times n$
- **The above definition lends itself to recursive programming**
  - $n = 0$ is the base case
  - $n > 0$ is the recursive case
    - factorial(*n*) = n x factorial(n-1)

Using Recursion (cont'd.)

```
# The factorial function uses recursion to
# calculate the factorial of its argument,
# which is assumed to be nonnegative.
def factorial(num):
    if num == 0:
        return 1
    else:
        return num * factorial(num - 1)
```

Figure 12-4 The value of `num` and the return value during each call of the function



Using Recursion (cont'd.)

- **Since each call to the recursive function reduces the problem:**
  - Eventually, it will get to the base case which does not require recursion, and the recursion will stop
- **Usually the problem is reduced by making one or more parameters smaller at each function call**

Direct and Indirect Recursion

- **Direct recursion: when a function directly calls itself**
  - All the examples shown so far were of direct recursion
- **Indirect recursion: when function A calls function B, which in turn calls function A**
  - also known as mutual recursion

Examples of Recursive Algorithms

- **Summing a range of list elements with recursion**
  - Function receives a list containing range of elements to be summed, index of starting item in the range, and index of ending item in the range
  - Base case:
    - `if start index > end index return 0`
  - Recursive case:
    - `return current_number + sum(list, start+1, end)`

Examples of Recursive Algorithms (cont'd.)

```
# The range_sum function returns the sum of a specified
# range of items in num_list. The start parameter
# specifies the index of the starting item. The end
# parameter specifies the index of the ending item.
def range_sum(num_list, start, end):
    if start > end:
        return 0
    else:
        return num_list[start] + range_sum(num_list, start + 1, end)
```

The Fibonacci Series

- **Fibonacci series: has two base cases**
  - `if n = 0 then Fib(n) = 0`
  - `if n = 1 then Fib(n) = 1`
  - `if n > 1 then Fib(n) = Fib(n-1) + Fib(n-2)`
- **Corresponding function code:**

```
# The fib function returns the nth number
# in the Fibonacci series.
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Finding the Greatest Common Divisor

- **Calculation of the greatest common divisor (GCD) of two positive integers**
  - If x can be evenly divided by y, then
  - gcd(x,y) = y
  - Otherwise, gcd(x,y) = gcd(y, remainder of x/y)
- **Corresponding function code:**

```
# The gcd function returns the greatest common
# divisor of two numbers.
def gcd(x, y):
    if x % y == 0:
        return y
    else:
        return gcd(x, x % y)
```

The Towers of Hanoi

- **Mathematical game commonly used to illustrate the power of recursion**
  - Uses three pegs and a set of discs in decreasing sizes
  - Goal of the game: move the discs from leftmost peg to rightmost peg
    - Only one disc can be moved at a time
    - A disc cannot be placed on top of a smaller disc
    - All discs must be on a peg except while being moved

The Towers of Hanoi (cont'd.)

Figure 12-5 The pegs and discs in the Tower of Hanoi game

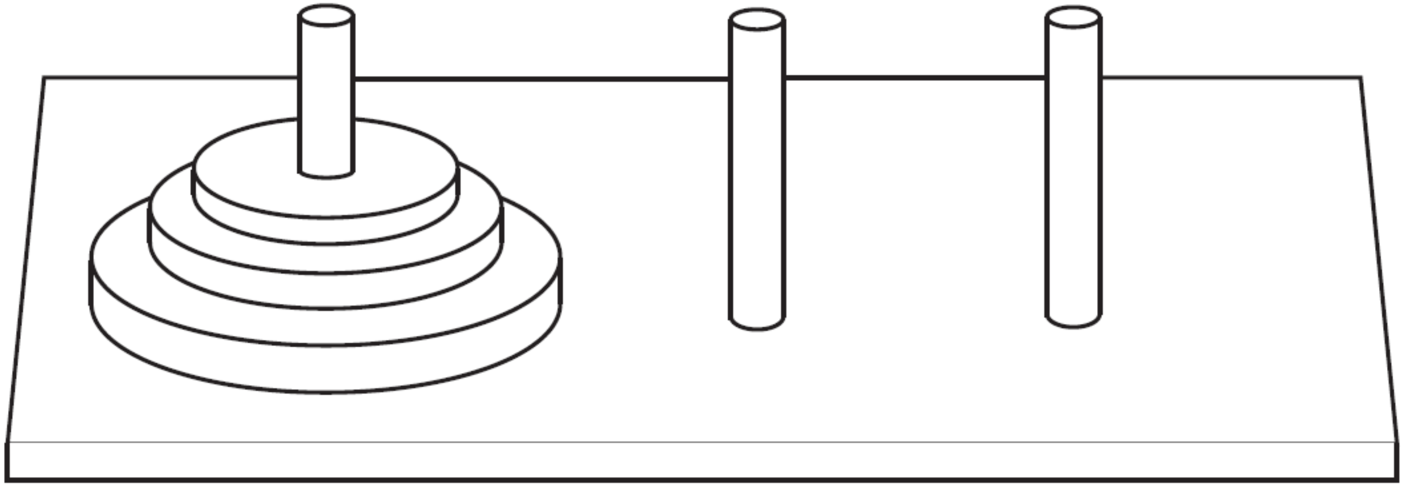
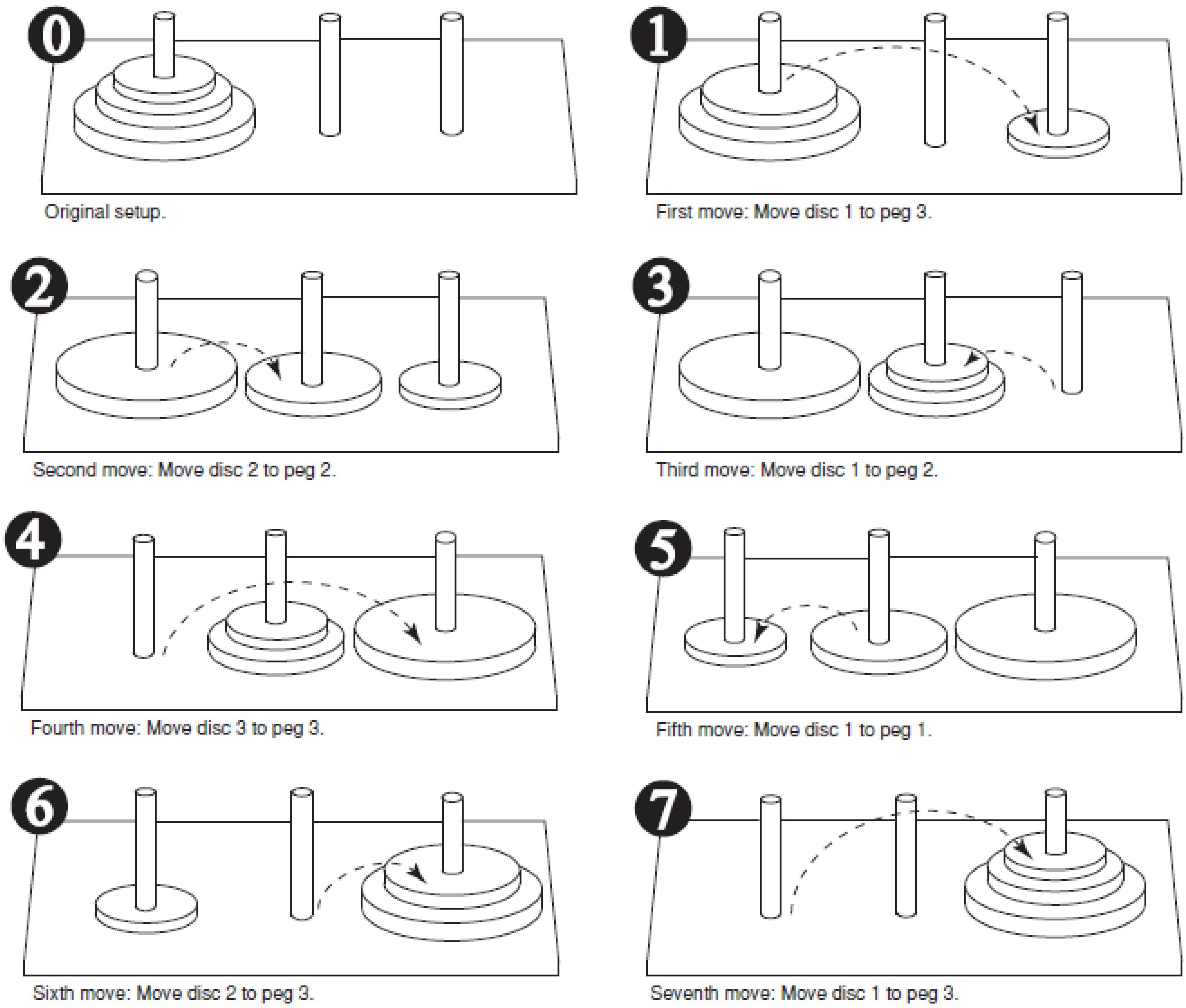


Figure 12-6 Steps for moving three pegs



The Towers of Hanoi (cont'd)

- **<u>Problem statement</u>: move n discs from peg 1 to peg 3 using peg 2 as a temporary peg**
- **<u>Recursive solution</u>:**
  - If n == 1: Move disc from peg 1 to peg 3
  - Otherwise:
    - Move n-1 discs from peg 1 to peg 2, using peg 3
    - Move remaining disc from peg 1 to peg 3
    - Move n-1 discs from peg 2 to peg 3, using peg 1

The Towers of Hanoi (cont'd.)

```
# The moveDiscs function displays a disc move in
# the Towers of Hanoi game.
# The parameters are:
#     num:          The number of discs to move.
#     from_peg:     The peg to move from.
#     to_peg:       The peg to move to.
#     temp_peg:     The temporary peg.
def move_discs(num, from_peg, to_peg, temp_peg):
    if num > 0:
        move_discs(num - 1, from_peg, temp_peg, to_peg)
        print('Move a disc from peg', from_peg, 'to peg', to_peg)
        move_discs(num - 1, temp_peg, to_peg, from_peg)
```

Recursion versus Looping

- **Reasons not to use recursion:**
  - Less efficient: entails function calling overhead that is not necessary with a loop
  - Usually a solution using a loop is more evident than a recursive solution
- **Some problems are more easily solved with recursion than with a loop**
  - Example: Factorial, where the mathematical definition lends itself to recursion

Sorting and Searching Lists

"There's nothing in your head the sorting hat can't see. So try me on and I will tell you where you ought to be."

-The Sorting Hat,
Harry Potter and the Sorcerer's Stone

Searching

- Given a list of ints find the index of the first occurrence of a target int

index	*0*	*1*	*2*	*3*	*4*	*5*
value	89	0	27	-5	42	11

- Given the above list and a target of 27 the method returns 2
- What if not present?
- What if more than one occurrence?

Using List Methods

```python
nums = [5, 17, 5, 12, -5, 0, 5]
print(nums.index(17))

x = 7
print(nums.index(x))  # Result in runtime error.

if x in nums:
    print(nums.index(x))
else:
    print(x, 'is not in the list.')
```

```
1
7 is not in the list.
```

linear or sequential search

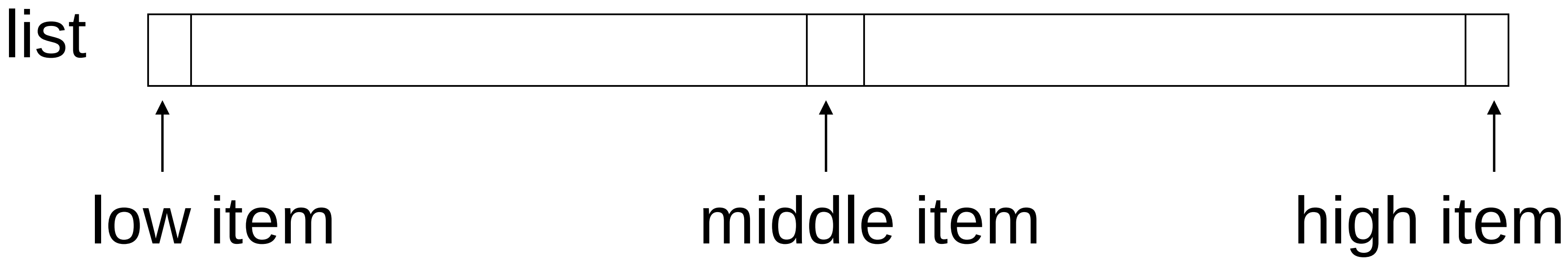
- Implement code for linear search in Python, give a list.

Binary Search

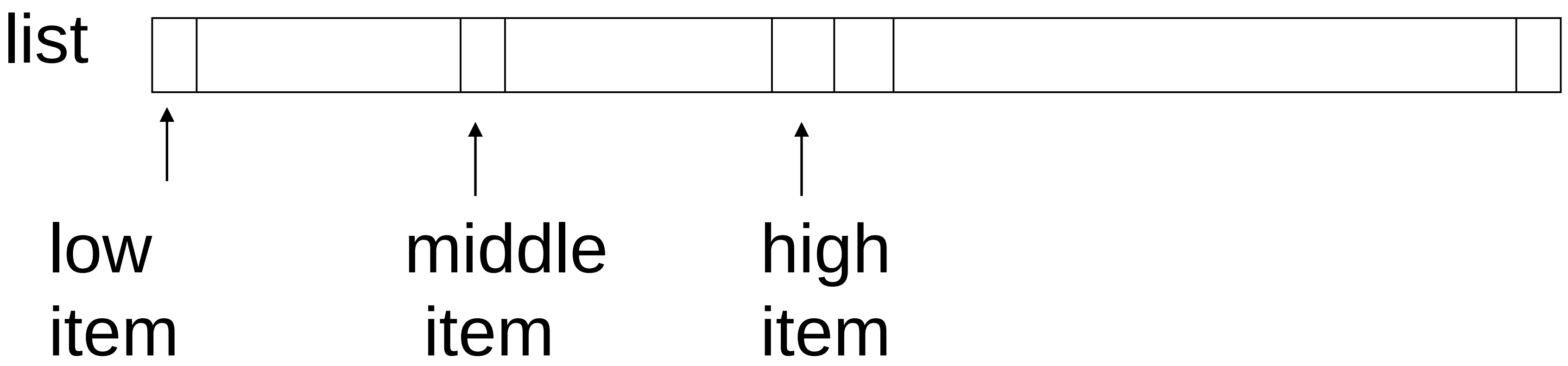
Searching in a Sorted List

- If items are sorted then we can *divide and conquer*
- dividing your work in half with each step
  - generally a good thing
- The Binary Search on List in Ascending order
  - Start at middle of list
  - is that the item?
  - If not is it less than or greater than the item?
  - less than, move to second half of list
  - greater than, move to first half of list
  - repeat until found or sub list size = 0

Binary Search



Is middle item what we are looking for? If not is it more or less than the target item? (Assume lower)



and so forth…

Implement Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53

Trace When Key == 3

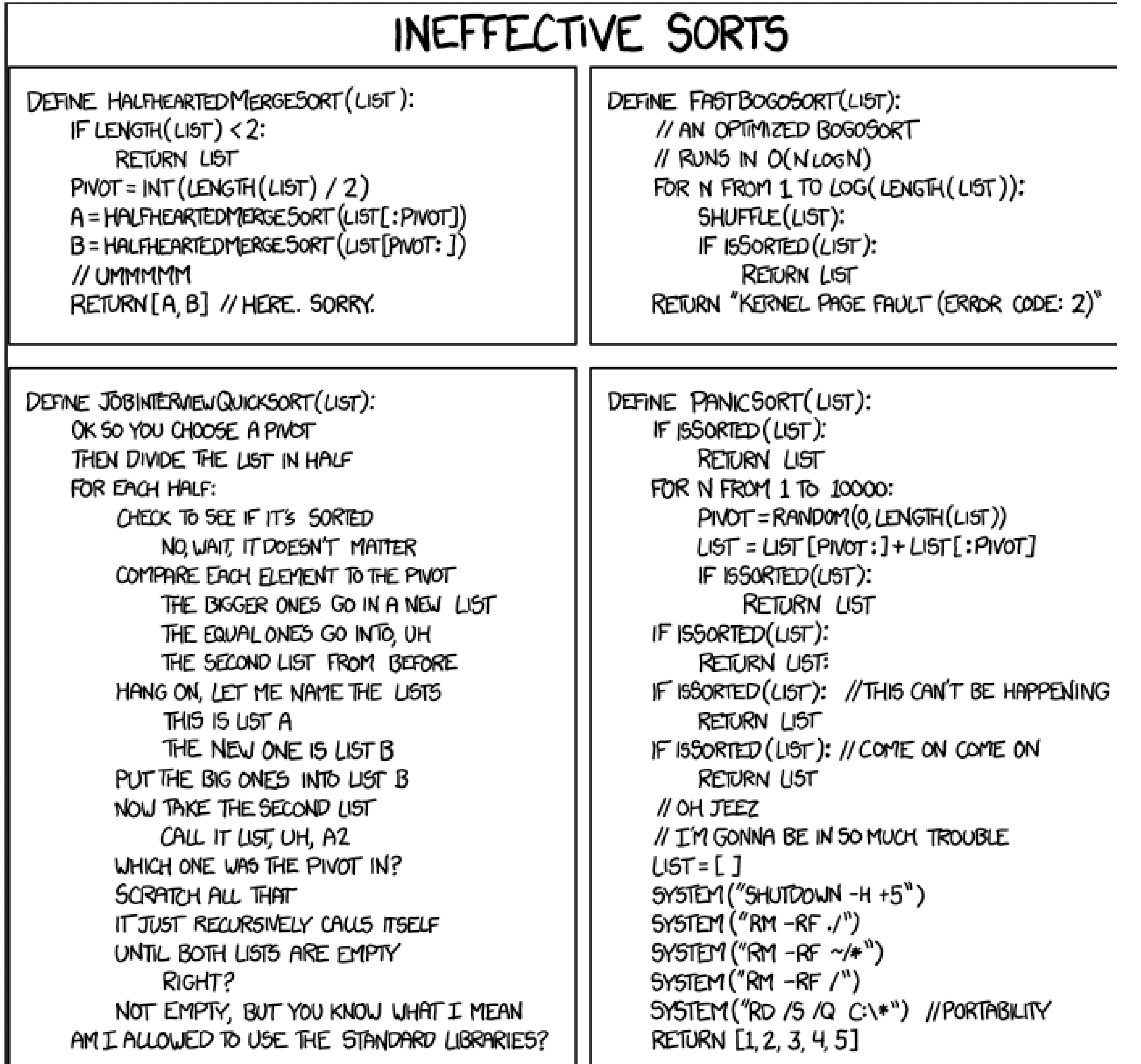
Trace When Key == 30

Variables of Interest?

Sorting

XKCD

http://xkcd.com/1185/



Sorting

- A fundamental application for computers
- Done to make finding data (searching) faster
- Many different algorithms for sorting
- One of the difficulties with sorting is working with a fixed size storage container (array)
  - if resize, that is expensive (slow)
  - Trying to apply a human technique of sorting can be difficult
  - try sorting a pile of papers and clearly write out the algorithm you follow

List sort Method

- List has a sort method
- Works with mixed ints and floats
- Works with Strings
- Does not work with strings and numbers mixed
- Can work with other data types

```
>>> nums = [5, 16, 5, 13]
>>> nums.sort()
>>> nums
[5, 5, 13, 16]
>>> nums.append(17.5)
>>> nums.insert(2, 15.4)
>>> nums
[5, 5, 15.4, 13, 16, 17.5]
>>> nums.sort()
>>> nums
[5, 5, 13, 15.4, 16, 17.5]
>>> nums.append('CS')
>>> nums,sort()
Traceback (most recent call last):
  File "<input>", line 1, in <module
```

Insertion Sort

- Another of the Simple sort
- The first item is sorted
- Compare the second item to the first
  - if smaller swap
- Third item, compare to item next to it
  - need to swap
  - after swap compare again
- And so forth…

Insertion Sort in Practice

44 68 191 119 119 37 83 82 191 45 158 130 76 153 39 25

http://tinyurl.com/d8spm2l

animation of insertion sort algorithm

Timing Question

- Determine how long it takes to sort an array with 100,000 elements in random order using insertion sort. When the number of elements is increased to 200,000 how long will it take to sort the array?

A. About the same
B. 1.5 times as long
C. 2 times as long
D. 4 times as long
E. 8 times as long