



# Chapter 1

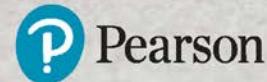
Beginnings

PEARSON

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody

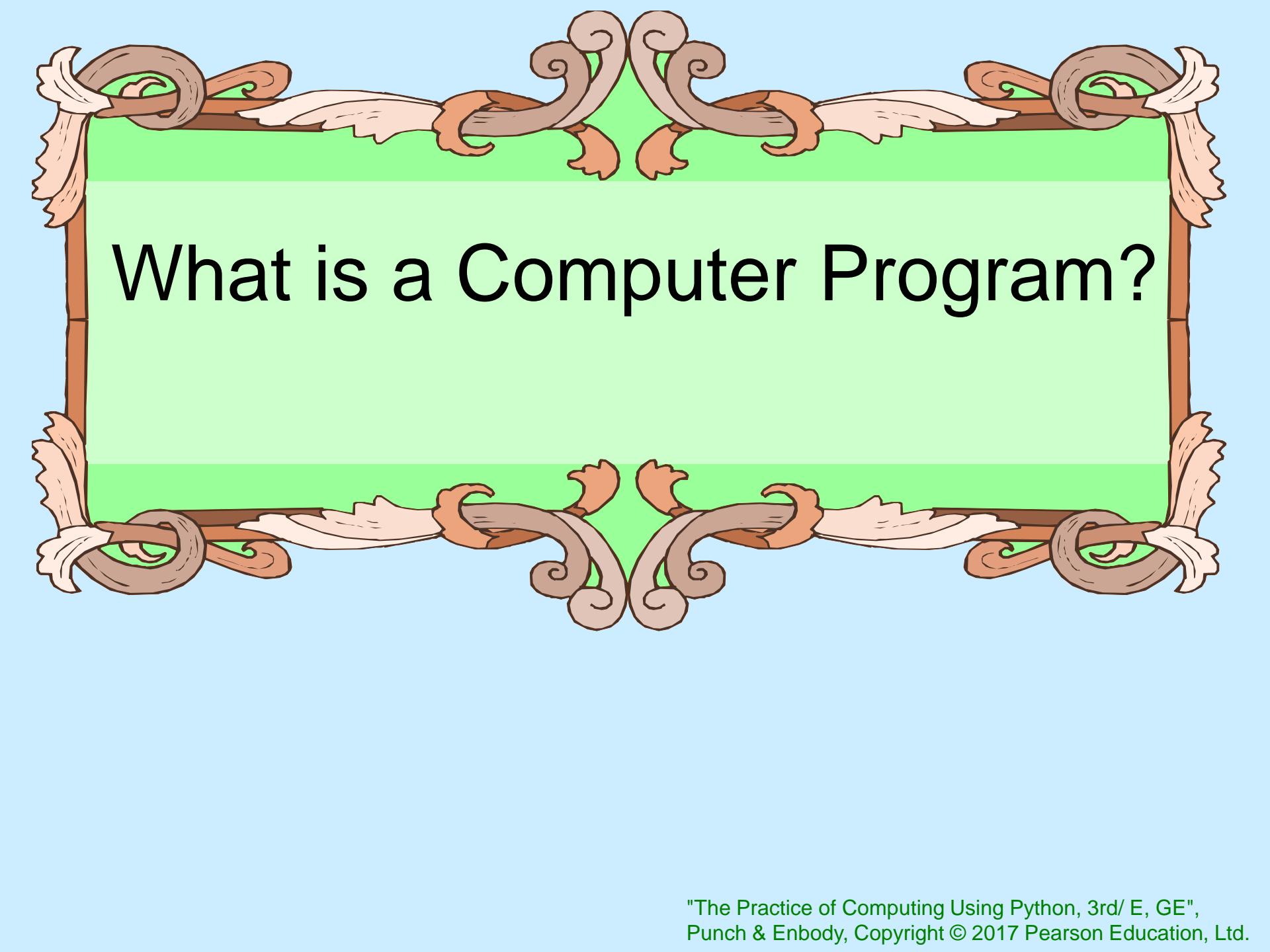


Pearson

ALWAYS LEARNING

# The Three Rules

- Rule 1: Think before you program
- Rule 2: A program is a human-readable essay on problem solving that also happens to execute on a computer
- Rule 3: The best way to improve your programming and problem solving skills is to practice.



# What is a Computer Program?

# Program

- A program is a sequence of instructions.
- To *run* a program is to:
  - create the sequence of instructions according to your design and the language rules
  - turn that program into the binary commands the processor understands
  - give the binary code to the OS, so it can give it to the processor
  - OS tells the processor to run the program
  - when finished (or it dies :-), OS cleans up.

# Interpreted

- Python is an *interpreted* language
- interpreted means that Python looks at each instruction, one at a time, and turns that instruction into something that can be run.
- That means that you can simply open the Python interpreter and enter instructions one-at-a-time.
- You can also *import* a program which causes the instructions in the program to be executed, as if you had typed them in.
- To rerun an imported program you *reload* it.

# Your First Program

## QuickStart 1

```
1 # Calculate the area and circumference of a circle from its radius.  
2 # Step 1: Prompt for a radius.  
3 # Step 2: Apply the area formula.  
4 # Step 3: Print out the results.  
5  
6 import math  
7  
8 radius_str = input("Enter the radius of your circle: ")  
9 radius_int = int(radius_str)  
10  
11 circumference = 2 * math.pi * radius_int  
12 area = math.pi * (radius_int ** 2)  
13  
14 print ("The circumference is:",circumference, \\\n        ", and the area is:",area)
```

# Getting input

The function:

```
input("Give me a value")
```

- prints “Give me a value” on the python screen and waits till the user types something (anything), ending with Enter
- Warning, it returns a string (sequence of characters), no matter what is given, even a number ('1' is not the same as 1, different types)

# import of math

- One thing we did was to import the math module with `import math`
- This brought in python statements to support math (try it in the python window)
- We precede all operations of math with `math.xxx`
- `math.pi`, for example, is pi.  
`math.pow(x, y)` raises  $x$  to the  $y^{\text{th}}$  power.

# Assignment

The `=` sign is the assignment statement

- The value on the right is associated with the variable name on the left
- It does *not* stand for equality!
- More on this later

# Conversion

Convert from string to integer

- Python requires that you must convert a sequence of characters to an integer
- Once converted, we can do math on the integers

# Printing output

```
my_var = 12  
print('My var has a value of: ', my_var)
```

- `print` takes a list of elements in parentheses separated by commas
  - if the element is a string, prints it as is
  - if the element is a variable, prints the value associated with the variable
  - after printing, moves on to a new line of output

# At the core of any language

- Control the flow of the program
- Construct and access data elements
- Operate on data elements
- Construct functions
- Construct classes
- Libraries and built-in classes

# Save as a “module”

- When you save a file, such as our first program, and place a `.py` suffix on it, it becomes a python module
- You run the module from the IDLE menu to see the results of the operation
- A module is just a file of python commands

# Errors

- If there are interpreter errors, that is Python cannot run your code because the code is somehow malformed, you get an error
- You can them import the program again until there are no errors

# Common Error

- Using IDLE, if you save the file without a `.py` suffix, it will stop colorizing and formatting the file.
- Resave with the `.py`, everything is fine

# Syntax

- Lexical components.
- A Python program is:.
  - A module (perhaps more than one)
  - Each module has python statements
  - Each statement has expressions

# Modules

- We've seen modules already, they are essentially files with Python statements.
- There are modules provided by Python to perform common tasks (math, database, web interaction, etc.)
- The wealth of these modules is one of the great features of Python

# Statements

- Statements are commands in Python.
- They perform some action, often called a side effect, but they **do not return any values**

# Expressions

- Expressions perform some operation and **return a value**
- Expressions can act as statements, but statements cannot act as expressions (more on this later).
- Expressions typically do not modify values in the interpreter

# side effects and returns

What is the difference between side effect and return?

- `1 + 2` returns a value (it's an expression). You can “catch”/assign the return value. However, nothing else changed as a result
- `print( "hello" )` doesn't return anything, but something else, the side effect, did happen. Something printed!

# Whitespace

- **white space** are characters that don't print (blanks, tabs, carriage returns etc.)
- For the most part, you can place white space (spaces) anywhere in your program
- use it to make a program more readable

1 +

2

- 4

# continuation

However, python is sensitive to end of line stuff. To make a line continue, use the \

```
print("this is a test", \
      " of continuation")
```

prints

```
this is a test of continuation
```

# also, tabbing is special

- The use of tabs is also something that Python is sensitive to.
- We'll see more of that when we get to control, but be aware that the tab character has meaning to Python

# Python comments

- A comment begins with a # (pound sign)
- This means that from the # to the end of that line, nothing will be interpreted by Python.
- You can write information that will help the reader with the code

# Code as essay, an aside

- What is the primary goal of writing code:
  - to get it to do something
  - an essay on my problem solving thoughts
- Code is something to be read. You provide comments to help readability.

# Knuth, Literate Programming (84)

The practitioner of ... programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.

# Some of the details

- OK, there are some details you have to get used to.
- Let's look at the syntax stuff
- We'll pick more up as we go along

# Python Tokens

**Keywords:**

You cannot  
use (are  
prevented  
from using)  
them in a  
variable name

|          |         |        |        |       |
|----------|---------|--------|--------|-------|
| and      | del     | from   | not    | while |
| as       | elif    | global | or     | with  |
| assert   | else    | if     | pass   | yield |
| break    | except  | import | print  |       |
| class    | exec    | in     | raise  |       |
| continue | finally | is     | return |       |
| def      | for     | lambda | try    |       |

# Python Operators

Reserved operators in Python (expressions)

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| +  | -  | *  | ** | /  | // | %  |
| << | >> | &  |    | ^  | ~  |    |
| <  | >  | <= | >= | == | != | <> |

# Python Punctuators

Python punctuation/delimiters (\$ and ? not allowed).

'        "        #        \  
(        )        [        ]        {        }        @  
,        :        .        `        =        ;  
+=        -=        \*=        /=        //=        %=  
&=        |=        ^=        >>=        <<=        \*\*=

# Literals

Literal is a programming notation for a ***fixed value***.

- For example, 123 is a fixed value, an integer
  - it would be weird if the symbol 123's value could change to be 3.14!

# Python name conventions

- must begin with a letter or underscore \_  
Ab\_123 is OK, but 123\_ABC is not.
- may contain letters, digits, and underscores  
this\_is\_an\_identifier\_123
- may be of any length
- upper and lower case letters are different  
Length\_Of\_Rope is not length\_of\_rope
- names starting with \_ (underline) have special meaning. Be careful!

# Naming conventions

- Fully described by PEP8 or Google Style Guide for Python
  - <http://google-styleguide.googlecode.com/svn/trunk/pyguide.html>
- the standard way for most things named in python is **lower with under**, lower case with separate words joined by an underline:
  - this\_is\_a\_var
  - my\_list
  - square\_root\_function

# Rule 4

*A foolish consistency is the hobgoblin of little minds*

Quote from Ralph Waldo Emerson

We name things using conventions, but admit that, under the right circumstances, we do what is necessary to help readability.

# Variable

- A variable is a name we designate to represent an object (number, data structure, function, etc.) in our program
- We use names to make our program more readable, so that the object is easily understood in the program

# Variable Objects

- Python maintains a list of pairs for every variable:
  - variable's name
  - variable's value
- A variable is created when a value is assigned the first time. It associates a name and a value
- subsequent assignments update the associated value.
- we say name references value

my\_int = 7 →

| Name   | Value |
|--------|-------|
| my_int | 7     |

# Namespace

- A **namespace** is the table that contains the association of a name with a value
- We will see more about namespaces as we get further into Python, but it is an essential part of the language.



# When = doesn't mean equal

- It is most confusing at first to see the following kind of expression:

```
my_int = my_int + 7
```

- You don't have to be a math genius to figure out something is wrong there.
- What's wrong is that = doesn't mean equal

# = is assignment

- In many computer languages, = means assignment.

```
my_int = my_int + 7
```

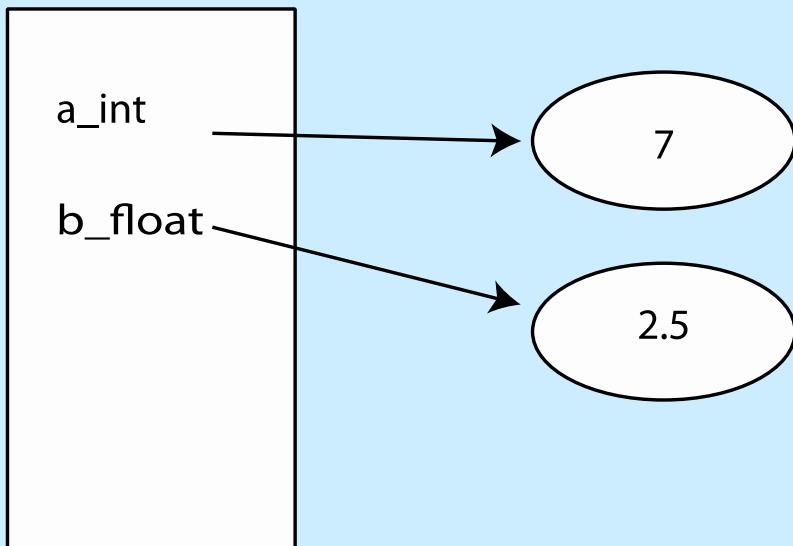
```
lhs = rhs
```

- What assignment means is:
  - evaluate the rhs of the =
  - take the resulting value and associate it with the name on the lhs

# More Assignment

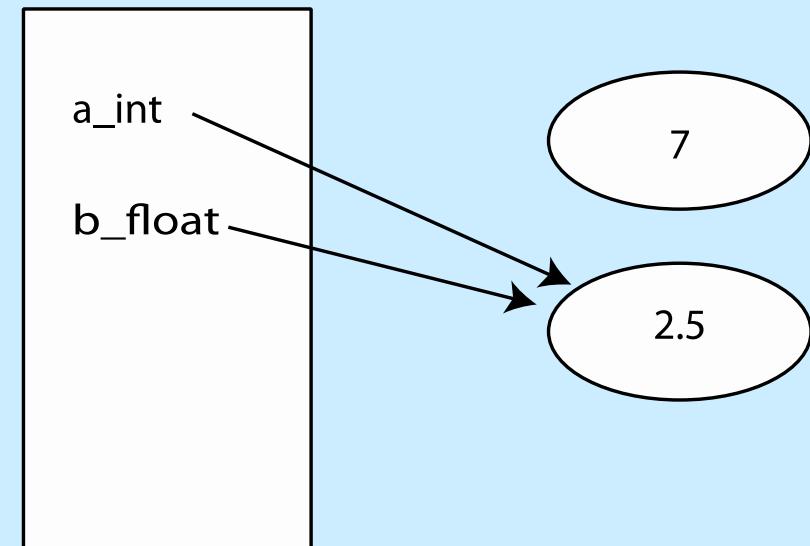
- Example: `my_var = 2 + 3 * 5`
  - evaluate expression ( $2+3*5$ ): 17
  - change the value of `my_var` to reference 17
- Example (`my_int` has value 2):  
`my_int = my_int + 3`
  - evaluate expression (`my_int + 3`): 5
  - change the value of `my_int` to reference 5

Name list



Values

Name list



`a_int = 7`  
`b_float = 2.5`

`a_int = b_float`

# variables and types

- Python does not require you to pre-define what type can be associated with a variable
- What type a variable holds can change
- Nonetheless, knowing the type can be important for using the correct operation on a variable. Thus proper naming is important!

# What can go on the lhs

- There are limits therefore as to what can go on the lhs of an assignment statement.
- The lhs must indicate a name with which a value can be associated
- must follow the naming rules

myInt = 5                          Yes

myInt + 5 = 7                      No

# Python “types”

- integers: **5**
- floats: **1.2**
- booleans: **True**
- strings: "anything" or 'something'
- lists: **[,] ['a',1,1.3]**
- others we will see

# What is a type

- a type in Python essentially defines two things:
  - the internal structure of the type (what it contains)
  - the kinds of operations you can perform
- `'abc'.capitalize()` is a method you can call on strings, but not integers
- some types have multiple elements (collections), we'll see those later

# Fundamental Types

- Integers
  - `1, -27` ( to  $+/- 2^{32} - 1$ )
  - `123L` L suffix means any length, but potentially very slow. Python will convert if an integer gets too long automatically
- Floating Point (Real)
  - `3.14, 10., .001, 3.14e-10, 0e0`
- Booleans (True or False values)
  - `True, False` note the capital

# Converting types

- A character '**1**' is not an integer **1**. We'll see more on this later, but take my word for it.
- You need to convert the value returned by the `input` command (characters) into an integer
- `int( "123" )` yields the integer 123

# Type conversion

- `int(some_var)` returns an integer
- `float(some_var)` returns a float
- `str(some_var)` returns a string
- should check out what works:
  - `int(2.1) → 2`, `int('2') → 2`, but `int('2.1')` fails
  - `float(2) → 2.0`, `float('2.0') → 2.0`, `float('2') → 2.0`, `float(2.0) → 2.0`
  - `str(2) → '2'`, `str(2.0) → '2.0'`, `str('a') → 'a'`

# Operators

- Integer
  - addition and subtraction:  $+$ ,  $-$
  - multiplication:  $*$
  - division
    - quotient:  $/$
    - integer quotient:  $//$
    - remainder:  $\%$
- Floating point
  - add, subtract, multiply, divide:  $+$ ,  $-$ ,  $*$ ,  $/$

# Binary operators

The operators addition(+), subtraction(-) and multiplication(\*) work normally:

- `a_int = 4`
- `b_int = 2`
- `a_int + b_int` → yields 6
- `a_int - b_int` → yields 2
- `a_int * b_int` → yields 8

# Two types of division

The standard division operator (/) yields a floating point result no matter the type of its operands:

- $2 / 3 \rightarrow$  yields  $0.6666666666666666$
- $4.0 / 2 \rightarrow$  yields  $2.0$

Integer division (//) yields only the integer part of the divide (its type depends on its operands):

- $2 // 3 \rightarrow$  0
- $4.0 // 2 \rightarrow$  2.0

# Modulus Operator

The modulus operator (%) give the integer remainder of division:

- $5 \% 3 \rightarrow 2$
- $7.0 \% 3 \rightarrow 1.0$

Again, the type of the result depends on the type of the operands.

# Mixed Types

What is the difference between 42 and  
42.0 ?

- their types: the first is an integer, the second is a float

What happens when you mix types:

- done so no information is lost

42 \* 3 → 126

42.0 \* 3 → 126.0

# Order of operations and parentheses

| Operator                 | Description                                   |
|--------------------------|---|
| <code>()</code>          | Parenthesis (grouping)                        |
| <code>**</code>          | Exponentiation                                |
| <code>+x, -x</code>      | Positive, Negative                            |
| <code>*, /, %, //</code> | Multiplication, Division, Remainder, Quotient |
| <code>+, -</code>        | Addition, Subtraction                         |

- Precedence of `*, /` over `+, -` is the same, but there precedents for other operators as well
- Remember, parentheses always takes precedence

# Augmented assignment

Shortcuts can be distracting, but one that is often used is augmented assignment:

- combines an operation and reassignment to the same variable
- useful for increment/decrement

| Shortcut                 | Equivalence                      |
|--------------------------|----------------------------------|
| <code>my_int += 2</code> | <code>my_int = my_int + 2</code> |
| <code>my_int -= 2</code> | <code>my_int = my_int - 2</code> |
| <code>my_int /= 2</code> | <code>my_int = my_int / 2</code> |
| <code>my_int *= 2</code> | <code>my_int = my_int * 2</code> |

# Modules

Modules are files that can be imported into your Python program.

- use other, well proven code with yours

Example is the math module

- we import a module to use its contents
- we use the name of the module as part of the content we imported

# math module

```
import math  
print(math.pi)      # constant in math module  
print(math.sin(1.0))# a function in math  
help(math.pow)      # help info on pow
```

# Developing an Algorithm

# Develop an Algorithm

How do we solve the following?

- If one inch of rain falls on an acre of land, how many gallons of water have accumulated on that acre?

# Algorithm

*A method – a sequence of steps – that describes how to solve a problem or class of problems*

# Rule 5

Test your code, often and thoroughly!

One thing we learn in writing our code is that we must test it, especially against a number of conditions, to assure ourselves that it works

- it turns out that testing is very hard and "correct" is a difficult thing to establish!

## Code Listing 1.2-1.3

```
# Calculate rainfall in gallons for some number of inches on 1 acre.  
inches_str = input("How many inches of rain have fallen: ")  
inches_int = int(inches_str)  
volume = (inches_int/12)*43560  
gallons = volume * 7.48051945  
print(inches_int, " in. rain on 1 acre is", gallons, "gallons")
```

```
# Calculate rainfall in gallons for some number of inches on 1 acre.  
inches_str = input("How many inches of rain have fallen: ")  
inches_float = float(inches_str)  
volume = (inches_float/12)*43560  
gallons = volume * 7.48051945  
print(inches_float, " in. rain on 1 acre is", gallons, "gallons")
```

# The Rules

1. Think before you program
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice.
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly!

GLOBAL  
EDITION



# Chapter 2

## Control

PEARSON

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



Pearson

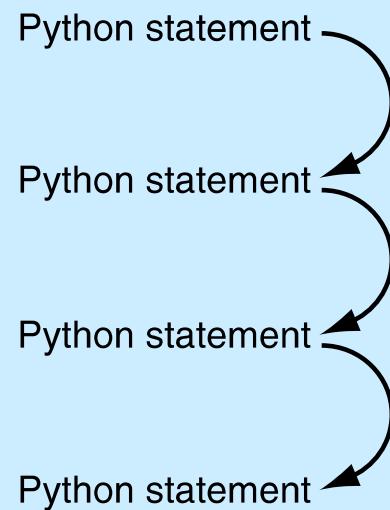
ALWAYS LEARNING

# Control, Quick Overview

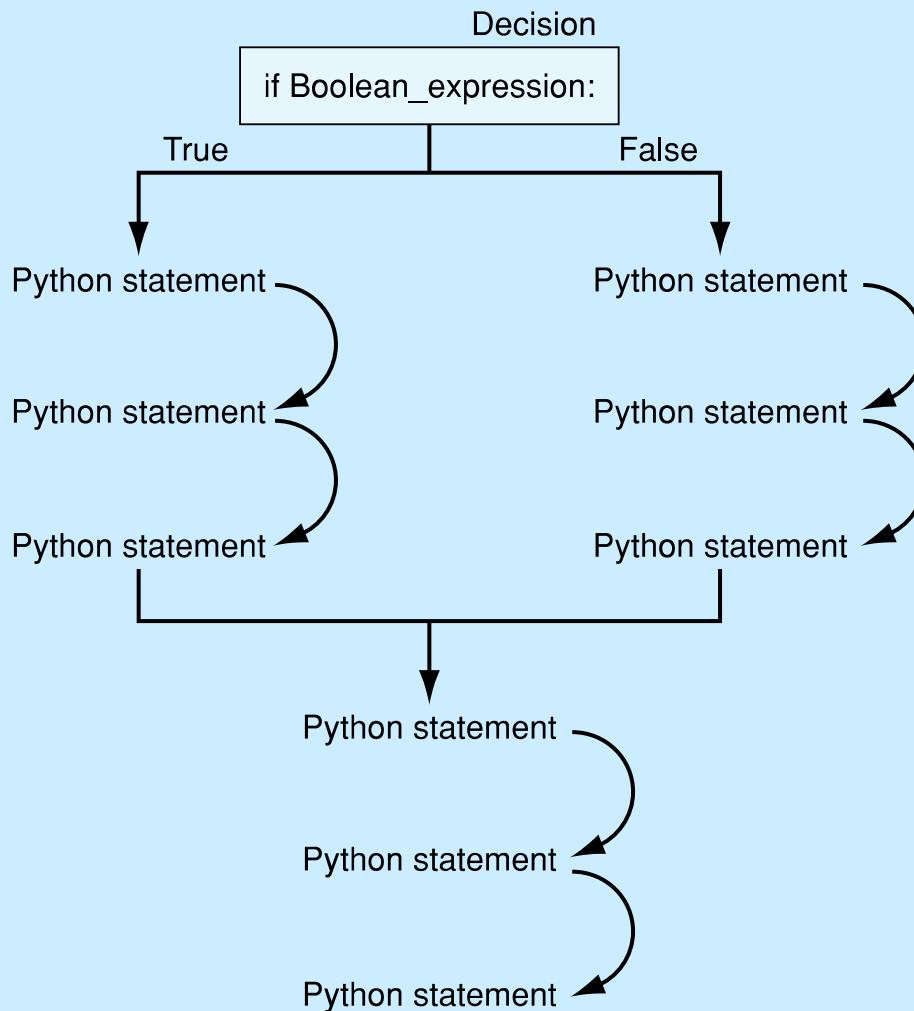
# Selection

# Selection

- Selection is how programs make choices, and it is the process of making choices that provides a lot of the power of computing



**FIGURE 2.1** Sequential program flow.



**FIGURE 2.2** Decision making flow of control.

|    |                          |
|----|--------------------------|
| <  | less than                |
| >  | greater than             |
| <= | less than or equal to    |
| >= | greater than or equal to |
| == | equal to                 |
| != | not equal to             |

**TABLE 2.1** Boolean Operators.

Note that **==** is equality,  
**=** is assignment

# Python if statement

```
if boolean expression :  
    suite
```

- evaluate the boolean (`True` or `False`)
- if `True`, execute all statements in the suite

# Warning about indentation

- Elements of the suite must all be indented the same number of spaces/tabs
- Python only recognizes suites when they are indented the same distance (***standard is 4 spaces***)
- You must be careful to get the indentation right to get suites right.

# Python Selection, Round 2

```
if boolean expression:
```

```
    suite1
```

```
else:
```

```
    suite2
```

The process is:

- evaluate the boolean
- if `True`, run suite1
- if `False`, run suite2

```
>>> first_int = 10
>>> second_int = 20
>>> if first_int > second_int:
    print("The first int is bigger!")
else:
    print("The second int is bigger!")
```

The second int **is** bigger!

>>>

# Safe Lead in Basketball

- Algorithm due to Bill James  
([www.slate.com](http://www.slate.com))
- under what conditions can you safely determine that a lead in a basketball game is insurmountable?

# The algorithm

- Take the number of points one team is ahead
- Subtract three
- Add  $\frac{1}{2}$  point if team that is ahead has the ball, subtract  $\frac{1}{2}$  point otherwise
- Square the result
- If the result is greater than the number of seconds left, the lead is safe

## Code Listing 2.3

# first cut

```
# 3. Add a half-point if the team that is ahead has the ball,  
#     and subtract a half-point if the other team has the ball.
```

```
has_ball_str = input("Does the lead team have the ball (Yes or No): ")  
  
if has_ball_str == "Yes":  
    lead_calculation_float = lead_calculation_float + 0.5  
else:  
    lead_calculation_float = lead_calculation_float - 0.5
```

Problem, what if the lead is less than 0?

## Code Listing 2.4

# second cut

```
# 3. Add a half-point if the team that is ahead has the ball,  
#     and subtract a half-point if the other team has the ball.
```

```
has_ball_str = input("Does the lead team have the ball (Yes or No) : ")  
  
if has_ball_str == 'Yes':  
    lead_calculation_float = lead_calculation_float + 0.5  
else:  
    lead_calculation_float = lead_calculation_float - 0.5  
  
# (Numbers less than zero become zero)  
if lead_calculation_float < 0:  
    lead_calculation_float = 0
```

catch the lead less than 0



## Code Listing 2.7

```

# 1. Take the number of points one team is ahead.
points_str = input("Enter the lead in points: ")
points_remaining_int = int(points_str)

# 2. Subtract three.
lead_calculation_float= float(points_remaining_int - 3)

# 3. Add a half-point if the team that is ahead has the ball,
# and subtract a half-point if the other team has the ball.
has_ball_str = input("Does the lead team have the ball (Yes or No): ")

if has_ball_str == 'Yes':
    lead_calculation_float= lead_calculation_float + 0.5
else:
    lead_calculation_float= lead_calculation_float - 0.5

# (Numbers less than zero become zero)
if lead_calculation_float< 0:
    lead_calculation_float= 0

# 4. Square that.
lead_calculation_float= lead_calculation_float** 2

# 5. If the result is greater than the number of seconds left in the game,
# the lead is safe.
seconds_remaining_int = int(input("Enter the number of seconds remaining: "))

if lead_calculation_float> seconds_remaining_int:
    print("Lead is safe.")
else:
    print("Lead is not safe.")

```

# Repetition, quick overview

# Repeating statements

- Besides selecting which statements to execute, a fundamental need in a program is repetition
  - repeat a set of statements under some conditions
- With both selection and repetition, we have the two most necessary programming statements

# While and For statements

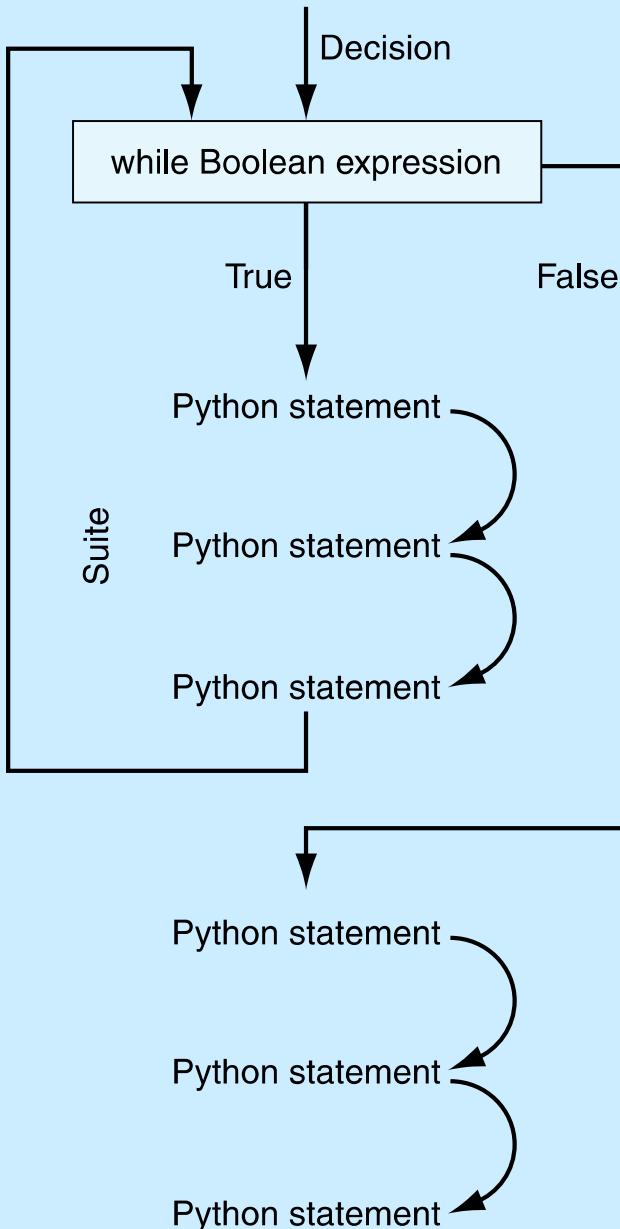
- The `while` statement is the more general repetition construct. It repeats a set of statements while some condition is True.
- The `for` statement is useful for iteration, moving through all the elements of data structure, one at a time.

# while loop

- Top-tested loop (pretest)
  - test the boolean before running
  - test the boolean before each iteration of the loop

```
while boolean expression:  
    suite
```

## FIGURE 2.4 while loop.



# repeat while the boolean is true

- while loop will repeat the statements in the suite while the boolean is True (or its Python equivalent)
- If the Boolean expression never changes during the course of the loop, the loop will continue forever.

## Code Listing 2.8

```
1 # simple while
2
3 x_int = 0      # initialize loop-control variable
4
5 # test loop-control variable at beginning of loop
6 while x_int < 10:
7     print(x_int, end=' ')  # print the value of x_int each time through the
                           # while loop
8     x_int = x_int + 1      # change loop-control variable
9
10 print()
11 print("Final value of x_int: ", x_int) # bigger than value printed in loop!
```

# General approach to a while

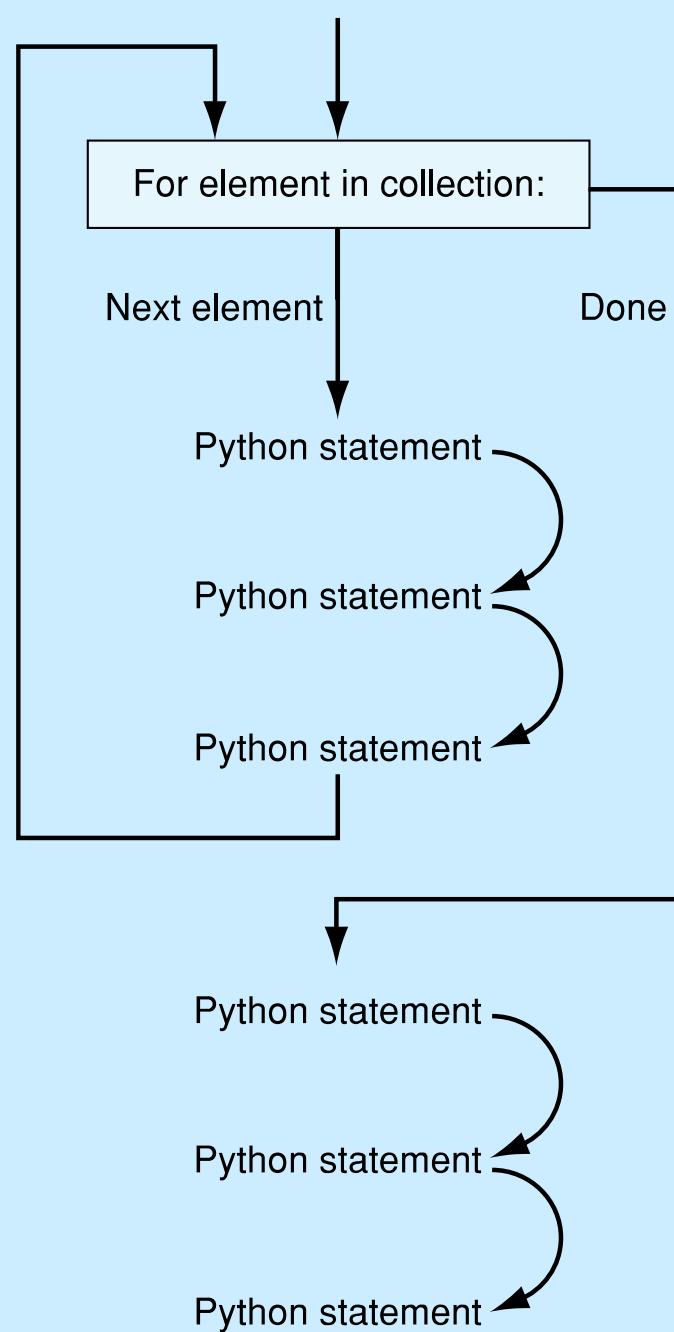
- outside the loop, initialize the boolean
- somewhere inside the loop you perform some operation which changes the state of the program, eventually leading to a False boolean and exiting the loop
- Have to have both!

# for and iteration

- One of Python's strength's is it's rich set of built-in data structures
- The for statement iterates through each element of a collection (list, etc.)

```
for element in collection:  
    suite
```

**FIGURE 2.5** Operation of a *for* loop.



# Perfect Number Example

# a perfect number

- numbers and their factors were mysterious to the Greeks and early mathematicians
- They were curious about the properties of numbers as they held some significance
- A perfect number is a number whose sum of factors (excluding the number) equals the number
- First perfect number is: 6 (1+2+3)

# abundant, deficient

- abundant numbers summed to more than the number.
  - 12:  $1+2+3+4+6 = 16$
- deficient numbers summed to less than the number.
  - 13: 1

# design

- prompt for a number
- for the number, collect all the factors
- once collected, sum up the factors
- compare the sum and the number and respond accordingly

## Code Listing 2.10,2.11

### Check Perfection

### Sum Divisors

## Code Listing 2.10

```
if number_int == sum_of_divisors_int:  
    print(number_int, "is perfect")  
else:  
    print(number_int, "is not perfect")
```

## Code Listing 2.11

```
divisor = 1  
sum_of_divisors = 0  
while divisor < number:  
    if number % divisor == 0:          # divisor evenly divides theNum  
        sum_of_divisors = sum_of_divisors + divisor  
    divisor = divisor + 1
```

# Improving the Perfect Number Program

Work with a range of numbers

For each number in the range of numbers:

- collect all the factors
- once collected, sum up the factors
- compare the sum and the number and respond accordingly

Print a summary

## Code Listing 2.13

### Examine a range of numbers

```
top_num_str = input("What is the upper number for the range:")
top_num = int(top_num_str)
number=2
while number <= top_num:
    # sum the divisors of number
    # classify the number based on its divisor sum
    number += 1
```

## Code Listing 2.15

### Classify range of numbers

## Code Listing 2.15

```
# classify a range of numbers with respect to perfect, abundant or deficient
# unless otherwise stated, variables are assumed to be of type int. Rule 4

top_num_str = input("What is the upper number for the range:")
top_num = int(top_num_str)
number=2

while number <= top_num:
    # sum up the divisors
    divisor = 1
    sum_of_divisors = 0
    while divisor < number:
        if number % divisor == 0:
            sum_of_divisors = sum_of_divisors + divisor
        divisor = divisor + 1

    # classify the number based on its divisor sum
    if number == sum_of_divisors:
        print(number,"is perfect")
    if number < sum_of_divisors:
        print(number,"is abundant")
    if number > sum_of_divisors:
        print(number,"is deficient")

    number += 1
```

# Control in Depth

# Booleans

# Boolean Expressions

- George Boole's (mid-1800's) mathematics of logical expressions
- Boolean expressions (conditions) have a value of True or False
- Conditions are the basis of choices in a computer, and, hence, are the basis of the appearance of intelligence in them.

# What is True, and what is False

- true: any nonzero number or nonempty object. `1, 100, "hello", [a,b]`
- false: a zero number or empty object. `0, "", []`
- Special values called `True` and `False`, which are just subs for 1 and 0. However, they print nicely (`True` or `False`)
- Also a special value, `None`, less than everything and equal to nothing

# Boolean expression

- Every boolean expression has the form:
  - expression booleanOperator expression
- The result of evaluating something like the above is also just true or false.
- However, remember what constitutes true or false in Python!

# Relational Operators

- $3 > 2 \rightarrow \text{True}$
- Relational Operators have low preference
  - $5 + 3 < 3 - 2$
  - $8 < 1 \rightarrow \text{False}$
- $'1' < 2 \rightarrow \text{Error}$ 
  - can only compare like types
- $\text{int}('1') < 2 \rightarrow \text{True}$ 
  - like types, regular compare

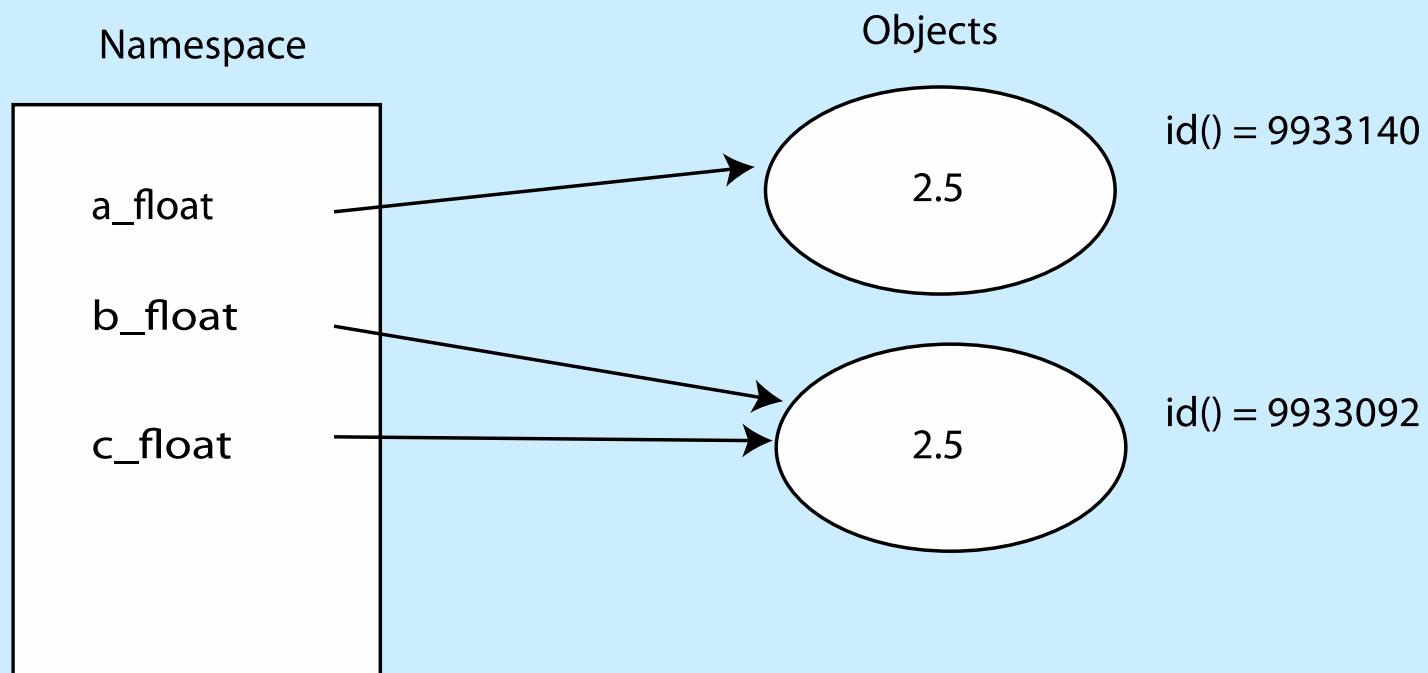
# What does Equality mean?

Two senses of equality

- two variables refer to different objects,  
each object representing the same value
- two variables refer to the same object. The  
`id()` function used for this.

## FIGURE 2.6 What is equality?

```
a_float = 2.5  
b_float = 2.5  
c_float = b_float
```



# equal vs. same

- `==` compares values of two variable's objects, do they represent the same value
- `is` operator determines if two variables are associated with the same value

From the figure:

`a_float == b_float` → True

`a_float is b_float` → False

`b_float is c_float` → True

# Chained comparisons

- In Python, chained comparisons work just like you would expect in a mathematical expression:
- Given myInt has the value 5
  - `0 <= myInt <= 5` → True
  - `0 < myInt <= 5 > 1` → False

# Pitfall

floating point arithmetic is approximate!

```
>>> u = 11111113
>>> v = -11111111
>>> w = 7.51111111
>>> (u + v) + w
9.51111111
>>> u + (v + w)
9.51111110448837
>>> (u + v) + w == u + (v + w)
False
```

# compare using "close enough"

Establish a level of "close enough" for equality

```
>>> u = 11111113
>>> v = -11111111
>>> w = 7.51111111
>>> x = (u + v) + w
>>> y = u + (v + w)
>>> x == y
False
>>> abs(x - y) < 0.0000001 # abs is absolute value
True
```

# Compound Expressions

Python allows bracketing of a value between two Booleans, as in math

```
a_int = 5
```

```
0 <= a_int <= 10 → True
```

- `a_int >= 0 and a_int <= 10`
- `and, or, not` are the three Boolean operators in Python

# Truth Tables

| p     | q     | not p | p and q | p or q |
|-------|-------|-------|---------|--------|
| True  | True  |       |         |        |
| True  | False |       |         |        |
| False | True  |       |         |        |
| False | False |       |         |        |

# Truth Tables

| p     | q     | not p | p and q | p or q |
|-------|-------|-------|---------|--------|
| True  | True  | False |         |        |
| True  | False | False |         |        |
| False | True  | True  |         |        |
| False | False | True  |         |        |

# Truth Tables

| p     | q     | not p | p and q | p or q |
|-------|-------|-------|---------|--------|
| True  | True  |       | True    |        |
| True  | False |       | False   |        |
| False | True  |       | False   |        |
| False | False |       | False   |        |

# Truth Tables

| p     | q     | not p | p and q | p or q |
|-------|-------|-------|---------|--------|
| True  | True  |       |         | True   |
| True  | False |       |         | True   |
| False | True  |       |         | True   |
| False | False |       |         | False  |

# Truth Tables

| p     | q     | not p | p and q | p or q |
|-------|-------|-------|---------|--------|
| True  | True  | False | True    | True   |
| True  | False | False | False   | True   |
| False | True  | True  | False   | True   |
| False | False | True  | False   | False  |

# Compound Evaluation

- Logically  $0 < \text{a\_int} < 3$  is actually  
 $(0 < \text{a\_int}) \text{ and } (\text{a\_int} < 3)$
- Evaluate using  $\text{a\_int}$  with a value of 5:  
 $(0 < \text{a\_int}) \text{ and } (\text{a\_int} < 3)$
- Parenthesis first: (True) and (False)
- Final value: False
- (Note: parenthesis are not necessary in this case.)

# Precedence & Associativity

Relational operators have precedence and associativity just like numerical operators.

| <i>Operator</i>      | <i>Description</i>                  |
|----------------------|-------------------------------------|
| ()                   | Parenthesis (grouping)              |
| **                   | Exponentiation                      |
| +x, -x               | Positive, Negative                  |
| *, /, %              | Multiplication, Division, Remainder |
| +, -                 | Addition, Subtraction               |
| <, <=, >, >=, !=, == | Comparisons                         |
| not x                | Boolean NOT                         |
| and                  | Boolean AND                         |
| or                   | Boolean OR                          |

**TABLE 2.2** Precedence of Relational and Arithmetic Operators: Highest to Lowest

# Boolean operators vs. relationals

- Relational operations always return True or False
- Boolean operators (`and`, `or`) are different in that:
  - They can return values (that represent True or False)
  - They have ***short circuiting***

# Remember!

- 0, '', [] or other “empty” objects are equivalent to False
- anything else is equivalent to True

# Ego Search on Google

- Google search uses Booleans
- by default, all terms are and'ed together
- you can specify or (using OR)
- you can specify not (using -)
- Example is:

'Punch' and ('Bill' or 'William') and not  
'gates'

Google Advanced Search

http://www.google.com/advanced\_search?hl=en

231 251 CNN NYT WSJ Markets accuweather Slashdot Wikipedia MacRumors MacPorts drudge bones News (824) Popular

Google Advanced Search

# Google™ Advanced Search

Advanced Search Tips | About Google

Use the form below and your advanced search will appear here

**Find web pages that have...**

all these words:

this exact wording or phrase:  tip

one or more of these words:  OR  OR  tip

**But don't show pages that have...**

any of these unwanted words:  tip

**Need more tools?**

Results per page:

Language:

File type:

Search within a site or domain:   
(e.g. youtube.com, .edu)

Date, usage rights, numeric range, and more

Topic-specific search engines from Google:

The screenshot shows the Google Advanced Search interface. At the top, there's a navigation bar with links like AD, IP, and various news sources. Below that is the main search form. It starts with a placeholder text 'Use the form below and your advanced search will appear here'. Then it has sections for finding pages with specific words ('Find web pages that have...'), excluding words ('But don't show pages that have...'), and setting search parameters like results per page and language. There are also dropdowns for file type and a field for searching within a specific site or domain. At the bottom left, there's a checkbox for adding date and usage rights filters, and at the bottom right, a large 'Advanced Search' button.

**FIGURE 2.7** The Google advanced search page.

# More on Assignments

# Remember Assignments?

- Format: `lhs = rhs`
- Behavior:
  - expression in the rhs is evaluated producing a value
  - the value produced is placed in the location indicated on the lhs

# Can do multiple assignments

```
a_int, b_int = 2, 3
```

first on right assigned to first on left, second  
on right assigned to second on left

```
print(a_int, b_int) # prints 2 3
```

```
a_int,b_int = 1,2,3 ➔ Error
```

counts on lhs and rhs must match

# traditional swap

- Initial values: `a_int= 2, b_int = 3`
- Behavior: swap values of x and Y
  - Note: `a_int = b_int`  
`a_int = b_int` doesn't work (why?)
  - introduce extra variable `temp`
    - `temp = a_int` # save `a_int` value in `temp`
    - `a_int = b_int` # assign `a_int` value to `b_int`
    - `b_int = temp` # assign `temp` value to `b_int`

# Swap using multiple assignment

```
a_int, b_int = 2, 3  
print(a_int, b_int) # prints 2 3
```

```
a_int, b_int = b_int, a_int  
print(a_int, b_int) # prints 3 2
```

remember, evaluate all the values on the rhs first, then assign to variables on the lhs

# Chaining for assignment

Unlike other operations which chain left to right, assignment chains right to left

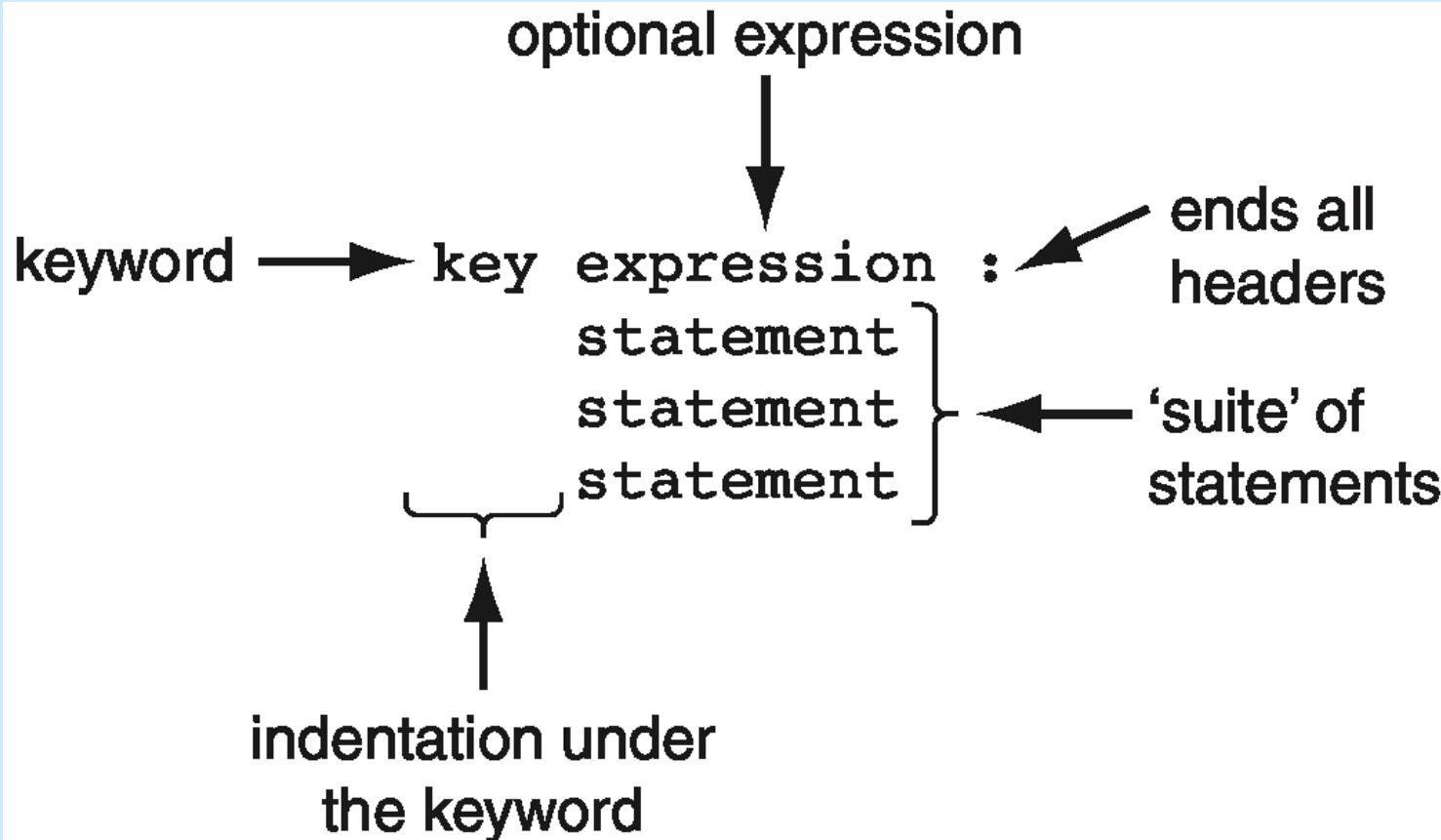
```
a_int = b_int = 5  
print(a_int, b_int) # prints 5 5
```

# More Control: Selection

# Compound Statements

- Compound statements involve a set of statements being used as a group
- Most compound statements have:
  - a header, ending with a : (colon)
  - a suite of statements to be executed
- `if`, `for`, `while` are examples of compound statements

# General format, suites



# Have seen 2 forms of selection

```
if boolean expression:  
    suite
```

```
if boolean expression:  
    suite  
else:  
    suite
```

# Python Selection, Round 3

```
if boolean expression1:  
    suite1  
elif boolean expression2:  
    suite2  
(as many elif's as you want)  
else:  
    suite_last
```

# if, elif, else, the process

- evaluate Boolean expressions until:
  - the Boolean expression returns True
  - none of the Boolean expressions return True
- if a boolean returns True, run the corresponding suite. Skip the rest of the *if*
- if no boolean returns True, run the *else* suite, the default suite

## Code Listing 2.16

### using elif

```
percent_float = float(input("What is your percentage? "))

if 90 <= percent_float < 100:
    print("you received an A")
elif 80 <= percent_float < 90:
    print("you received a B")
elif 70 <= percent_float < 80:
    print("you received a C")
elif 60 <= percent_float < 70:
    print("you received a D")
else:
    print("oops, not good")
```

What happens if `elif` are replaced by `if`?

## Code Listing 2.19

### Updated Perfect Number classification

```
# classify the number based on its divisor sum
if number == sum_of_divisors:
    print(number, "is perfect")
elif number < sum_of_divisors:
    print(number, "is abundant")
else:
    print(number, "is deficient")
number += 1
```

# More Control: Repetition

# Developing a while loop

Working with the *loop control variable*:

- Initialize the variable, typically outside of the loop and before the loop begins.
- The condition statement of the while loop involves a Boolean using the variable.
- Modify the value of the control variable during the course of the loop

# Issues:

Loop never starts:

- the control variable is not initialized as you thought (or perhaps you don't always want it to start)

Loop never ends:

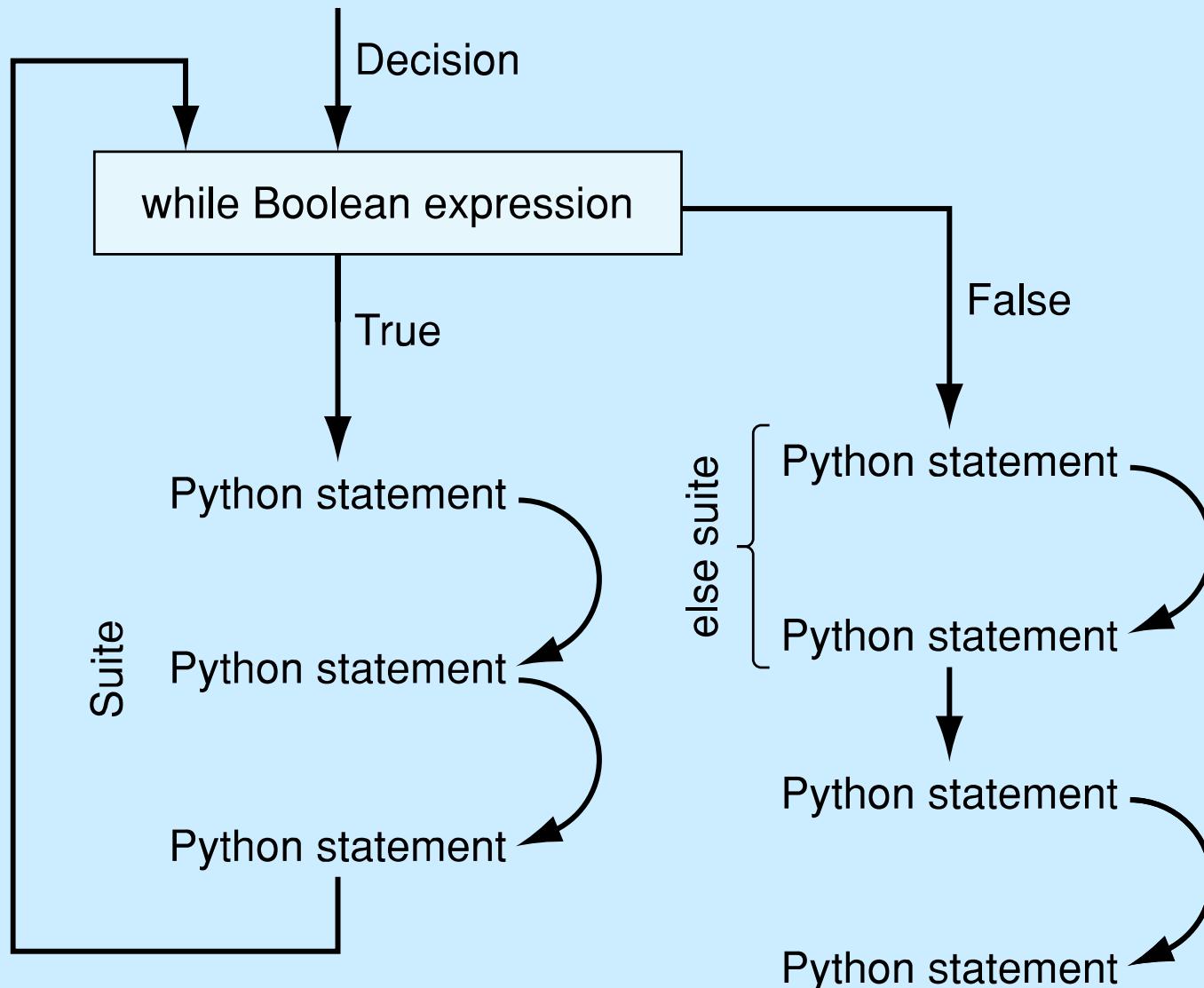
- the control variable is not modified during the loop (or not modified in a way to make the Boolean come out `False`)

# while loop, round two

- while loop, oddly, can have an associated else suite
- else suite is executed when the loop finishes under normal conditions
  - basically the last thing the loop does as it exits

# while with else

```
while booleanExpression:  
    suite  
    suite  
else:  
    suite  
    suite  
rest of the program
```



**FIGURE 2.9** `while-else`.

# Break statement

- A `break` statement in a loop, if executed, exits the loop
- It exists immediately, skipping whatever remains of the loop as well as the `else` statement (if it exists) of the loop

## Code Listing 2.20

### Loop, Hi Lo Game



```
14 # get an initial guess
15 guess_str = input("Guess a number: ")
16 guess = int(guess_str) # convert string to number
17
18 # while guess is range, keep asking
19 while 0 <= guess <= 100:
20     if guess > number:
21         print("Guessed Too High.")
22     elif guess < number:
23         print("Guessed Too Low.")
24     else:                      # correct guess, exit with break
25         print("You guessed it. The number was:",number)
26         break
27     # keep going, get the next guess
28     guess_str = input("Guess a number: ")
29     guess = int(guess_str)
30 else:
31     print("You quit early, the number was:",number)
```

# Continue statement

- A `continue` statement, if executed in a loop, means to immediately jump back to the top of the loop and re-evaluate the conditional
- Any remaining parts of the loop are skipped for the one iteration when the `continue` was executed

## Code Listing 2.21

### Part of the guessing numbers program

```
7 # initialize the input number and the sum
8 number_str = input("Number: ")
9 the_sum = 0
10
11 # Stop if a period (.) is entered.
12 # remember, number_str is a string until we convert it
13 while number_str != ".":
14     number = int(number_str)
15     if number % 2 == 1: # number is not even (it is odd)
16         print ("Error, only even numbers please.")
17         number_str = input("Number: ")
18         continue # if the number is not even, ignore it
19     the_sum += number
20     number_str = input("Number: ")
21
22 print ("The sum is:",the_sum)
```

# change in control: Break and Continue

- while loops are easiest read when the conditions of exit are clear
- Excessive use of continue and break within a loop suite make it more difficult to decide when the loop will exit and what parts of the suite will be executed each loop.
- Use them judiciously.

# While overview

```
while test1:  
    statement_list_1  
    if test2: break      # Exit loop now; skip else  
    if test3: continue   # Go to top of loop now  
    # more statements  
else:  
    statement_list_2    # If we didn't hit a 'break'  
  
# 'break' or 'continue' lines can appear anywhere
```

# Range and for loop

# Range function

- The range function represents a sequence of integers
- the range function takes 3 arguments:
  - the beginning of the range. Assumed to be 0 if not provided
  - the end of the range, but not inclusive (up to but not including the number). Required
  - the step of the range. Assumed to be 1 if not provided
- if only one arg provided, assumed to be the end value

# Iterating through the sequence

```
for num in range(1, 5):  
    print(num)
```

- range represents the sequence 1, 2, 3, 4
- for loop assigns num to each of the values in the sequence, one at a time, in sequence
- prints each number (one number per line)

# range generates on demand

Range generates its values on demand

```
>>> range(1,10)
range(1, 10)
>>> my_range=range(1,10)
>>> type(my_range)
<class 'range'>
>>> len(my_range)
9
>>> for i in my_range:
        print(i, end=' ')
```

1 2 3 4 5 6 7 8 9

>>>

# Hailstone example

# Collatz

- The Collatz sequence is a simple algorithm applied to any positive integer
- In general, by applying this algorithm to your starting number you generate a sequence of other positive numbers, ending at 1
- Unproven whether every number ends in 1 (though strong evidence exists)

# Algorithm

while the number does not equal one

- If the number is odd, multiply by 3 and add 1
- If the number is even, divide by 2
- Use the new number and reapply the algorithm

# Even and Odd

Use the remainder operator

- if num % 2 == 0: # even
- if num % 2 == 1: # odd
- if num %2: # odd (why???)

## Code Listing 2.25

### Hailstone Sequence, loop



```
1 # Generate a hailstone sequence
2 number_str = input("Enter a positive integer:")
3 number = int(number_str)
4 count = 0
5
6 print("Starting with number:",number)
7 print("Sequence is: ", end=' ')
8
9 while number > 1: # stop when the sequence reaches 1
10
11     if number%2:          # number is odd
12         number = number*3 + 1
13     else:                  # number is even
14         number = number/2
15     print(number, ", ", end=' ')    # add number to sequence
16
17     count +=1             # add to the count
18
19 else:
20     print()      # blank line for nicer output
21     print("Sequence is ",count," numbers long")
```

# The Rules

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly



# Chapter 3

## Working with Strings

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



Pearson

PEARSON

ALWAYS LEARNING

# Sequence of characters

- We've talked about strings being a sequence of characters.
- A string is indicated between ' ' or " "
- The exact sequence of characters is maintained

# And then there is """ """

- triple quotes preserve both the vertical and horizontal formatting of the string
- allows you to type tables, paragraphs, whatever and preserve the formatting

```
" " "this is  
a test  
today" " "
```

# non-printing characters

If inserted directly, are preceded by a backslash (the \ character)

- new line                ' \n '
- tab                      ' \t '

# String Representation

- every character is "mapped" (associated) with an integer
- UTF-8, subset of Unicode, is such a mapping
- the function `ord()` takes a character and returns its UTF-8 integer value, `chr()` takes an integer and returns the UTF-8 character.

# Subset of UTF-8

See Appendix F  
for the full set

| Char | Dec | Char | Dec | Char | Dec |
|------|-----|------|-----|------|-----|
| SP   | 32  | @    | 64  | `    | 96  |
| !    | 33  | A    | 65  | a    | 97  |
| "    | 34  | B    | 66  | b    | 98  |
| #    | 35  | C    | 67  | c    | 99  |
| \$   | 36  | D    | 68  | d    | 100 |
| %    | 37  | E    | 69  | e    | 101 |
| &    | 38  | F    | 70  | f    | 102 |
| '    | 39  | G    | 71  | g    | 103 |
| (    | 40  | H    | 72  | h    | 104 |
| )    | 41  | I    | 73  | i    | 105 |
| *    | 42  | J    | 74  | j    | 106 |
| +    | 43  | K    | 75  | k    | 107 |
| ,    | 44  | L    | 76  | l    | 108 |
| -    | 45  | M    | 77  | m    | 109 |
| .    | 46  | N    | 78  | n    | 110 |
| /    | 47  | O    | 79  | o    | 111 |
| 0    | 48  | P    | 80  | p    | 112 |
| 1    | 49  | Q    | 81  | q    | 113 |
| 2    | 50  | R    | 82  | r    | 114 |
| 3    | 51  | S    | 83  | s    | 115 |
| 4    | 52  | T    | 84  | t    | 116 |

# Strings

Can use single or double quotes:

- `S = "spam"`
- `s = 'spam'`

Just don't mix them

- `my_str = 'hi mom"` ⇒ ERROR

Inserting an apostrophe:

- `A = "knight's"` # *mix up the quotes*
- `B = 'knight\'s'` # *escape single quote*

# The Index

- Because the elements of a string are a sequence, we can associate each element with an *index*, a location in the sequence:
  - positive values count up from the left, beginning with index 0
  - negative values count down from the right, starting with -1

|            |   |   |   |   |   |   |   |   |     |    |    |
|------------|---|---|---|---|---|---|---|---|-----|----|----|
| characters | H | e | I | I | o |   | W | o | r   | I  | d  |
| index      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8   | 9  | 10 |
|            |   |   |   |   |   |   |   |   | ... | -2 | -1 |

# Accessing an element

A particular element of the string is accessed by the index of the element surrounded by square brackets [ ]

```
hello_str = 'Hello World'  
print(hello_str[1])    => prints e  
print(hello_str[-1])   => prints d  
print(hello_str[11])   => ERROR
```

# Slicing, the rules

- slicing is the ability to select a subsequence of the overall sequence
- uses the syntax `[start : finish]`, where:
  - `start` is the index of where we start the subsequence
  - `finish` is the index of one after where we end the subsequence
- if either `start` or `finish` are not provided, it defaults to the beginning of the sequence for `start` and the end of the sequence for `finish`

# half open range for slices

- slicing uses what is called a half-open range
- the first index is included in the sequence
- the last index is one **after** what is included

helloString[6:10]

characters

|   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|---|----|
| H | e | I | I | o |   | W | o | r | I | d  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

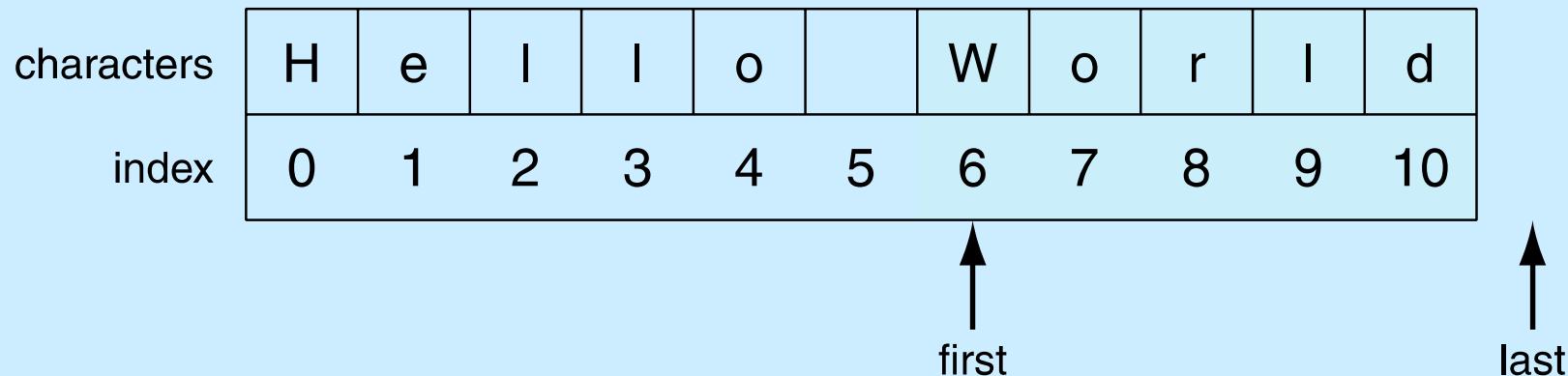


first

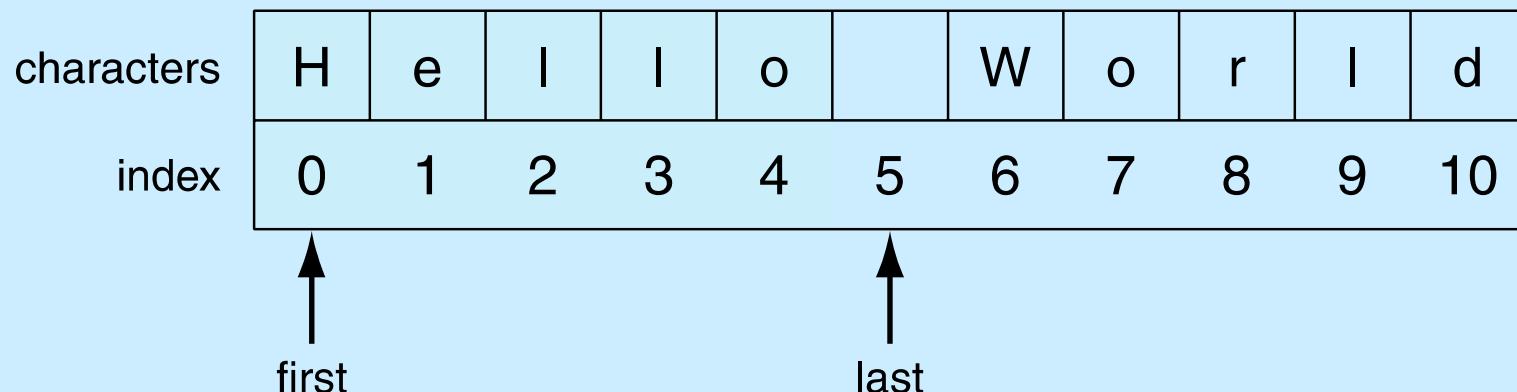


last

helloString[6:]



helloString[:5]





example.

# Extended Slicing

- also takes three arguments:
  - [start:finish:countBy]
- defaults are:
  - start is beginning, finish is end, countBy is 1

```
my_str = 'hello world'
```

```
my_str[0:11:2] ⇒ 'hlowrd'
```

- every other letter

helloString[::2]

Characters

|   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|---|----|
| H | e | I | I | o |   | W | o | r | I | d  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Index



# Some python idioms

- idioms are python “phrases” that are used for a common task that might be less obvious to non-python folk
- how to make a copy of a string:

```
my_str = 'hi mom'
```

```
new_str = my_str[:]
```

- how to reverse a string

```
my_str = "madam I'm adam"
```

```
reverseStr = my_str[::-1]
```

# String Operations

# Sequences are iterable

The for loop iterates through each element of a sequence in order. For a string, this means character by character:

```
>>> for char in 'Hi mom':  
        print(char, type(char))
```

```
H <class 'str'>  
i <class 'str'>  
 <class 'str'>  
m <class 'str'>  
o <class 'str'>  
m <class 'str'>  
>>>
```

# Basic String Operations

```
s = 'spam'
```

- length operator `len()`

```
len(s) ⇒ 4
```

- `+` is concatenate

```
new_str = 'spam' + ' - ' + 'spam - '
```

```
print(new_str) ⇒ spam-spam-
```

- `*` is repeat, the number is how many times

```
new_str * 3 ⇒
```

```
'spam-spam-spam-spam-spam-spam - '
```

# some details

- both `+` and `*` on strings makes a new string, does not modify the arguments
- order of operation is important for concatenation, irrelevant for repetition
- the types required are specific. For concatenation you need two strings, for repetition a string and an integer

# what does $a + b$ mean?

- what operation does the above represent?  
It depends on the types!
  - two strings, concatenation
  - two integers addition
- the operator `+` is **overloaded**.
  - The operation `+` performs depends on the types it is working on

# The type function

- You can check the type of the value associated with a variable using `type`

```
my_str = 'hello world'
```

```
type(my_str) ⇒ <type 'str'>
```

```
my_str = 245
```

```
type(my_str) ⇒ <type 'int'>
```

# String comparisons, single char

- Python 3 uses the Unicode mapping for characters.
  - Allows for representing non-English characters
- UTF-8, subset of Unicode, takes the English letters, numbers and punctuation marks and maps them to an integer.
- Single character comparisons are based on that number

# comparisons within sequence

- It makes sense to compare within a sequence (lower case, upper case, digits).
  - 'a' < 'b' → True
  - 'A' < 'B' → True
  - '1' < '9' → True
- Can be weird outside of the sequence
  - 'a' < 'A' → False
  - 'a' < '0' → False

# Whole strings

- Compare the first element of each string
  - if they are equal, move on to the next character in each
  - if they are not equal, the relationship between those two characters are the relationship between the strings
  - if one ends up being shorter (but equal), the shorter is smaller

# examples

- 'a' < 'b' → True
- 'aaab' < 'aaac'
  - first difference is at the last char. 'b' < 'c' so 'aaab' is less than 'aaac'. True
- 'aa' < 'aaz'
  - The first string is the same but shorter. Thus it is smaller. True

# Membership operations

- can check to see if a substring exists in the string, the `in` operator. Returns True or False

```
my_str = 'aabbcdd'
```

```
'a' in my_str => True
```

```
'abb' in my_str => True
```

```
'x' in my_str => False
```

# Strings are immutable

- strings are immutable, that is you cannot change one once you make it:
  - `a_str = 'spam'`
  - `a_str[1] = 'l'` → ERROR
- However, you can use it to make another string (copy it, slice it, etc.)
  - `new_str = a_str[:1] + 'l' + a_str[2:]`
  - `a_str` → 'spam'
  - `new_str` → 'slam'

# String methods and functions

# Functions, first cut

- a function is a program that performs some operation. Its details are hidden (encapsulated), only its interface provided.
- A function takes some number of inputs (arguments) and returns a value based on the arguments and the function's operation.

# String function: len

- The `len` function takes as an argument a string and returns an integer, the length of a string.

```
my_str = 'Hello World'
```

```
len(my_str) ⇒ 11 # space counts!
```

# String method

- a ***method*** is a variation on a function
  - like a function, it represents a program
  - like a function, it has input arguments and an output
- Unlike a function, it is applied in the context of a particular object.
- This is indicated by the *dot notation* invocation

# Example

- `upper` is the name of a method. It generates a new string that has all upper case characters of the string it was called with.

```
my_str = 'Python Rules!'  
my_str.upper()    ⇒ 'PYTHON RULES!'
```

- The `upper()` method was called in the context of `my_str`, indicated by the dot between them.

# more dot notation

- in generation, dot notation looks like:
  - object.method(...)
- It means that the object in front of the dot is calling a method that is associated with that object's type.
- The method's that can be called are tied to the type of the object calling it. Each type has different methods.

# Find

```
my_str = 'hello'  
my_str.find('l')          # find index of 'l' in  
my_str  
⇒ 2
```

Note how the method 'find' operates on the string object my\_str and the two are associated by using the “dot” notation: my\_str.find('l').

Terminology: the thing(s) in parenthesis, i.e. the 'l' in this case, is called an argument.

# Chaining methods

Methods can be chained together.

- Perform first operation, yielding an object
- Use the yielded object for the next method

```
my_str = 'Python Rules!'
```

```
my_str.upper() ⇒ 'PYTHON RULES!'
```

```
my_str.upper().find('O')
```

```
⇒ 4
```

# Optional Arguments

Some methods have optional arguments:

- if the user doesn't provide one of these, a default is assumed
- `find` has a default second argument of 0, where the search begins

```
a_str = 'He had the bat'
```

```
a_str.find('t') ⇒ 7 # 1st 't', start at 0
```

```
a_str.find('t', 8) ⇒ 13 # 2nd 't'
```

# Nesting Methods

- You can “nest” methods, that is the result of one method as an argument to another
- remember that parenthetical expressions are did “inside out”: do the inner parenthetical expression first, then the next, using the result as an argument

```
a_str.find('t', a_str.find('t')+1)
```

- translation: find the second 't'.

# How to know?

- You can use PyCharm to find available methods for any type. You enter a variable of the type, followed by the ' . ' (dot) and then a tab.
- Remember, methods match with a type. Different types have different methods
- If you type a method name, PyCharm will remind you of the needed and optional arguments.

IPython console

Console 29443/A

In [3]: my\_str.

- capitalize
- casefold
- center
- count
- encode
- endswith
- expandtabs
- find
- format
- format\_map
- index

# Figure 4.7

The screenshot shows a Jupyter Notebook interface with the title bar "Console 29443/A". In the code editor area, the following Python code is displayed:

```
In [3]: my_str.i
      index
      isalnum
      isalpha
      isdecimal
      isdigit
      isidentifier
      islower
      isnumeric
      isprintable
      isspace
      istitle
```

The word "index" is highlighted in a light gray box, indicating it is the current item being viewed or selected in the list.

## Figure 4.8

IP: Console 29443/A

```
In [3]: my_str.find(  
    Arguments  
    find(sub[, start[, end]])
```

# Figure 4.9

|                                       |   |
|---------------------------------------|---|
| capitalize( )                         | lstrip( [chars] )                       |
| center( width [, fillchar] )          | partition( sep )                        |
| count( sub [, start [, end] ] )       | replace( old, new [, count] )           |
| decode( [encoding [, errors] ] )      | rfind( sub [,start [,end] ] )           |
| encode( [encoding [,errors] ] )       | rindex( sub [, start [, end] ] )        |
| endswith( suffix [, start [, end] ] ) | rjust( width [, fillchar] )             |
| expandtabs( [tabsize] )               | rpartition( sep )                       |
| find( sub [, start [, end] ] )        | rsplit( [sep [,maxsplit] ] )            |
| index( sub [, start [, end] ] )       | rstrip( [chars] )                       |
| isalnum( )                            | split( [sep [,maxsplit] ] )             |
| isalpha( )                            | splitlines( [keepends] )                |
| isdigit( )                            | startswith( prefix [, start [, end] ] ) |
| islower( )                            | strip( [chars] )                        |
| isspace( )                            | swapcase( )                             |
| istitle( )                            | title( )                                |
| isupper( )                            | translate( table [, deletechars] )      |
| join(seq)                             | upper( )                                |
| lower( )                              | zfill( width)                           |
| ljust( width[, fillchar] )            |   |

**TABLE 4.2** Python String Methods

# String formatting

CSE 231, Bill Punch

# String formatting, better printing

- So far, we have just used the defaults of the print function
- We can do many more complicated things to make that output “prettier” and more pleasing.
- We will use it in our display function

# Basic form

- To understand string formatting, it is probably better to start with an example.

```
print( "Sorry, is this the {} minute  
{}? ".format(5, 'ARGUMENT' ) )
```

prints Sorry, is this the 5 minute  
ARGUMENT

# format method

- `format` is a method that creates a new string where certain elements of the string are re-organized i.e., *formatted*
- The elements to be re-organized are the curly bracket elements in the string.
- Formatting is complicated, this is just some of the easy stuff (see the docs)

# map args to { }

- The string is modified so that the { } elements in the string are replaced by the format method arguments
- Their replacement is in order: first { } is replaced by the first argument, second { } by the second argument and so forth.

*string indicated by quotes*

```
print('Sorry, is this the { } minute { }?' .format(5,'ARGUMENT'))
```

Sorry, is this the 5 minute ARGUMENT?

# Format string

- the content of the curly bracket elements are the format string, descriptors of how to organize that particular substitution.
  - types are the kind of thing to substitute, numbers indicate total spaces.

|   |        |
|---|--------|
| < | left   |
| > | right  |
| ^ | center |

**TABLE 4.4** Width alignments.

# Each format string

- Each bracket looks like

```
{:align width .precision type}
```

- align is optional (default left)
- width is how many spaces (default just enough)
- .precision is for floating point rounding (default no rounding)
- type is the expected type (error if the arg is the wrong type)

```
print('{:>10s} is {:<10d} years old.' format('Bill', 25))
```

String 10 spaces wide  
including the object,  
right justified (>).

Decimal 10 spaces wide  
including the object,  
left justified (<).

OUTPUT:

Bill is 25 years old.  
[ ] [ ]  
10 spaces 10 spaces

**FIGURE 4.11** String formatting with width descriptors and alignment.

# Nice table

```
>>> for i in range(5):  
    print("{ :10d} --> { :4d}".format(i,i**2))
```

|   |     |    |
|---|-----|----|
| 0 | --> | 0  |
| 1 | --> | 1  |
| 2 | --> | 4  |
| 3 | --> | 9  |
| 4 | --> | 16 |

# Floating Point Precision

Can round floating point to specific number of decimal places

# iteration

# iteration through a sequence

- To date we have seen the while loop as a way to iterate over a suite (a group of python statements)
- We briefly touched on the for statement for iteration, such as the elements of a list or a string

# for statement

We use the for statement to process each element of a list, one element at a time

```
for item in sequence:  
    suite
```

# What for means

```
my_str='abc'  
for char in 'abc':  
    print(char)
```

- first time through, char = 'a' (my\_str[0])
- second time through, char='b' (my\_str[1])
- third time through, char='c' (my\_str[2])
- no more sequence left, for ends

# Power of the for statement

- Sequence iteration as provided by the for state is very powerful and very useful in python.
- Allows you to write some very “short” programs that do powerful things.

## Code Listing 4.1

### Find a letter

```
1 # Our implementation of the find function. Prints the index where
2 # the target is found; a failure message, if it isn't found.
3 # This version only searches for a single character.
4
5 river = 'Mississippi'
6 target = input('Input a character to find: ')
7 for index in range(len(river)):           # for each index
8     if river[index] == target:            # check if the target is found
9         print("Letter found at index: ", index) # if so, print the index
10        break                           # stop searching
11 else:
12     print('Letter',target,'not found in',river)
```

# enumerate function

- The enumerate function prints out two values: the index of an element and the element itself
- Can use it to iterate through both the index and element simultaneously, doing dual assignment

## Code Listings 4.2

### find with enumerate

```
# Our implementation of the find function. Prints the index where
# the target is found; a failure message, if it isn't found.
# This version only searches for a single character.

river = 'Mississippi'
target = input('Input a character to find: ')
for index,letter in enumerate(river):                      # for each index
    if letter == target:                                    # check if the target is found
        print("Letter found at index: ", index)            # if so, print the index
        break                                              # stop searching
else:
    print('Letter',target,'not found in',river)
```

# split function

- The `split` function will take a string and break it into multiple new string parts depending on the argument character.
- by default, if no argument is provided, split is on any whitespace character (tab, blank, etc.)
- you can assign the pieces with multiple assignment if you know how many pieces are yielded.

# reorder a name

```
>>> name = 'John Marwood Cleese'  
>>> first, middle, last = name.split()  
>>> transformed = last + ', ' + first + ' ' + middle  
>>> print(transformed)  
Cleese, John Marwood  
>>> print(name)  
John Marwood Cleese  
>>> print(first)  
John  
>>> print(middle)  
Marwood
```

# Palindromes and the rules

- A palindrome is a string that prints the same forward and backwards
- same implies that:
  - case does not matter
  - punctuation is ignored
- "Madam I'm Adam" is thus a palindrome

# lower case and punctuation

- every letter is converted using the `lower` method
- `import string`, brings in a series of predefined sequences (`string.digits`,  
`string.punctuation`,  
`string.whitespace`)
- we remove all non-wanted characters with the `replace` method. First arg is what to replace, the second the replacement.

## Code Listing 4.4

### Palindromes

```
1 # Palindrome tester
2 import string
3
4 original_str = input('Input a string:')
5 modified_str = original_str.lower()
6
7 bad_chars = string.whitespace + string.punctuation
8
9 for char in modified_str:
10     if char in bad_chars: # remove bad characters
11         modified_str = modified_str.replace(char, '')
12
13 if modified_str == modified_str[::-1]: # it is a palindrome
14     print(\n
15 'The original string is: {}\n\
16 the modified string is: {}\n\
17 the reversal is:       {}\n\
18 String is a palindrome'.format(original_str, modified_str, modified_str[::-1]))
19 else:
20     print(\n
21 'The original string is: {}\n\
22 the modified string is: {}\n\
23 the reversal is:       {}\n\
24 String is not a palindrome'.format(original_str, modified_str, modified_str[::-1]))
```

# More String Formatting

We said a format string was of the following form:

```
{ :align width .precision type }
```

Well, it can be more complicated than that

```
{arg : fill= align sign # 0  
width , .precision type }
```

That's a lot, so let's look at the details

# arg

To over-ride the {}-to-argument matching we have seen, you can indicate the argument you want in the bracket

- if other descriptor stuff is needed, it goes behind the arg, separated by a :

```
>>> print('{0} is {2} and {0} is also {1}'.format('Bill',25,'tall'))  
Bill is tall and Bill is also 25
```

# `fill, =`

Besides alignment, you can fill empty spaces with a fill character:

- `0=` fill with 0's
- `+=` fill with +

# sign

- + means a sign for both positive and negative numbers
- - means a sign for only negative numbers
- space means space for positive, minus for negative

# example

args are before the :, format after

```
>>> print('{0:>12s} / {1:0=+10d} / {2:->5d}'.format('abc', 35, 22))  
.....abc | +000000035 | ---22
```

for example {1:0=10d} means:

- 1 → second (count from 0) arg of format, 35
- : → separator
- 0= → fill with 0's
- + → plus or minus sign
- 10d → occupy 10 spaces (left justify)  
decimal

# # , and 0

- # is complicated, but the simple version is that it forces a decimal point
- 0 forces fill of zero's (equivalent to 0=)
- , put commas every three digits

```
>>> print('{ :#6.0f}'.format(3)) # decimal point forced  
3.
```

```
>>> print('{ :04d}'.format(4)) # zero preceeds width  
0004  
>>> print('{ :,d}'.format(1234567890))  
1,234,567,890
```

# nice for tables

```
>>> for n in range(3,11):  
    print('{:4}-sides{:6}{:10.2f}{:10.2f}'.format(n,180*(n-2),180*(n-2)/n,360/n))
```

|           |      |        |        |
|-----------|------|--------|--------|
| 3-sides:  | 180  | 60.00  | 120.00 |
| 4-sides:  | 360  | 90.00  | 90.00  |
| 5-sides:  | 540  | 108.00 | 72.00  |
| 6-sides:  | 720  | 120.00 | 60.00  |
| 7-sides:  | 900  | 128.57 | 51.43  |
| 8-sides:  | 1080 | 135.00 | 45.00  |
| 9-sides:  | 1260 | 140.00 | 40.00  |
| 10-sides: | 1440 | 144.00 | 36.00  |

# Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.

GLOBAL  
EDITION



# chapter 4

## Functions -- QuickStart

PEARSON

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



P Pearson

ALWAYS LEARNING

# What is a function?

# Functions

- From Mathematics we know that functions perform some operation and return one value.
- They "encapsulate" the performance of some particular operation, so it can be used by others (for example, the `sqrt()` function)

# Why have them?

- Support divide-and-conquer strategy
- Abstraction of an operation
- Reuse. Once written, use again
- Sharing. If tested, others can use
- Security. Well tested, then secure for reuse
- Simplify code. More readable.

# Mathematical Notation

- Consider a function which converts temperatures in Celsius to temperatures in Fahrenheit.
  - Formula:  $F = C * 1.8 + 32.0$
  - Functional notation:  
 $F \sim \text{celsius\_to\_Fahrenheit}(C)$  where  
 $\text{celsius\_to\_Fahrenheit}(C) = C * 1.8 + 32.0$

# Python Invocation

- Math:  $F = \text{celsius\_to\_Fahrenheit}(C)$
- Python, the invocation is much the same

```
F = celsius_to_Fahrenheit(cel_float)
```

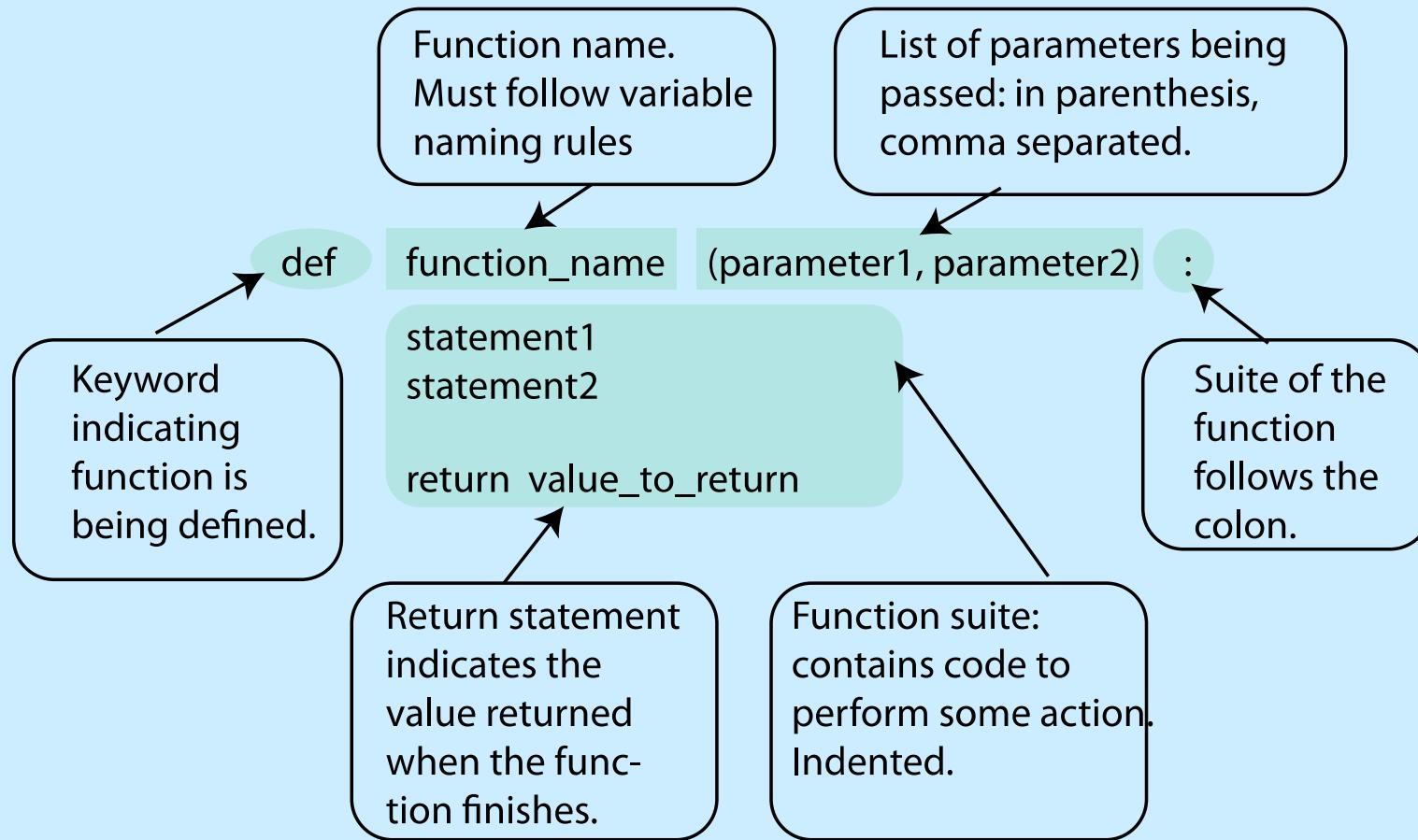
Terminology: cel\_float is the ***argument***

# Function definition

- Math:  $g(C) = C * 1.8 + 32.0$
- Python

```
def celsius_to_Fahrenheit(param_float):  
    return param_float * 1.8 + 32.0
```

- Terminology: `cel_float` is the ***parameter***



## Figure 5.1 Function Parts

# return statement

- The `return` statement indicates the value that is returned by the function
- The statement is optional (the function can return nothing). If no `return`, function is often called a procedure.

## Code Listing 5.1

### Temp convert

```
1 # Temperature conversion
2
3 def celsius_to_fahrenheit(celsius_float):
4     """ Convert Celsius to Fahrenheit. """
5     return celsius_float * 1.8 + 32
```

---

# Triple quoted string in function

- A triple quoted string just after the def is called a ***docstring***
- docstring is documentation of the function's purpose, to be used by other tools to tell the user what the function is used for. More on that later

# Operation

```
F = celsius_to_fahrenheit(C)
```

1. Call copies argument C to parameter Temp

2. Control transfers to function

```
def celsius_to_Fahrenheit (param):  
    return param * 1.8 + 32.0
```

# Operation (con't)

```
F = celsius_to_fahrenheit(C)
```

4. Value of expression is returned to the invoker

3. Expression in function is evaluated

```
def celsius_to_Fahrenheit (param):  
    return param * 1.8 + 32.0
```

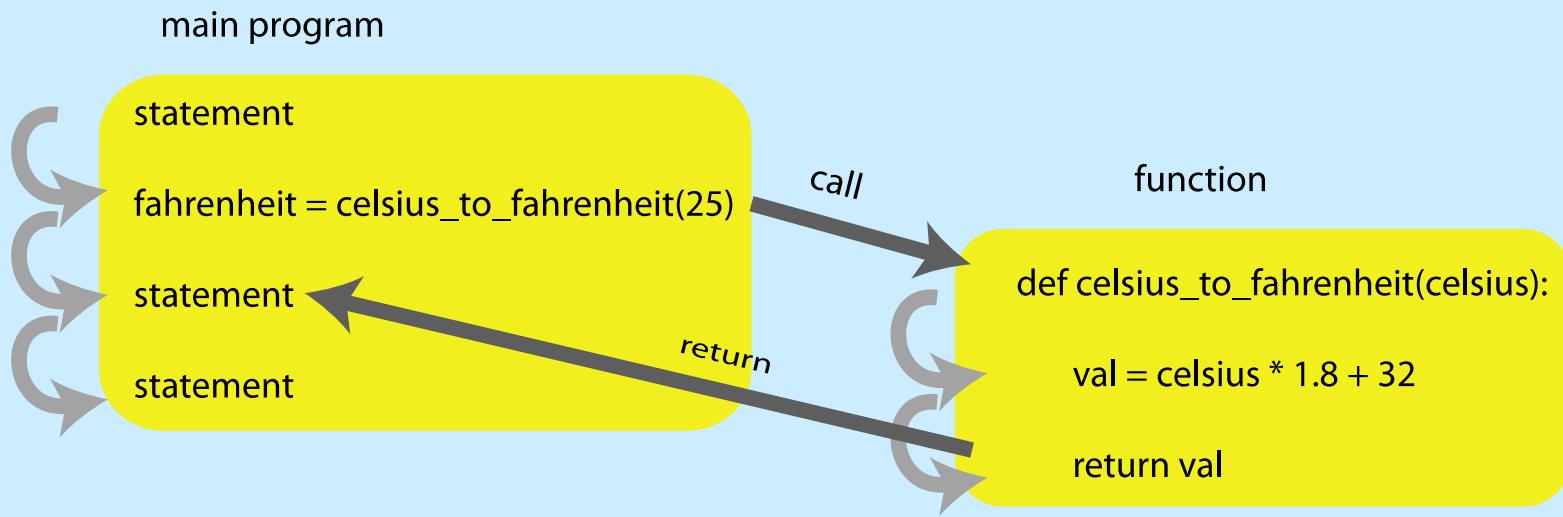


Figure 5.1 Function flow of control

## Code Listing 5.2

### Full Temp Program

```
1 # Conversion program
2
3 def celsius_to_fahrenheit(celsius_float):
4     """ Convert Celsius to Fahrenheit."""
5     return celsius_float * 1.8 + 32
6
7 # main part of the program
8 print("Convert Celsius to Fahrenheit.")
9 celsius_float = float(input("Enter a Celsius temp: "))
10 # call the conversion function
11 fahrenheit_float = celsius_to_fahrenheit(celsius_float)
12 # print the returned value
13 print(celsius_float, " converts to ", fahrenheit_float, " Fahrenheit")
```

## Code Listing 5.3

### digit extraction

```
def get_digit(number, position):
    '''return digit at position in number, counting from right'''
    return number//(10**position)%10
```

# Area of a Triangle

The next few functions can be used together to find the area of a triangle.

Note how we decompose the problem and then re-assemble the overall solution using the functions created

## Code Listing 5.4

### Input



```
def get_vertex():
    x = float(input("      Please enter x: "))
    y = float(input("      Please enter y: "))
    return x, y
```

## Code Listing 5.5

### get\_triangle

```
def get_triangle():
    triangle = []
    for i in range(3):
        triangle.append([int(x) for x in input().split()])
    return triangle
```

```
def get_triangle():
    print("First vertex")
    x1,y1 = get_vertex()
    print("Second vertex")
    x2,y2 = get_vertex()
    print("Third vertex")
    x3,y3 = get_vertex()
    return x1, y1, x2, y2, x3, y3
```

## Code Listing 5.6

### side\_length

```
def side_length(x1,y1,x2,y2):
    ''' return length of a side (Euclidean distance) '''
return math.sqrt((x1-x2)**2 + (y1-y2)**2)
```

## Code Listing 5.7

### calculate\_area

```
def calculate_area(x1,y1,x2,y2,x3,y3):
    ''' return area using Heron's formula '''
    a = side_length(x1,y1,x2,y2)
    b = side_length(x2,y2,x3,y3)
    c = side_length(x3,y3,x1,y1)
    s = (1/2)*(a + b + c)
    return math.sqrt(s*(s-a)*(s-b)*(s-c))
```

## Code Listing 5.8

### Full Triangle Program

```
import math
```

```
def get_vertex():
    x = float(input("    Please enter x: "))
    y = float(input("    Please enter y: "))
    return x,y
```

```
def get_triangle():
    print("First vertex")
    x1,y1 = get_vertex()
    print("Second vertex")
    x2,y2 = get_vertex()
    print("Third vertex")
    x3,y3 = get_vertex()
    return x1, y1, x2, y2, x3, y3
```

```
def side_length(x1,y1,x2,y2):
    "return length of a side (Euclidean distance)"
    return math.sqrt((x1-x2)**2 + (y1-y2)**2)
```

```
def calculate_area(x1,y1,x2,y2,x3,y3):
    "return area using Heron's formula"
    a = side_length(x1,y1,x2,y2)
    b = side_length(x2,y2,x3,y3)
    c = side_length(x3,y3,x1,y1)
    s = (1/2)*(a + b + c)
    return math.sqrt(s*(s-a)*(s-b)*(s-c))
```

```
x1, y1, x2, y2, x3, y3 = get_triangle()
area = calculate_area(x1,y1,x2,y2,x3,y3)
print("Area is",area)
```

# Did functions help?

- Made our problem solving easier (solved smaller problems as functions)
- main program very readable (details hid in the functions)

# How to write a function

- ***Does one thing.*** If it does too many things, it should be broken down into multiple functions (refactored)
- ***Readable.*** How often should we say this? If you write it, it should be readable
- ***Reusable.*** If it does one thing well, then when a similar situation (in another program) occurs, use it there as well.

# More on functions

- ***Complete.*** A function should check for all the cases where it might be invoked. Check for potential errors.
- ***Not too long.*** Kind of synonymous with do one thing. Use it as a measure of doing too much.

# Rule 8

A function should do one thing

# Procedures

- Functions that have no return statements are often called *procedures*.
- Procedures are used to perform some duty (print output, store a file, etc.)
- Remember, return is not required.

# Multiple returns in a function

- A function can have multiple `return` statements.
- Remember, the first `return` statement executed ends the function.
- Multiple returns can be confusing to the reader and should be used judiciously.

# Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.

GLOBAL  
EDITION



# Chapter 5

## Files and Exceptions I

PEARSON

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



P Pearson

ALWAYS LEARNING

# What is a file?

- A file is a collection of data that is stored on secondary storage like a disk or a thumb drive
- accessing a file means establishing a connection between the file and the program and moving data between the two

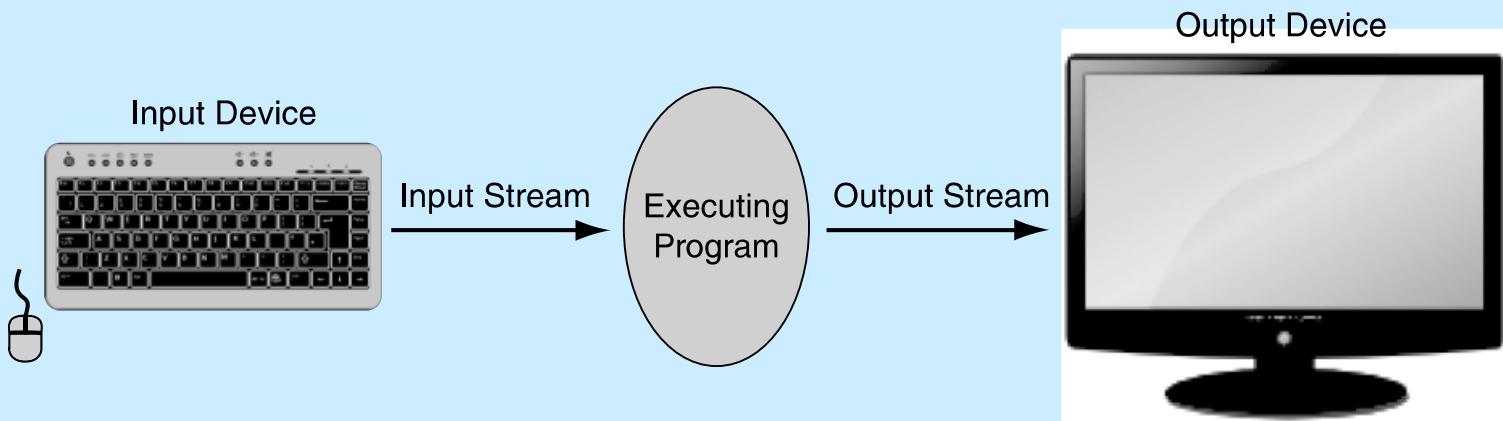
# Two types of files

Files come in two general types:

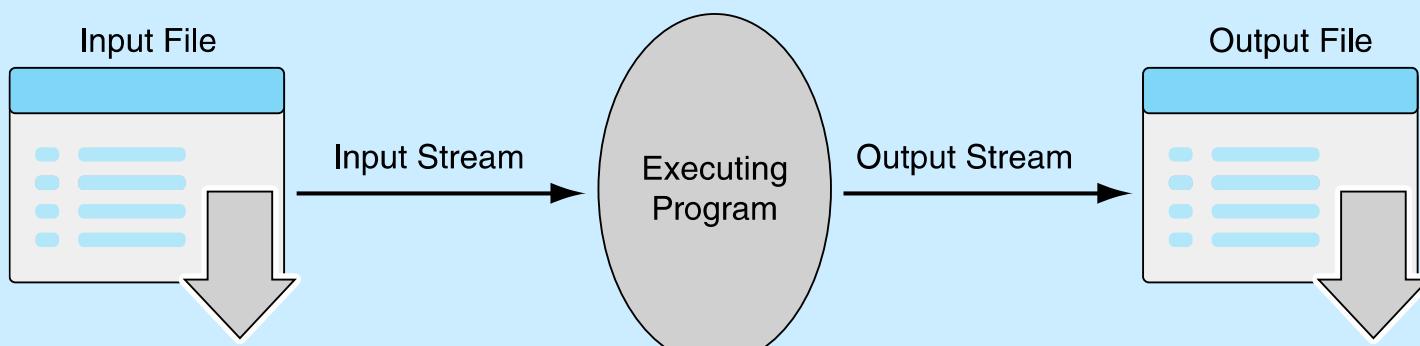
- *text files*. Files where control characters such as "/n" are translated. This are generally human readable
- *binary files*. All the information is taken directly without translation. Not readable and contains non-readable info.

# File Objects or stream

- When opening a file, you create a file object or file stream that is a connection between the file information on disk and the program.
- The stream contains a buffer of the information from the file, and provides the information to the program



a) Standard input and output.



b) File input and output.

**Figure 6.1 Input-output streams.**

# Buffering

- Reading from a disk is very slow. Thus the computer will read a lot of data from a file in the hopes that, if you need the data in the future, it will be buffered in the file object.
- This means that the file object contains a copy of information from the file called a cache (pronounced "cash")

# Making a file object

```
my_file = open("my_file.txt", "r")
```

- `my_file` is the file object. It contains the buffer of information. The `open` function creates the connection between the disk file and the file object. The first quoted string is the file name on disk, the second is the mode to open it (here, "`r`" means to read)

# Where is the disk file?

- When opened, the name of the file can come in one of two forms:
- "file.txt" assumes the file name is file.txt and it is located in the current program directory
- "c:\bill\file.txt" is the fully qualified file name and includes the directory information

# Different modes

| Mode | How Opened     | File Exists  | File Does Not Exist                 |
|------|----------------|--|-------------------------------------|
| 'r'  | read-only      | <b>Opens that file</b>   | Error                               |
| 'w'  | write-only     | Clears the file contents   | Creates and opens a new file        |
| 'a'  | write-only     | <b>File contents left intact and new data appended at file's end</b> | <b>Creates and opens a new file</b> |
| 'r+' | read and write | Reads and overwrites from the file's beginning                       | Error                               |
| 'w+' | read and write | <b>Clears the file contents</b>                                      | <b>Creates and opens a new file</b> |
| 'a+' | read and write | File contents left intact and read and write at file's end           | Creates and opens a new file        |

Table 6.1 File modes.

# Careful with write modes

- Be careful if you open a file with the '`w`' mode. It sets an existing file's contents to be empty, destroying any existing data.
- The '`a`' mode is nicer, allowing you to write to the end of an existing file without changing the existing contents

# Text files use strings

- If you are interacting with text files (which is all we will do in this book), remember that *everything is a string*
  - everything read is a string
  - if you write to a file, you can only write a string

# writing to a file

Once you have created a file object, opened for reading, you can use the print command

- you add `file=file` to the print command

```
# open file for writing:  
#     creates file if it does not exist  
#     overwrites file if it exists  
>>> temp_file = open("temp.txt", "w")  
>>> print("first line", file=temp_file)  
>>> print("second line", file=temp_file)  
>>> temp_file.close()
```

# close

When the program is finished with a file, we close the file

- flush the buffer contents from the computer to the file
- tear down the connection to the file
- `close` is a method of a file obj  
`file_obj.close()`
- All files should be closed!

## Code Listing 6.1

### Reverse file lines

```
input_file = open("input.txt", "r")
output_file = open("output.txt", "w")

for line_str in input_file:
    new_str = ''
    line_str = line_str.strip()                      # get rid of carriage return
    for char in line_str:
        new_str = char + new_str                    # concat at the left (reverse)
    print(new_str,file=output_file)                # print to output_file

# include a print to shell so we can observe progress
print('Line: {:12s} reversed is: {:s}'.format(line_str, new_str))
input_file.close()
output_file.close()
```

# Word Puzzle

The following listings show how one might solve the following puzzle: look through a file of words, one word per line, and identify any word that has all the vowels in order, with only one example of each vowel.

For example, "facetious"

## Code Listing 6.3

### clean\_word

```
def clean_word(word):
    """Return word in lower case stripped of whitespace."""
    return word.strip().lower()
```

## Code Listing 6.4

```
data_file = open("dictionary.txt", "r")

def clean_word(word):
    """Return word in lower case stripped of whitespace."""
    return word.strip().lower()

# main program
for word in data_file:          # for each word in the file
    word = clean_word(word)   # clean the word
    if len(word) <= 6:        # if word is too small, skip it
        continue
    print(word)
```

## Code Listing 6.5

### get\_vowels

```
def get_vowels_in_word(word):
    """Return vowels in string word--include repeats."""
    vowel_str = "aeiou"
    vowels_in_word = ""
    for char in word:
        if char in vowel_str:
            vowels_in_word += char
    return vowels_in_word
```

## Code Listing 6.6

### Full Solution

```
# Find a word with a single example of the vowels a, e, i, o, u in that order

data_file = open("dictionary.txt", "r")

def clean_word(word):
    """Return word in lower case stripped of whitespace."""
    return word.strip().lower()

def get_vowels_in_word(word):
    """Return vowels in string word--include repeats."""
    vowel_str = "aeiou"
    vowels_in_word = ""
    for char in word:
        if char in vowel_str:
            vowels_in_word += char
    return vowels_in_word

# main program
print("Find words containing vowels 'aeiou' in that order:")
for word in data_file:      # for each word in the file
    word = clean_word(word) # clean the word
    if len(word) <= 6:       # if word is too small, skip it
        continue
    vowel_str = get_vowels_in_word(word) # get vowels in word
    if vowel_str == 'aeiou':           # check all vowels in order
        print(word)
```

# Exceptions

## First Cut

# How to deal with problems

- Most modern languages provide methods to deal with ‘exceptional’ situations
- Gives the programmer the option to keep the user from having the program stop without warning
- Again, this is not about fundamental CS, but about doing a better job as a programmer

# What counts as exceptional

- errors. indexing past the end of a list, trying to open a nonexistent file, fetching a nonexistent key from a dictionary, etc.
- events. search algorithm doesn't find a value (not really an error), mail message arrives, queue event occurs

# exceptions (2)

- ending conditions. File should be closed at the end of processing, list should be sorted after being filled
- weird stuff. For rare events, keep from clogging your code with lots of if statements.

# Error Names

Errors have specific names, and Python shows them to us all the time.

```
>>> input_file = open("no_such_file.txt", 'r')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    input_file = open("no_such_file.txt", 'r')
IOError: [Errno 2] No such file or directory: 'no_such_file.txt'
>>> my_int = int('a string')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    my_int = int('a string')
ValueError: invalid literal for int() with base 10: 'a string'
>>>
```

You can recreate an error to find the correct name. Spelling counts!

# a kind of non-local control

Basic idea:

- keep watch on a particular section of code
- if we get an exception, raise/throw that exception (let it be known)
- look for a catcher that can handle that kind of exception
- if found, handle it, otherwise let Python handle it (which usually halts the program)

# Doing better with input

- In general, we have assumed that the input we receive is correct (from a file, from the user).
- This is almost never true. There is always the chance that the input could be wrong
- Our programs should be able to handle this.

# Worse yet, input is evil

- "Writing Secure Code", by Howard and LeBlanc
  - “All input is evil until proven otherwise”
- Most security holes in programs are based on assumptions programmers make about input
- Secure programs protect themselves from evil input

# Rule 7

All input is evil, until proven otherwise

# General form, version 1

```
try:  
    suite  
except a_particular_error:  
    suite
```

# try suite

- the `try` suite contains code that we want to monitor for errors during its execution.
- if an error occurs anywhere in that `try` suite, Python looks for a handler that can deal with the error.
- if no special handler exists, Python handles it, meaning the program halts and with an error message as we have seen so many times ☹

# except suite

- an `except` suite (perhaps multiple `except` suites) is associated with a `try` suite.
- each exception names a type of exception it is monitoring for.
- if the error that occurs in the `try` suite matches the type of exception, then that `except` suite is activated.

# try/except group

- if no exception in the `try` suite, skip all the `try/except` to the next line of code
- if an error occurs in a `try` suite, look for the right exception
- if found, run that `except` suite and then skip past the `try/except` group to the next line of code
- if no exception handling found, give the error to Python

## Code Listing 6.7

### Find a line in a file

```
# read a particular line from a file. User provides both the line
# number and the file name

file_str = input( "Open what file:" )
find_line_str = input( "Which line (integer):" )

try:
    input_file = open(file_str)                      # potential user error
    find_line_int = int(find_line_str)      # potential user error
    line_count_int = 1
    for line_str in input_file:
        if line_count_int == find_line_int:
            print("Line {} of file {} is {}".format(find_line_int, file_str,
line_str))
            break
        line_count_int += 1
    else:
        # get here if line sought doesn't exist
        print("Line {} of file {} not found".format(find_line_int, file_str))
    input_file.close()

except FileNotFoundError:
    print("The file", file_str , "doesn't exist.")

except ValueError:
    print("Line", find_line_str, " Isn't a legal line number.")

print("End of the program")
```

# Counting Poker Hands

# Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.

GLOBAL  
EDITION



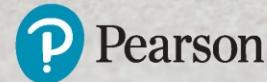
# chapter 6

## Lists and Tuples

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



PEARSON

ALWAYS LEARNING

# Data Structures

# Data Structures and algorithms

- Part of the "science" in computer science is the design and use of data structures and algorithms
- As you go on in CS, you will learn more and more about these two areas

# Data Structures

- Data structures are particular ways of storing data to make some operation easier or more efficient. That is, they are tuned for certain tasks
- Data structures are suited to solving certain problems, and they are often associated with algorithms.

# Kinds of data structures

Roughly two kinds of data structures:

- built-in data structures, data structures that are so common as to be provided by default
- user-defined data structures (classes in object oriented programming) that are designed for a particular task

# Python built in data structures

- Python comes with a general set of built in data structures:
  - lists
  - tuples
  - string
  - dictionaries
  - sets
  - others...

# Lists

# The Python List Data Structure

- a list is an ordered sequence of items.
- you have seen such a sequence before in a string. A string is just a particular kind of list (what kind)?

# Make a List

- Like all data structures, lists have a ***constructor***, named the same as the data structure. It takes an iterable data structure and ***adds each item*** to the list
- It also has a shortcut, the use of square brackets [ ] to indicate explicit items.

# make a list

```
>>> a_list = [1,2,'a',3.14159]
>>> week_days_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists = [[1,2,3], ['a','b','c']]
>>> list_from_collection = list('Hello')
>>> a_list
[1, 2, 'a', 3.141589999999999]
>>> week_days_list
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists
[[1, 2, 3], ['a', 'b', 'c']]
>>> list_from_collection
['H', 'e', 'l', 'l', 'o']
>>> []
[]
>>>
```

# Similarities with strings

- concatenate/+ (but only of lists)
- repeat/\*
- indexing (the [ ] operator)
- slicing ([:])
- membership (the in operator)
- len (the length operator)

# Operators

[1, 2, 3] + [4] ⇒ [1, 2, 3, 4]

[1, 2, 3] \* 2 ⇒ [1, 2, 3, 1, 2, 3]

1 in [1, 2, 3] ⇒ True

[1, 2, 3] < [1, 2, 4] ⇒ True

compare index to index, first difference  
determines the result

# differences between lists and strings

- lists can contain a mixture of any python object, strings can only hold characters
  - 1,"bill",1.2345, True
- lists are mutable, their values can be changed, while strings are immutable
- lists are designated with [ ], with elements separated by commas, strings use " " or ''

```
myList = [1, 'a', 3.14159, True]
```

myList

|    |     |         |      |
|----|-----|---------|------|
| 1  | 'a' | 3.14159 | True |
| 0  | 1   | 2       | 3    |
| -4 | -3  | -2      | -1   |

Index forward

Index backward

```
myList[1] → 'a'
```

```
myList[:3] → [1, 'a', 3.14159]
```

**FIGURE 7.1** The structure of a list.

# Indexing

- can be a little confusing, what does the [ ] mean, a list or an index?

[1, 2, 3][1]  $\Rightarrow$  2

- Context solves the problem. Index always comes at the end of an expression, and is preceded by something (a variable, a sequence)

# List of Lists

```
my_list = ['a', [1, 2, 3], 'z']
```

- What is the second element (index 1) of that list? Another list.

```
my_list[1][0] # apply left to right
```

```
my_list[1] ⇒ [1, 2, 3]
```

```
[1, 2, 3][0] ⇒ 1
```

# List Functions

- `len(lst)`: number of elements in list (top level). `len([1, [1, 2], 3])`  $\Rightarrow$  3
- `min(lst)`: smallest element. Must all be the same type!
- `max(lst)`: largest element, again all must be the same type
- `sum(lst)`: sum of the elements, numeric only

# Iteration

You can iterate through the elements of a list like you did with a string:

```
>>> my_list = [1,3,4,8]
>>> for element in my_list:      # iterate through list elements
        print(element ,end=' ') # prints on one line
```

```
1 3 4 8
```

```
>>>
```

# Mutable

# Change an object's contents

- strings are immutable. Once created, the object's contents cannot be changed. New objects can be created to reflect a change, but the object itself cannot be changed

```
my_str = 'abc'  
my_str[0] = 'z'    # cannot do!  
# instead, make new str  
new_str = my_str.replace('a','z')
```

# Lists are mutable

Unlike strings, lists are mutable. You ***can*** change the object's contents!

```
my_list = [1, 2, 3]
my_list[0] = 127
print(my_list) => [127, 2, 3]
```

# List methods

- Remember, a function is a small program (such as `len`) that takes some arguments, the stuff in the parenthesis, and returns some value
- a method is a function called in a special way, the ***dot call***. It is called in the context of an object (or a variable associated with an object)

# Again, lists have methods

```
my_list = ['a', 1, True]
```

```
my_list.append('z')
```



the object that  
we are calling the  
method with

the name of  
the method

arguments to  
the method

# Some new methods

- A list is mutable and can change:

- `my_list[0] = 'a'` #index assignment
  - `my_list.append()`, `my_list.extend()`
  - `my_list.pop()`
  - `my_list.insert()`, `my_list.remove()`
  - `my_list.sort()`
  - `my_list.reverse()`

# More about list methods

- most of these methods ***do not return a value***
- This is because lists are mutable, so the methods modify the list directly. No need to return anything.
- Can be confusing

# Unusual results

```
my_list = [4, 7, 1, 2]
my_list = my_list.sort()
my_list => None      # what happened?
```

What happened was the sort operation changed the order of the list in place (right side of assignment). Then the sort method returned `None`, which was assigned to the variable. The list was lost and `None` is now the value of the variable.

# Split

- The string method split generates a sequence of characters by splitting the string at certain split-characters.
- ***It returns a list*** (we didn't mention that before)

```
split_list = 'this is a test'.split()  
split_list  
⇒ ['this', 'is', 'a', 'test']
```

# Sorting

Only lists have a built in sorting method.  
Thus you often convert your data to a list if it  
needs sorting

```
my_list = list('xyzabc')  
my_list → ['x', 'y', 'z', 'a', 'b', 'c']  
my_list.sort()      # no return  
my_list →  
['a', 'b', 'c', 'x', 'y', 'z']
```

# reverse words in a string

join method of string places the calling  
string between every element of a list

```
>>> my_str = 'This is a test'  
>>> string_elements = my_str.split()                      # list of words  
>>> string_elements  
['This', 'is', 'a', 'test']  
>>> reversed_elements = []  
>>> for element in string_elements:                      # for each word  
...     reversed_elements.append(element[::-1])      # reverse, append  
...  
>>> reversed_elements  
['sihT', 'si', 'a', 'tset']  
>>> new_str = ' '.join(reversed_elements)    # join with space separator  
>>> new_str  
'sihT si a tset'                                         # each words reversed  
>>>
```

# Sorted function

The `sorted` function will break a sequence into elements and sort the sequence, placing the results in a list

```
sort_list = sorted('hi mom')
```

```
sort_list ⇒
```

```
[ ' ', 'h', 'i', 'm', 'm', 'o' ]
```

# Some Examples

# Anagram example

- Anagrams are words that contain the same letters arranged in a different order.  
For example: 'iceman' and 'cinema'
- Strategy to identify anagrams is to take the letters of a word, sort those letters, than compare the sorted sequences. Anagrams should have the same sorted sequence

# Code Listing

7.1

```
1 def are_anagrams(word1, word2):
2     """Return True, if words are anagrams."""
3     #2. Sort the characters in the words
4     word1_sorted = sorted(word1)      # sorted returns a sorted list
5     word2_sorted = sorted(word2)
6
7     #3. Check that the sorted words are identical.
8     if word1_sorted == word2_sorted:  # compare sorted lists
9         return True
10    else:
11        return False
```

## Code Listing 7.3

### Full Program

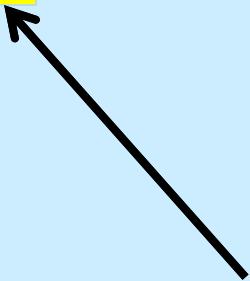
```
def are_anagrams(word1, word2) :  
    """Return True, if words are anagrams. """  
    #2. Sort the characters of the words.  
    word1_sorted = sorted(word1)      # sorted returns a sorted list  
    word2_sorted = sorted(word2)  
  
    #3. Check that the sorted words are identical.  
    return word1_sorted == word2_sorted  
  
print("Anagram Test")  
  
# 1. Input two words.  
two_words = input("Enter two space separated words: ")  
word1,word2 = two_words.split()  # split into a list of words  
  
if are_anagrams(word1, word2) :  # return True or False  
    print("The words are anagrams.")  
else:  
    print("The words are not anagrams.")
```

## Code Listing 7.4

### Check those errors

# repeat input prompt for valid input

```
valid_input_bool = False  
while not valid_input_bool:  
    try:  
        two_words = input("Enter two ...")  
        word1, word2 = two_words.split()  
        valid_input_bool = True  
except ValueError:  
    print("Bad Input")
```



only runs when no error,  
otherwise go around again

```

def are_anagrams(word1, word2):
    """Return True, if words are anagrams."""
    #2. Sort the characters of the words.
    word1_sorted = sorted(word1)      # sorted returns a sorted list
    word2_sorted = sorted(word2)

    #3. Check that the sorted words are identical.
    return word1_sorted == word2_sorted

print("Anagram Test")

# 1. Input two words, checking for errors now
valid_input_bool = False
while not valid_input_bool:
    try:
        two_words = input("Enter two space separated words: ")
        word1,word2 = two_words.split()  # split the input string into a list
                                         # of words
        valid_input_bool = True
    except ValueError:
        print("Bad Input")

if are_anagrams(word1, word2):    # function returned True or False
    print("The words {} and {} are anagrams.".format(word1, word2))
else:
    print("The words {} and {} are not anagrams.".format(word1, word2))

```

## Code Listing 7.5

### Words from text file

```
def make_word_list(a_file):
    """Create a list of words from the file."""
    word_list = []          # list of speech words: initialized to be empty

    for line_str in a_file:           # read file line by line
        line_list = line_str.split() # split each line into a list of words
        for word in line_list:       # get words one at a time from list
            if word != "--":         # if the word is not "--"
                word_list.append(word) # add the word to the speech list
    return word_list
```

## Code Listing 7.7

### Unique Words, Gettysburg Address

```

# Gettysburg address analysis
# count words, unique words

def make_word_list(a_file):
    """Create a list of words from the file."""
    word_list = []      # list of speech words: initialized to be empty

    for line_str in a_file:          # read file line by line
        line_list = line_str.split() # split each line into a list of words
        for word in line_list:       # get words one at a time from list
            if word != "--":         # if the word is not "--"
                word_list.append(word) # add the word to the speech list
    return word_list

def make_unique(word_list):
    """Create a list of unique words."""
    unique_list = [] # list of unique words: initialized to be empty

    for word in word_list:          # get words one at a time from speech
        if word not in unique_list: # if word is not already in unique list,
            unique_list.append(word) # add word to unique list

    return unique_list

#####
gba_file = open("gettysburg.txt", "r")
speech_list = make_word_list(gba_file)

# print the speech and its lengths
print(speech_list)
print("Speech Length: ", len(speech_list))
print("Unique Length: ", len(make_unique(speech_list)))

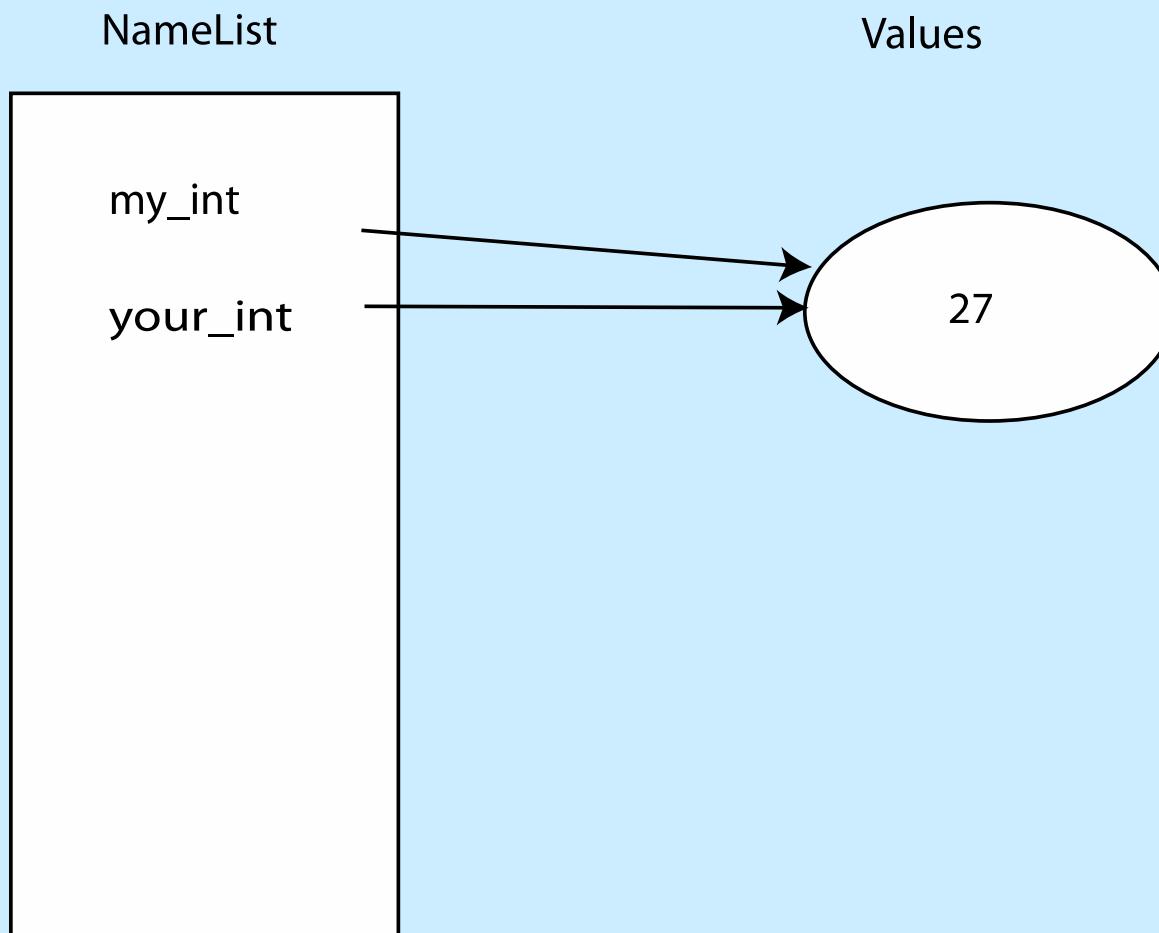
```

# More about mutables

# Reminder, assignment

- Assignment takes an object (the final object after all operations) from the RHS and associates it with a variable on the left hand side
- When you assign one variable to another, you ***share the association*** with the same object

```
my_int = 27  
your_int = my_int
```

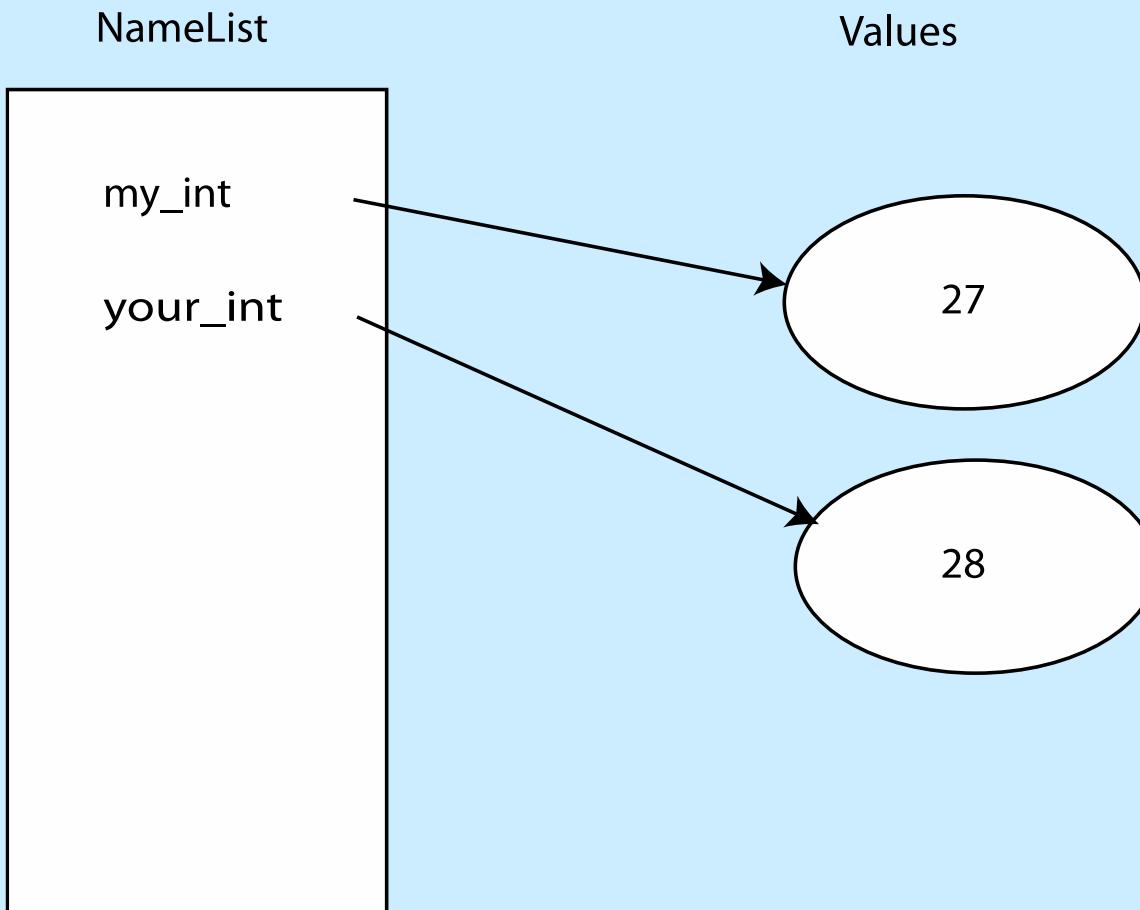


**FIGURE 7.2** Namespace snapshot #1.

# immutables

- Object sharing, two variables associated with the same object, is not a problem since the object cannot be changed
- Any changes that occur generate a ***new*** object.

```
my_int = 27  
your_int = my_int  
your_int = your_int + 1
```

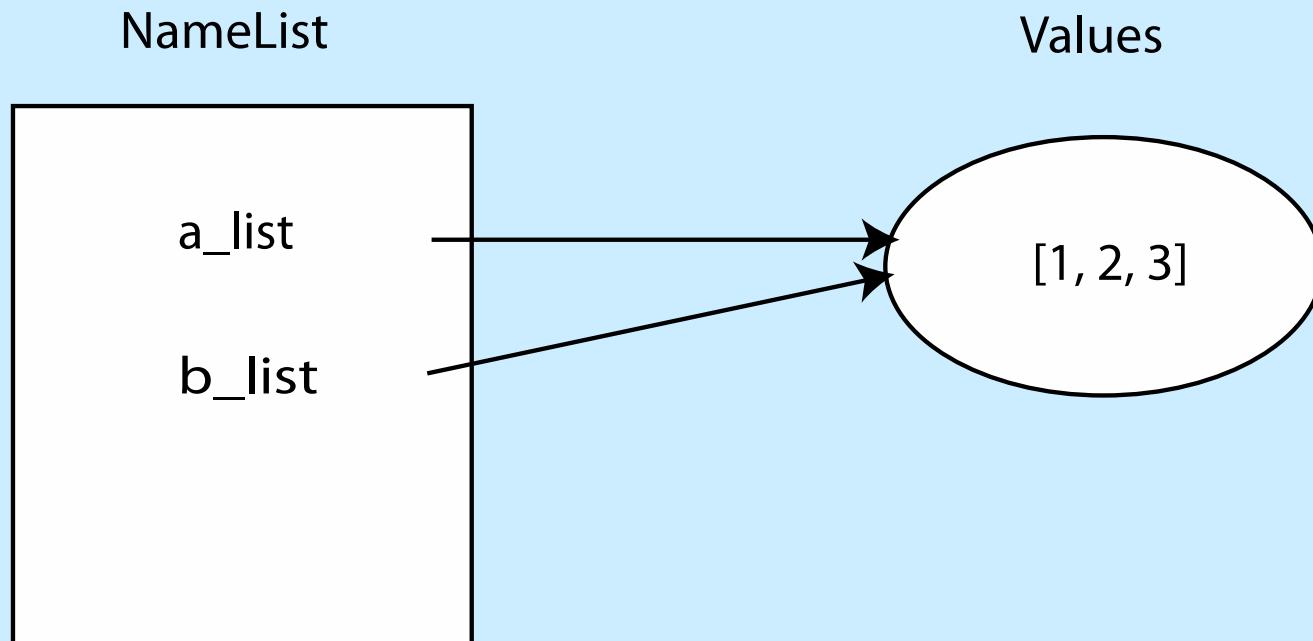


**FIGURE 7.3** Modification of a reference to an immutable object.

# Mutability

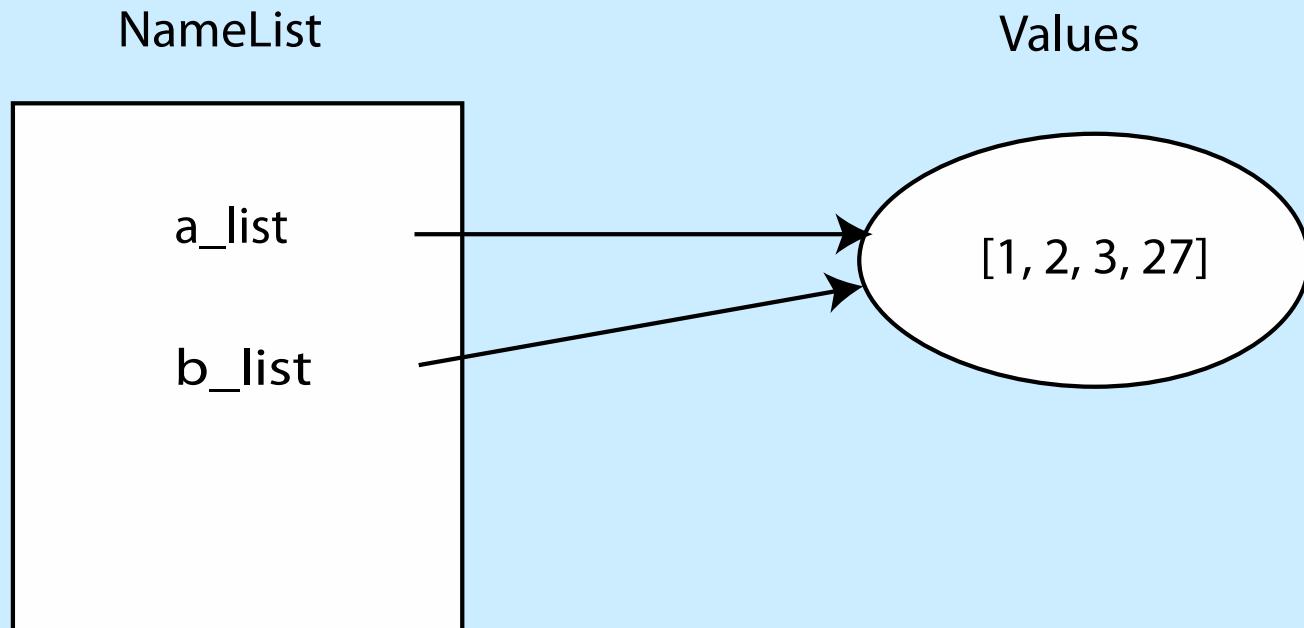
- If two variables associate with the same object, then ***both reflect*** any change to that object

```
a_list = [1,2,3]  
b_list = a_list
```



**FIGURE 7.4** Namespace snapshot after assigning mutable objects.

```
a_list = [1,2,3]  
b_list = a_list  
a_list.append(27)
```



**FIGURE 7.5** Modification of shared, mutable objects.

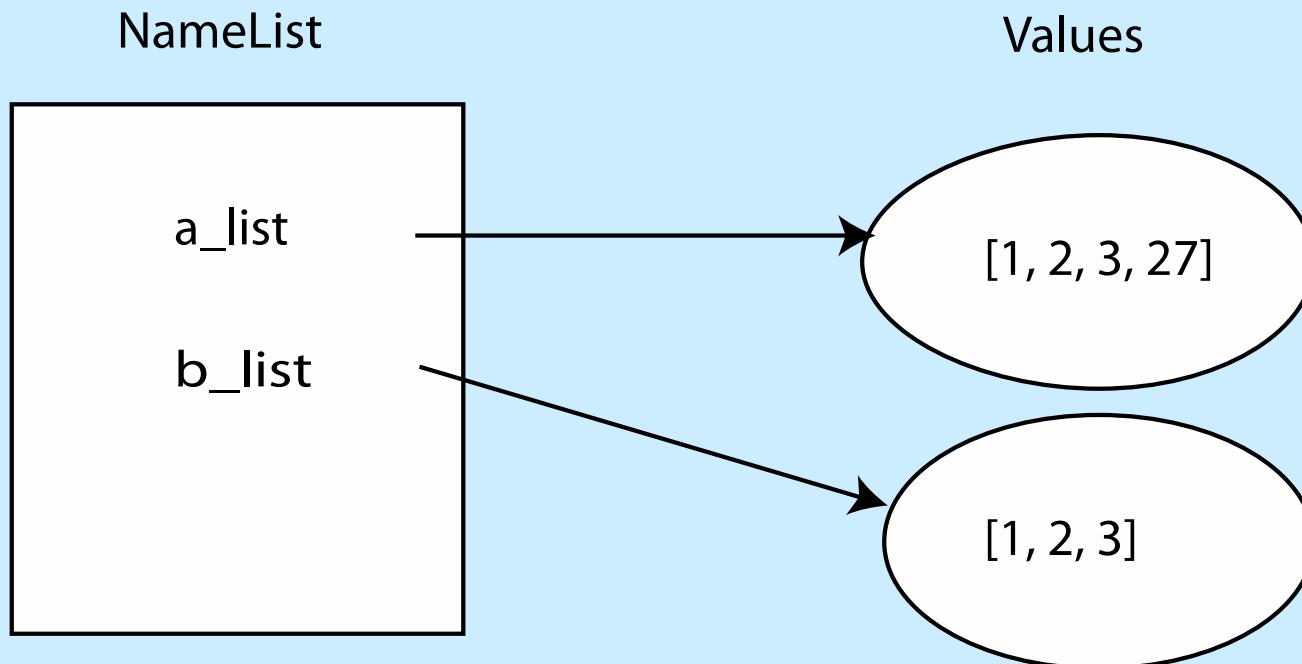
# Copying

If we copy, does that solve the problem?

```
my_list = [1, 2, 3]
```

```
newLst = my_list[:]
```

```
a_list = [1,2,3]
b_list = a_list[:]    # explicitly make a distinct copy
a_list.append(27)
```



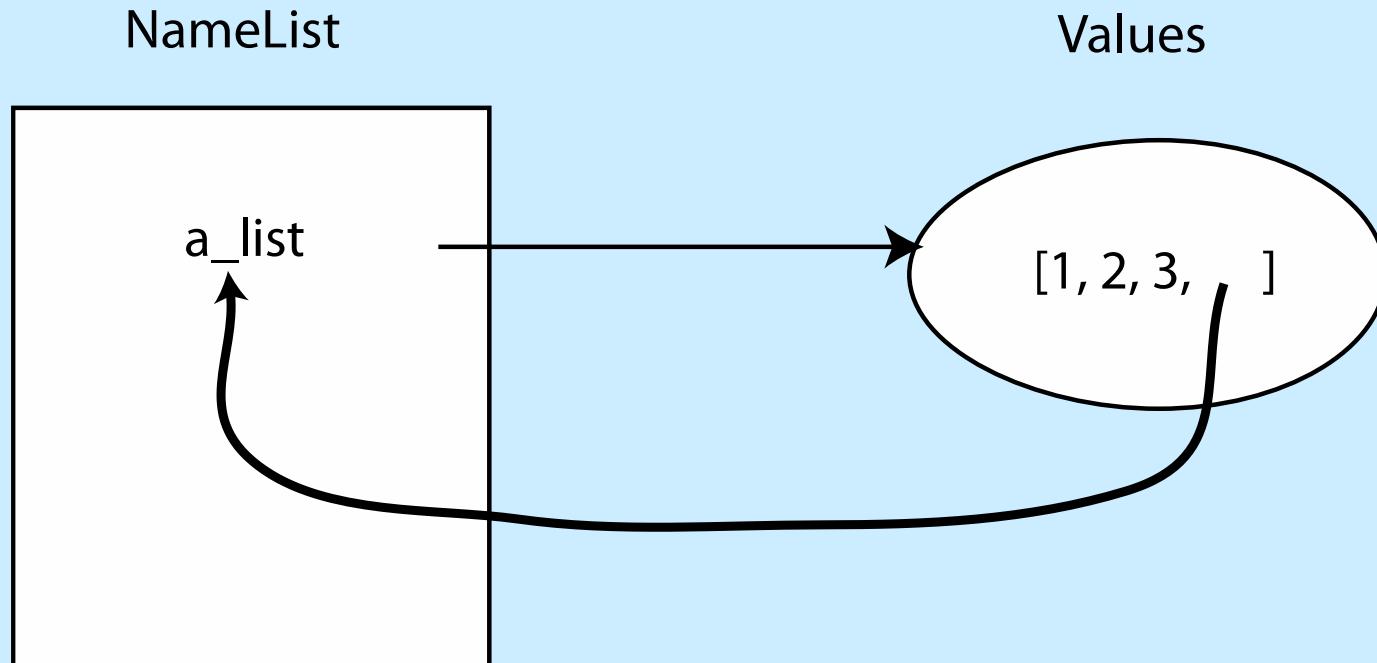
**FIGURE 7.6** Making a distinct copy of a mutable object.

# Sort\_of/depends

The big question is, what gets copied?

- What actually gets copied is the top level reference. If the list has nested lists or uses other associations, the association gets copied. This is termed a ***shallow copy***.

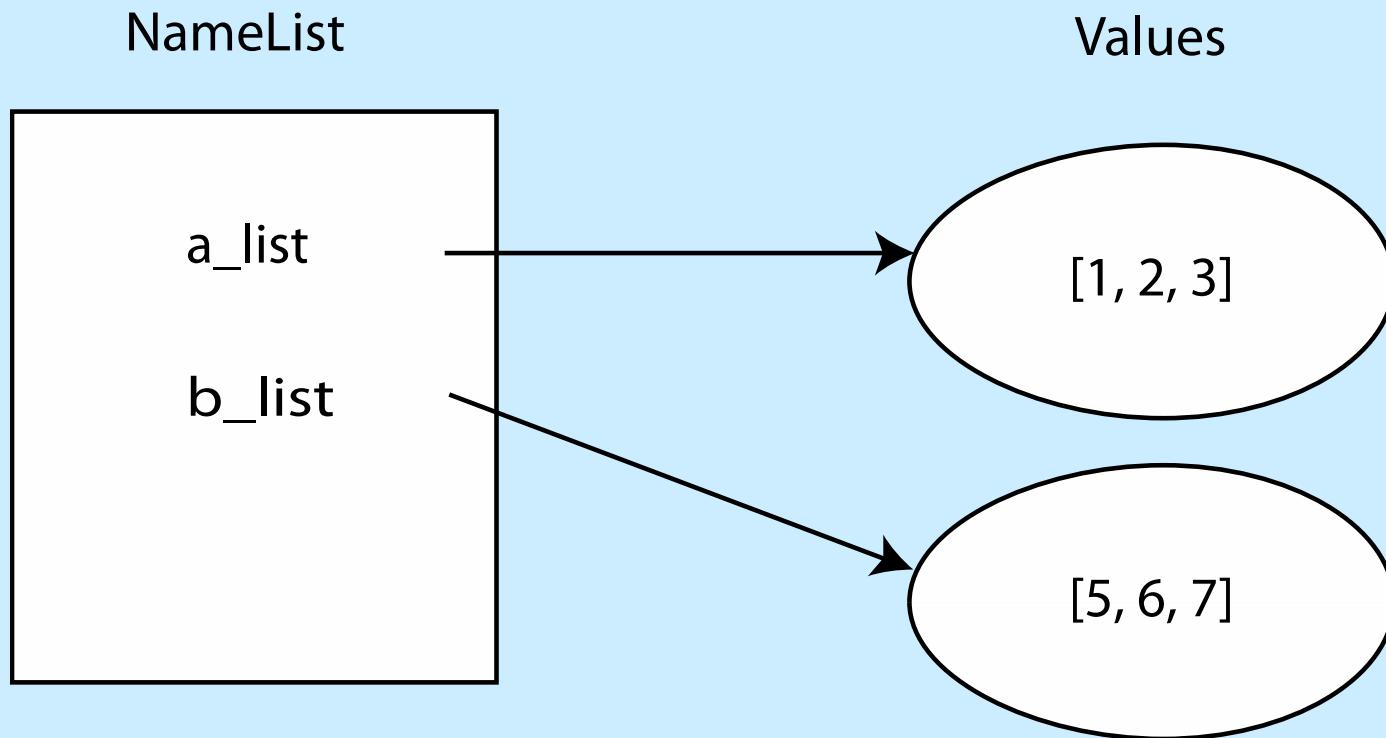
```
a_list = [1,2,3]  
a_list.append(a_list)  
print(a_list)    → [1, 2, 3, [...]]
```



**FIGURE 7.7** Self-referencing.

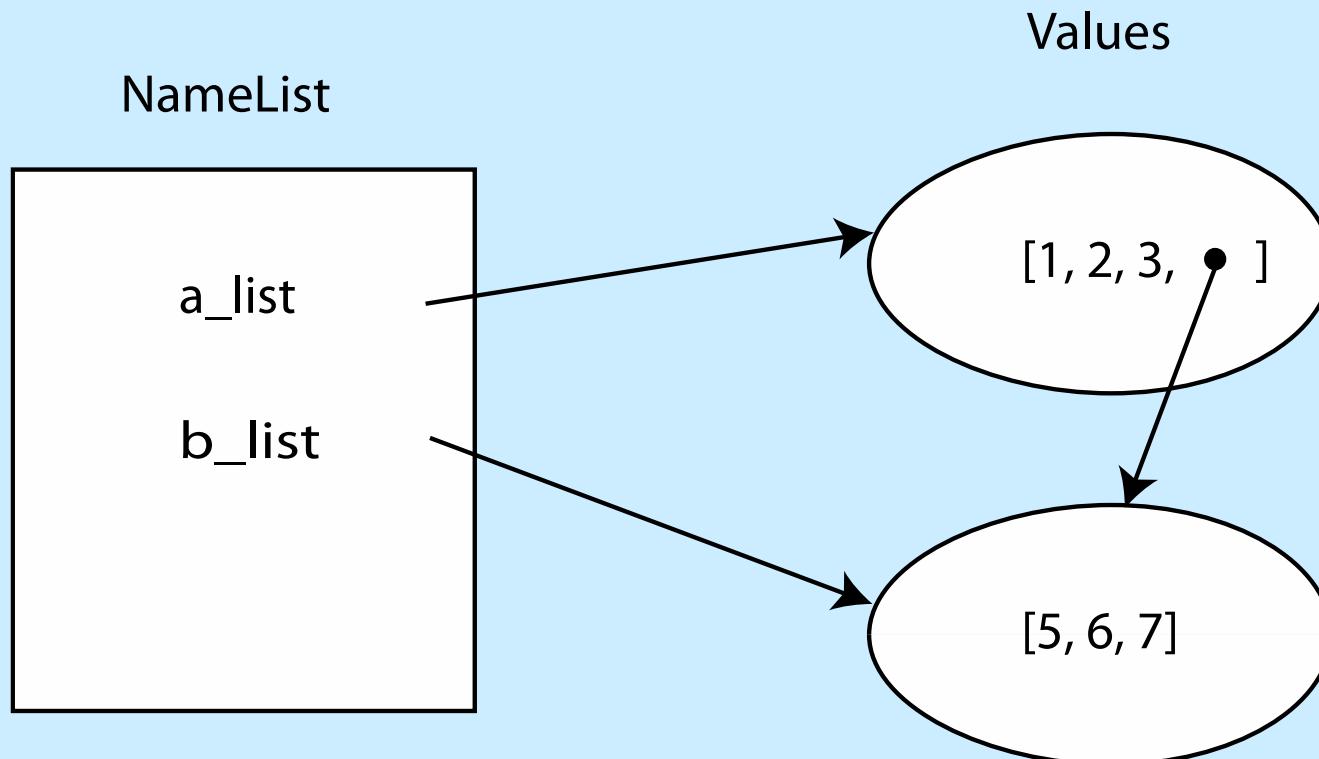
```
a_list = [1,2,3]
```

```
b_list = [5,6,7]
```



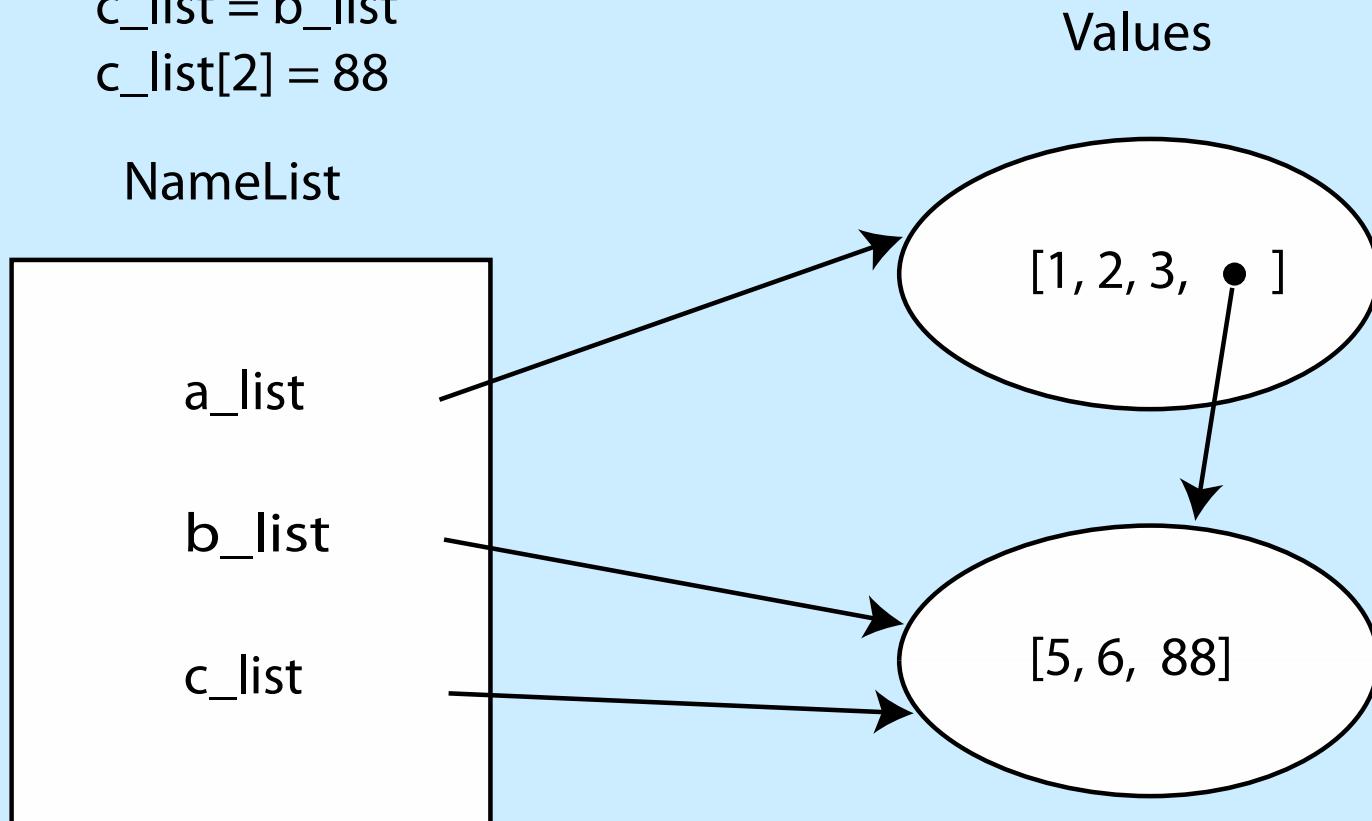
**FIGURE 7.8** Simple lists before append.

```
a_list = [1,2,3]  
b_list = [5,6,7]  
a_list.append(b_list)
```



**FIGURE 7.9** Lists after append.

```
a_list = [1,2,3]
b_list = [5,6,7]
a_list.append(b_list)
c_list = b_list
c_list[2] = 88
```



**FIGURE 7.10** Final state of copying example.

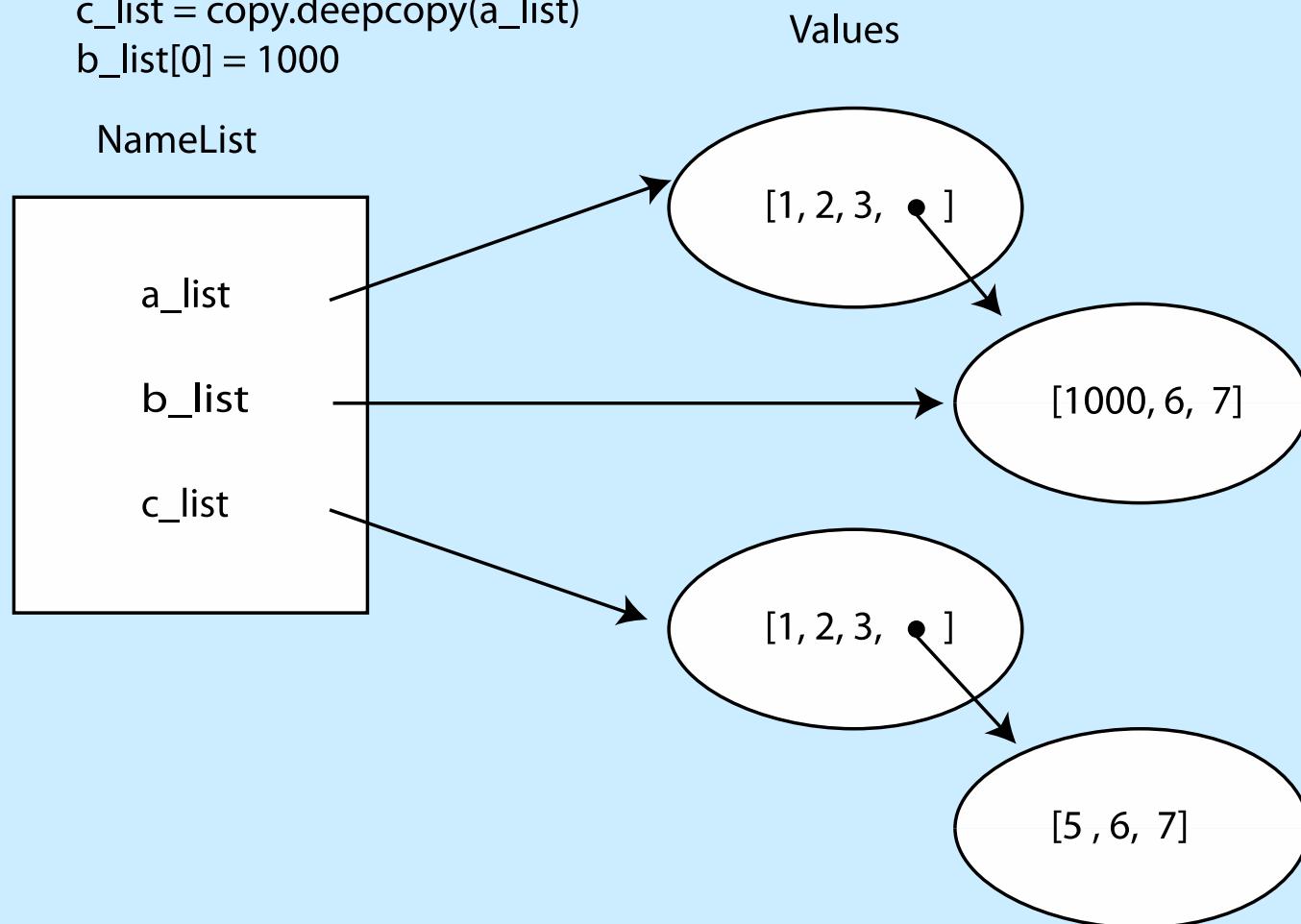
# shallow vs deep

Regular copy, the `[ : ]` approach, only copies the top level reference/association

- if you want a full copy, you can use `deepcopy`

```
>>> a_list = [1, 2, 3]
>>> b_list = [5, 6, 7]
>>> a_list.append(b_list)
>>> import copy
>>> c_list = copy.deepcopy(a_list)
>>> b_list[0]=1000
>>> a_list
[1, 2, 3, [1000, 6, 7]]
>>> c_list
[1, 2, 3, [5, 6, 7]]
>>>
```

```
a_list = [1,2,3]
b_list = [5,6,7]
a_list.append(b_list)
c_list = copy.deepcopy(a_list)
b_list[0] = 1000
```



**FIGURE 7.12** Using the `copy` module for a deep copy.

# Tuples

# Tuples

- Tuples are simply immutable lists
- They are printed with (,)

```
>>> 10,12          # Python creates a tuple  
(10, 12)  
>>> tup = 2,3      # assigning a tuple to a variable  
>>> tup  
(2, 3)  
>>> (1)           # not a tuple, a grouping  
1  
>>> (1,)          # comma makes it a tuple  
(1,)  
>>> x,y = 'a',3.14159    # from on right, multiple assignments  
>>> x  
'a'  
>>> y  
3.14159  
>>> x,y          # create a tuple  
('a', 3.14159)
```

# The question is, Why?

- The real question is, why have an immutable list, a tuple, as a separate type?
- An immutable list gives you a data structure with some integrity, some permanent-ness if you will
- You know you cannot accidentally change one.

# Lists and Tuple

- Everything that works with a list works with a tuple **except** methods that modify the tuple
- Thus indexing, slicing, len, print all work as expected
- However, **none** of the mutable methods work: append, extend, del

# Commas make a tuple

For tuples, you can think of a comma as the operator that makes a tuple, where the ( ) simply acts as a grouping:

```
myTuple = 1,2  # creates (1,2)
myTuple = (1,) # creates (1)
myTuple = (1)  # creates 1 not (1)
myTuple = 1,    # creates (1)
```

# Data Structures in General

# Organization of data

- We have seen strings, lists and tuples so far
- Each is an organization of data that is useful for some things, not as useful for others.

# A good data structure

- Efficient with respect to us (some algorithm)
- Efficient with respect to the amount of space used
- Efficient with respect to the time it takes to perform some operations

# List Comprehensions

# Lists are a big deal!

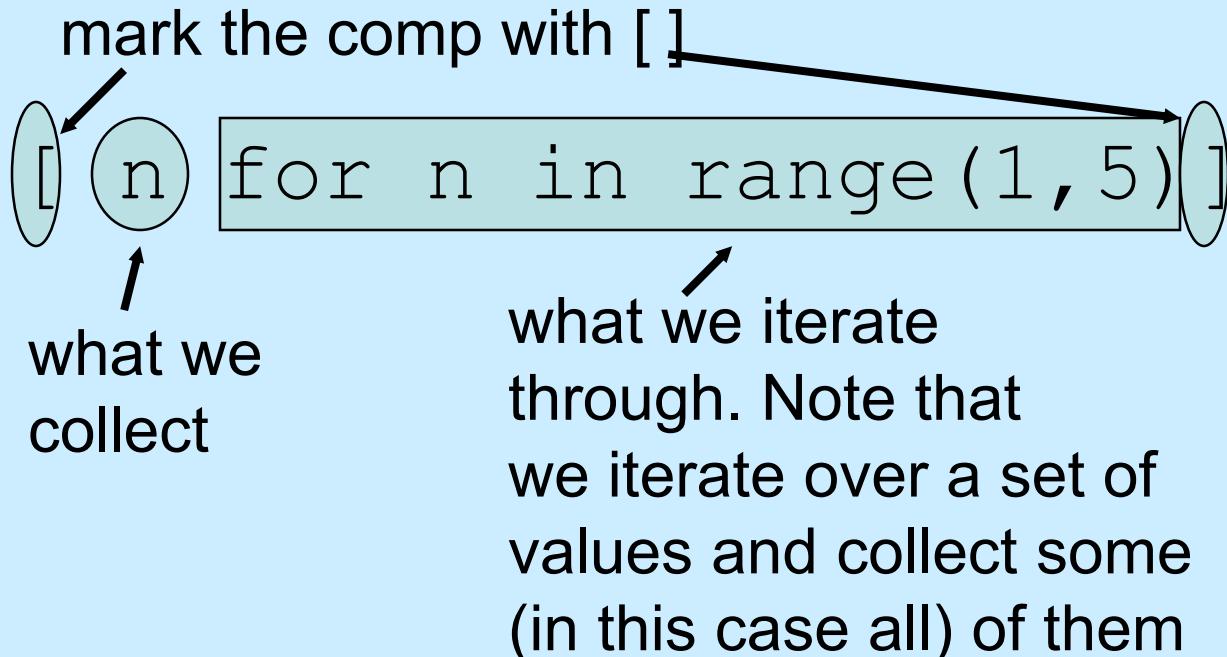
- The use of lists in Python is a major part of its power
- Lists are very useful and can be used to accomplish many tasks
- Therefore Python provides some pretty powerful support to make common list tasks easier

# Constructing lists

One way is a "list comprehension"

```
[n for n in range(1, 5)]
```

returns  
[1,2,3,4]



# modifying what we collect

```
[ n**2 for n in range(1, 6) ]
```

returns [1, 4, 9, 16, 25]. Note that we can only change the values we are iterating over, in this case n

# multiple collects

```
[x+y for x in range(1,4) for y in range (1,4)]
```

It is as if we had done the following:

```
my_list = [ ]  
for x in range (1,4):  
    for y in range (1,4):  
        my_list.append(x+y)  
⇒ [2,3,4,3,4,5,4,5,6]
```

# modifying what gets collected

```
[c for c in "Hi There Mom" if c.isupper()]
```

- The `if` part of the comprehensive controls which of the iterated values is collected at the end. Only those values which make the if part true will be collected

⇒ ['H', 'T', 'M']

# Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.



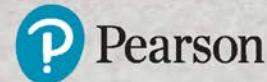
# chapter 7

## More On Functions

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



Pearson

# First cut, scope

# Defining scope

“The set of program statements over which a variable exists, i.e., can be referred to”

- it is about understanding, for any variable, what its associated value is.
- the problem is that multiple namespaces might be involved

# Find the namespace

- For Python, there are potentially multiple namespaces that could be used to determine the object associated with a variable.
- Remember, namespace is an association of name and objects
- We will begin by looking at functions.

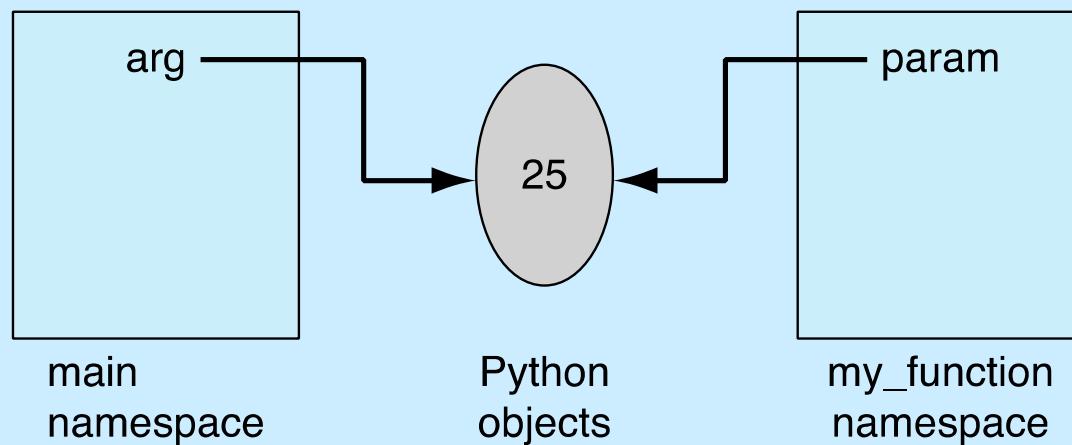
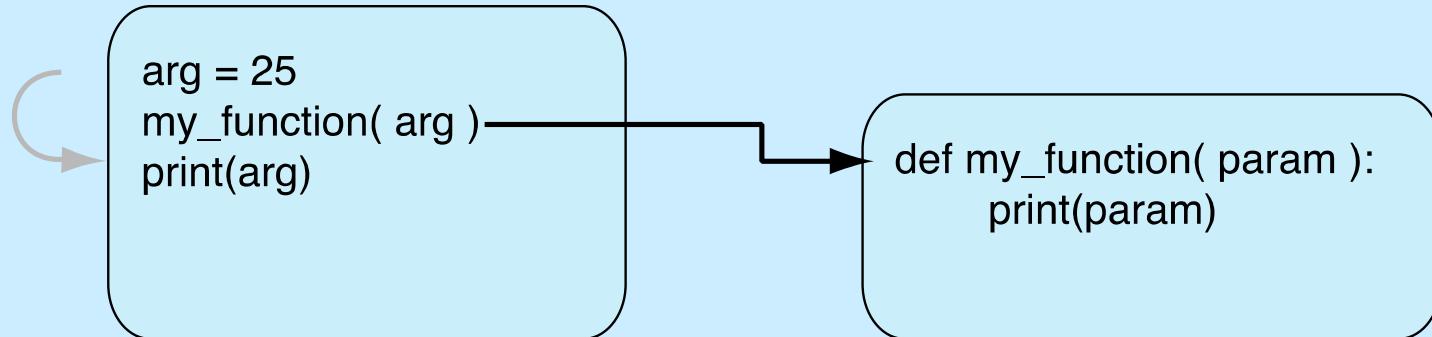
# A function's namespace

- Each function maintains a namespace for names defined *locally within the function.*
- Locally means one of two things:
  - a name assigned within the function
  - an argument received by invocation of the function

# Passing argument to parameter

For each argument in the function invocation, the argument's *associated object* is passed to the corresponding parameter in the function

# Passing immutable objects

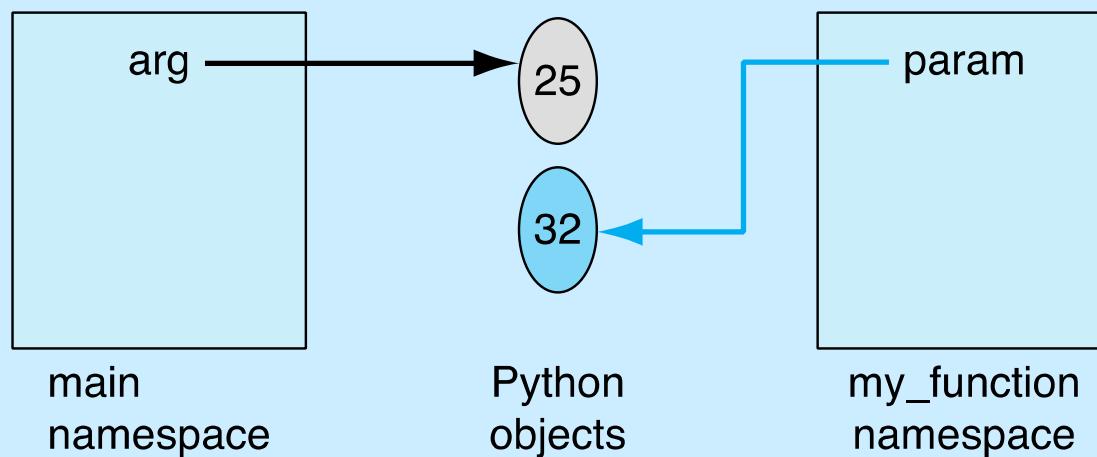
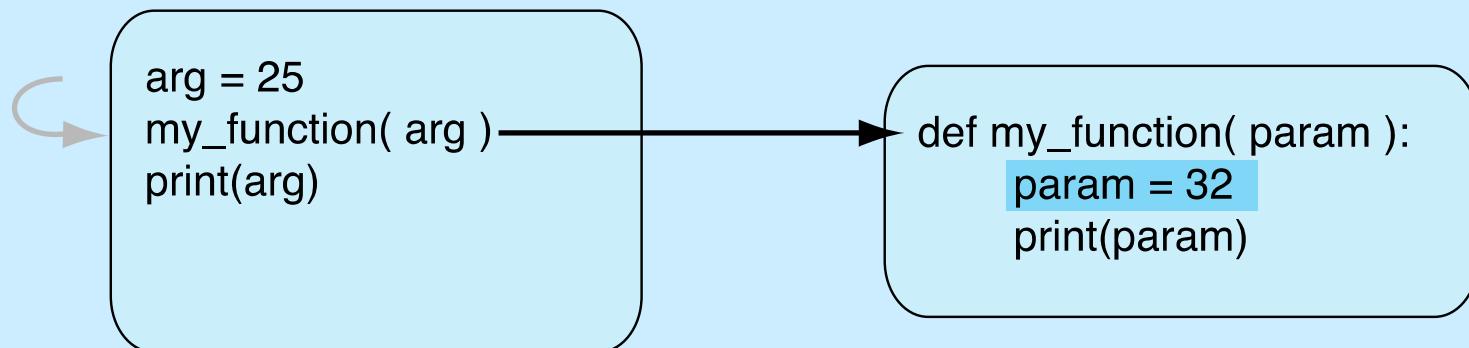


# What does “pass” mean?

- The diagram should make it clear that the parameter name is local to the function namespace
- Passing means that the argument and the parameter, named in two different namespaces, share an association with the same object
- So “passing” means “sharing” in Python

# Assignment changes association

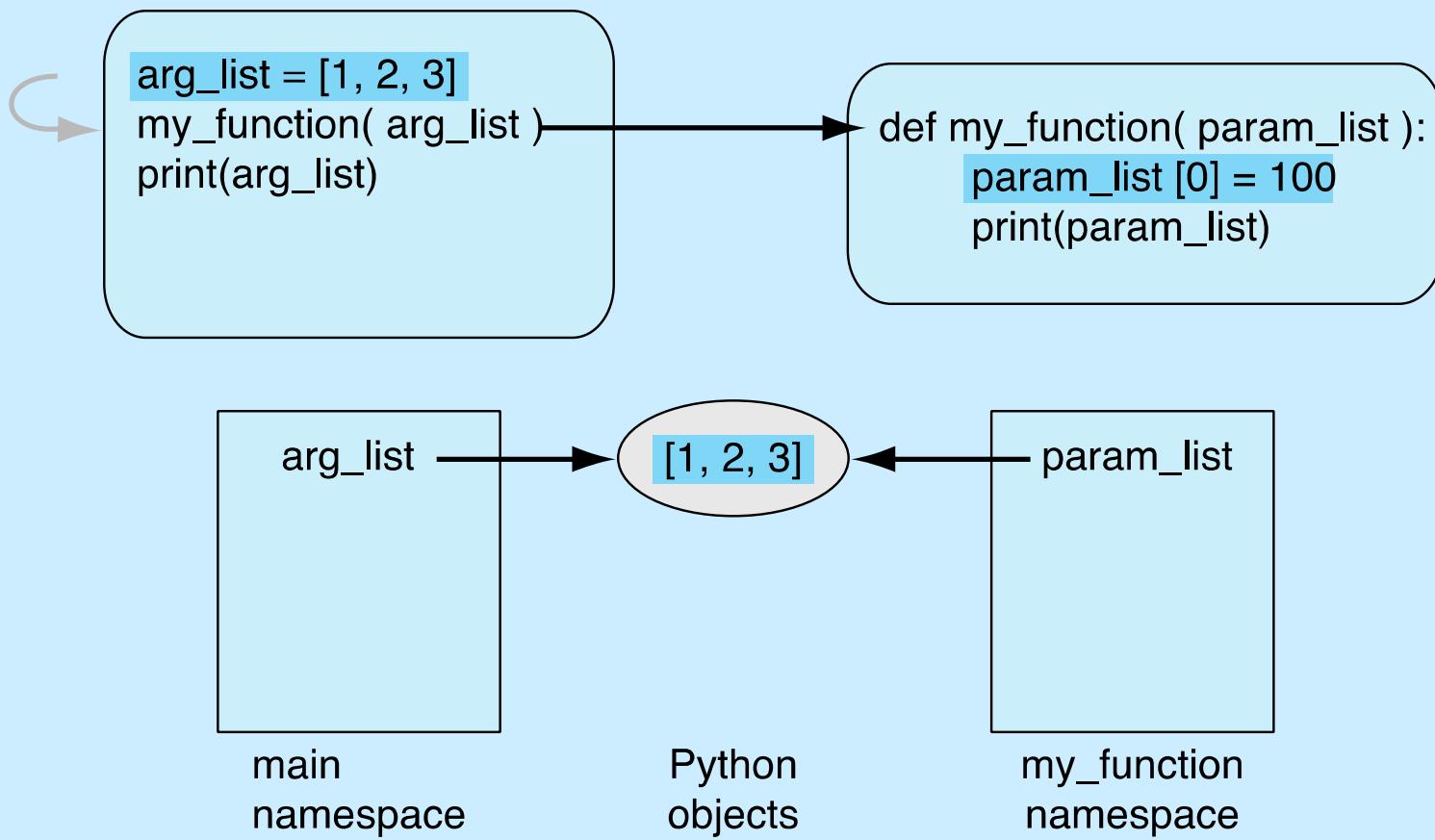
- if a parameter is assigned to a new value, then just like any other assignment, a new association is created
- This assignment does not affect the object associated with the argument, as a new association was made with the parameter



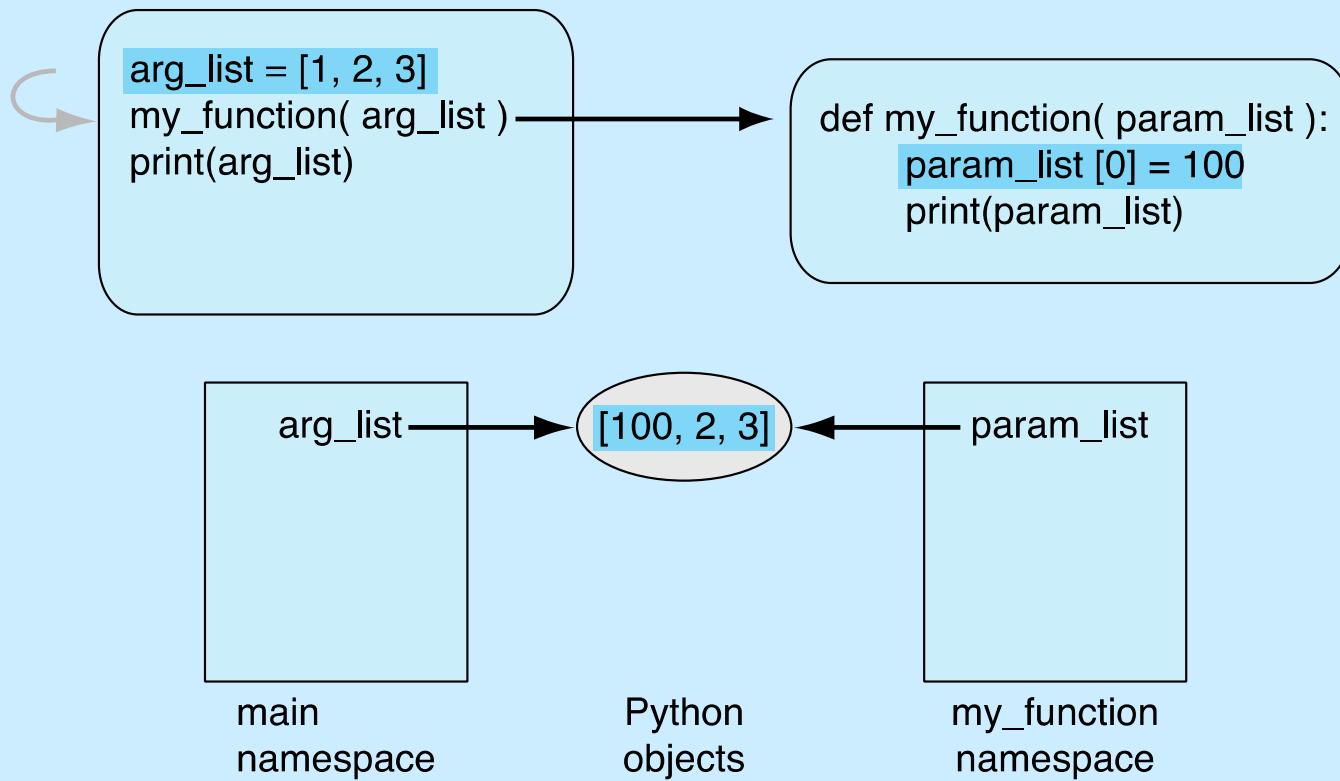
# passing mutable objects

# Sharing mutables

- When passing mutable data structures, it is possible that if the shared object is directly modified, both the parameter and the argument reflect that change
- Note that the operation must be a mutable change, a change of the object. An assignment is not such a change.



**FIGURE 8.3** Function namespace with mutable objects: at function start.



# More on Functions

# Functions return one thing

Functions return one thing, but it can be a ‘chunky’ thing. For example, it can return a tuple

```
>>> def mirror(pair):
    '''reverses first two elements;
       assumes "pair" is as a collection with at least two elements'''
    return pair[1], pair[0]
>>> mirror((2,3))
(3, 2)      # the return was comma separated: implicitly handled as a tuple
>>> first,second = mirror((2,3)) # comma separated works on the left-hand-side also
>>> first
3
>>> second
2
>>> first,second          # reconstruct the tuple
(3, 2)
>>> a_tuple = mirror((2,3)) # if we return and assign to one name, we get a tuple!
>>> a_tuple
(3, 2)
```

# assignment in a function

- if you assign a value in a function, that name becomes part of the local namespace of the function
- it can have some odd effects

# Example

```
def my_fun (param):  
    param.append( 4 )  
    return param  
  
my_list = [1,2,3]  
new_list = my_fun(my_list)  
print(my_list,new_list)
```

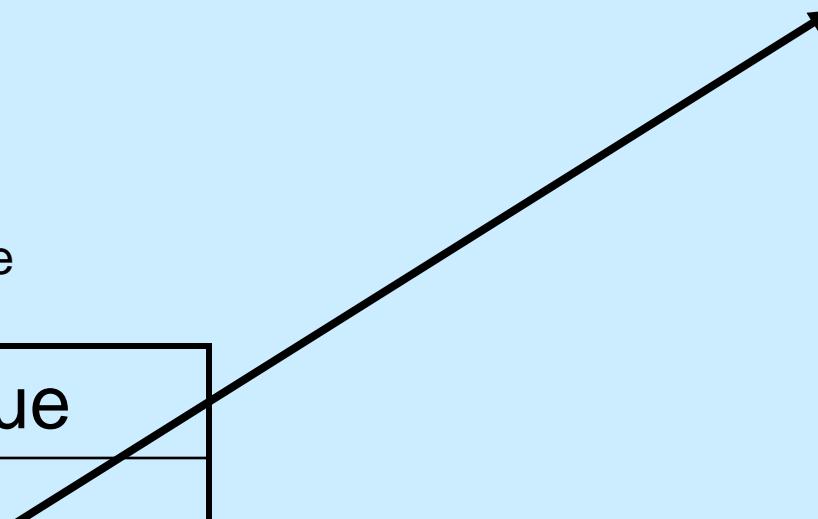
## Main Namespace

| Name    | value |
|---------|-------|
| my_list |       |



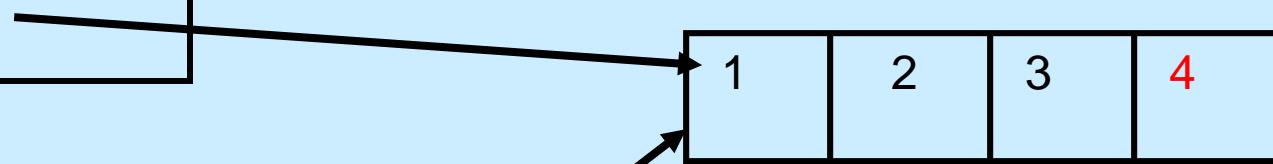
## my\_fun Namespace

| Name  | value |
|-------|-------|
| param |       |



## Main Namespace

| Name    | value |
|---------|-------|
| my_list |       |



## my\_fun Namespace

| Name  | value |
|-------|-------|
| param |       |

# Example

```
def my_fun (param):  
    param=[1,2,3]  
    param.append(4)  
    return param  
  
my_list = [1,2,3]  
new_list = my_fun(my_list)  
print(my_list,new_list)
```

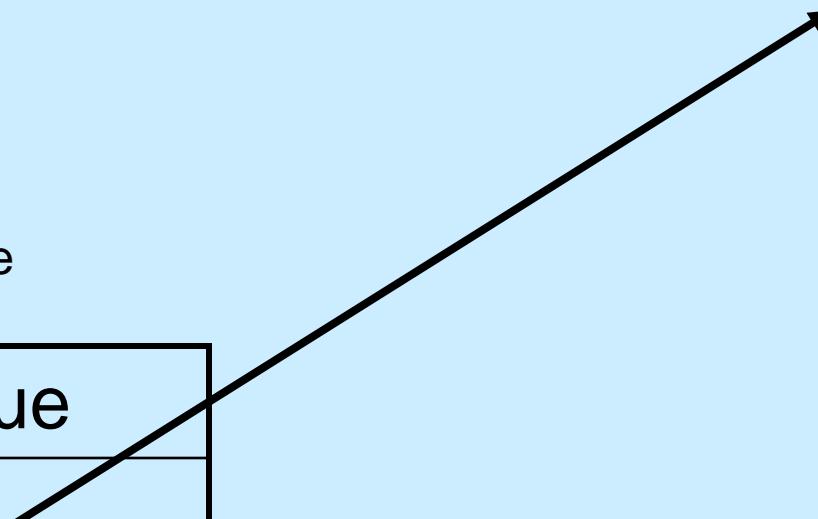
## Main Namespace

| Name    | value |
|---------|-------|
| my_list |       |

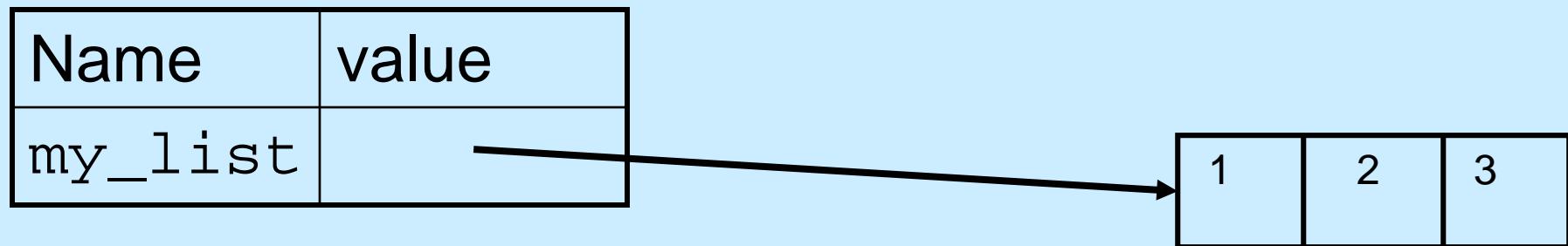


## my\_fun Namespace

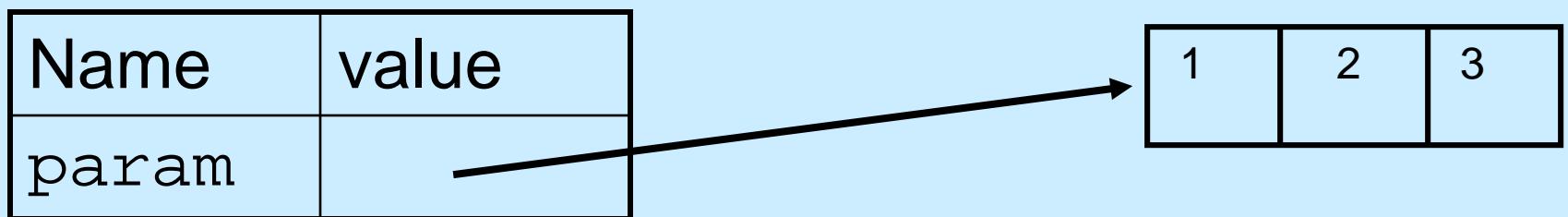
| Name  | value |
|-------|-------|
| param |       |



## Main Namespace

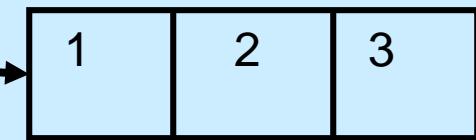


## my\_fun Namespace



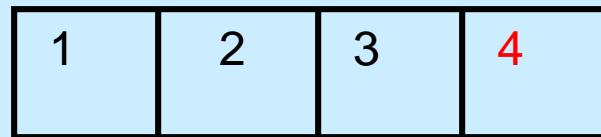
## Main Namespace

| Name    | value |
|---------|-------|
| my_list |       |



## my\_fun Namespace

| Name  | value |
|-------|-------|
| param |       |



# Example

```
def my_fun (param):  
    param=param.append( 4 )  
    return param  
  
my_list = [ 1 , 2 , 3 ]  
new_list = my_fun(my_list)  
print(my_list,new_list)
```

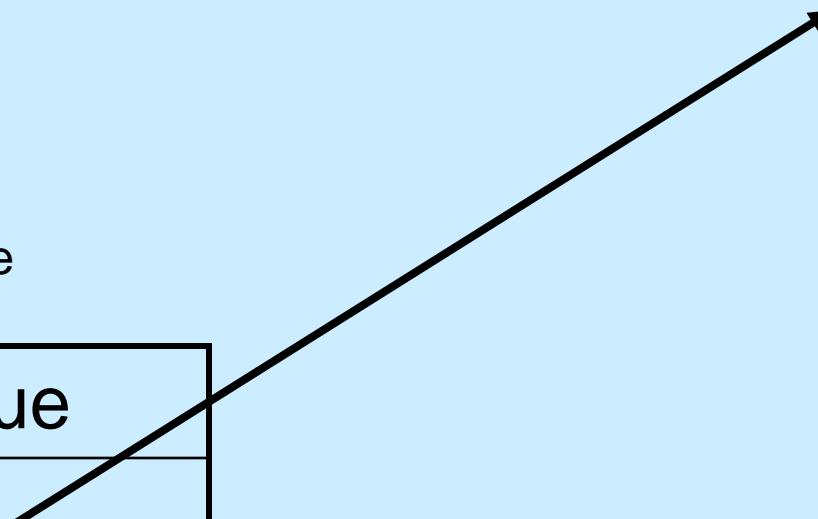
## Main Namespace

| Name    | value |
|---------|-------|
| my_list |       |



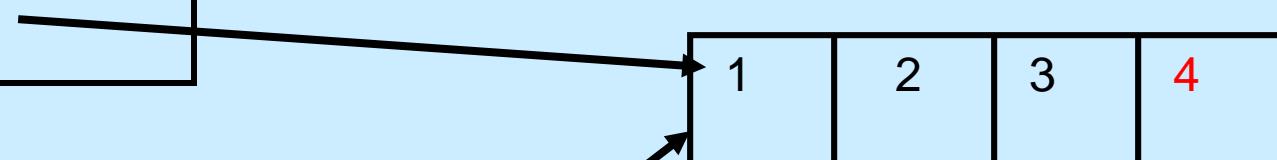
## my\_fun Namespace

| Name  | value |
|-------|-------|
| param |       |



## Main Namespace

| Name    | value |
|---------|-------|
| my_list |       |

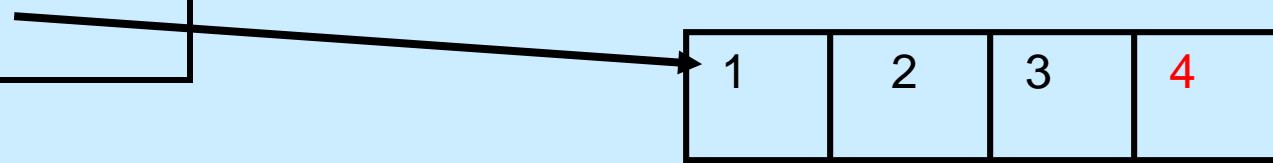


## my\_fun Namespace

| Name  | value |
|-------|-------|
| param |       |

## Main Namespace

| Name    | value |
|---------|-------|
| my_list |       |



## my\_fun Namespace

| Name  | value |
|-------|-------|
| param | None  |

# assignment to a local

- assignment creates a local variable
- changes to a local variable affects only the local context, even if it is a parameter and mutable
- If a variable is assigned locally, cannot reference it before this assignment, even if it exists in main as well

# Default and Named parameters

```
def box(height=10, width=10, depth=10,  
        color= "blue" ):  
    ... do something ...
```

The parameter assignment means two things:

- if the caller does not provide a value, the default is the parameter assigned value
- you can get around the order of parameters by using the name

# Defaults

```
def box(height=10,width=10,length=10):  
    print(height,width,length)  
  
box( )      # prints 10 10 10
```

# Named parameter

```
def box (height=10,width=10,length=10):  
    print(height,width,length)  
  
box(length=25,height=25)  
# prints 25 10 25  
  
box(15,15,15)  # prints 15 15 15
```

# Name use works in general case

```
def my_fun(a,b):  
    print(a,b)
```

```
my_fun(1,2)
```

# prints 1 2

```
my_fun(b=1,a=2)
```

# prints 2 1

# Default args and mutables

- One of the problem with default args occurs with mutables. This is because:
  - the default value is created once, when the function is defined, and stored in the function name space
  - a mutable can change that value of that default

# weird

```
def fn1 (arg1=[ ] , arg2=27) :  
    arg1.append(arg2)  
    return arg1  
  
my_list = [1,2,3]  
print(fn1(my_list,4)) # [1, 2, 3, 4]  
print(fn1(my_list)) # [1, 2, 3, 4, 27]  
print(fn1( )) # [27]  
print(fn1( )) # [27, 27]
```

arg1 is either assigned to the passed arg or to the function default for the arg

fn1 Namespace

| Name | Value |
|------|-------|
| arg1 |       |
| arg2 |       |

```
graph LR; A[ ] --> B[ ]; C[ ] --> D[27]
```

Now the function default, a mutable, is updated and will remain so for the next call

fn1 Namespace

| Name | Value |
|------|-------|
| arg1 |       |
| arg2 |       |

```
graph LR; A[arg1] --> B[27]; C[arg2] --> D[27]
```

# Functions as objects and docstrings

# Functions are objects too!

- Functions are objects, just like anything else in Python.
- As such, they have attributes:
  - `__name__` : function name
  - `__str__` : string function
  - `__dict__` : function namespace
  - `__doc__` : docstring

# function annotations

You can associate strings of information, ignored by Python, with a parameter

- to be used by the reader or user the colon ":" indicates the parameter annotation
- the "->" the annotation is associated with the return value
- stored in dictionary  
`name_fn.__annotations__`

```
def my_func (param1 : int, param2 : float) -> None :
    print('Result is:', param1 + param2)

>>> my_func(1, 2.0)
Result is: 3.0
>>> my_func(1, 2)
Result is: 3
>>> my_func('a', 'b')
Result is: ab
>>>

>>> my_func.__annotations__
{'return': None, 'param2': <class 'float'>, 'param1': <class 'int'>}
>>>
```

# Docstring

- If the first item after the def is a string, then that string is specially stored as the docstring of the function
- This string describes the function and is what is shown if you do a help on a function
- Usually triple quoted since it is multilined

# Can ask for docstring

- Every object (function, whatever) can have a docstring. It is stored as an attribute of the function (the `__doc__` attribute)
- `listMean.__doc__`  
`'Takes a list of integers, returns the average of the list.'`
- Other programs can use the docstring to report to the user (for example, Spyder).

# Determining final grade

The following code shows how you can read in a file of grades. Each line of the file contains five comma-separated fields:

- last name
- first name
- exam1, exam2, final\_exam

print name and final grade

## Code Listing 8.2

### Weighted Grade Function

```
1 def weighted_grade(score_list, weights_tuple=(0.3,0.3,0.4)):  
2     '''Expects 3 elements in score_list. Multiples each grade  
3     by its weight. Returns the sum.'''  
4     grade_float = \  
5         (score_list[0]*weights_tuple[0]) + \  
6         (score_list[1]*weights_tuple[1]) + \  
7         (score_list[2]*weights_tuple[2])  
8     return grade_float
```

## Code Listing 8.3

### parse\_line

```
def parse_line(line_str):
    ''' Expects a line of form last, first, exam1, exam2, final.
returns a tuple containing first+last and list of scores. '''
    field_list = line_str.strip().split(',')
    name_str = field_list[1] + ' ' + field_list[0]
    score_list = []
    # gather the scores, now strings, as a list of ints
    for element in field_list[2:]:
        score_list.append(int(element))
    return name_str, score_list
```

## Code Listing 8.4

### main

```
def main ():
    ''' Get a line_str from the file,
       print the final grade nicely. '''
file_name = input('Open what file: ')
grade_file = open(file_name, 'r')
print('{:>13s} {:>15s}'.format('Name', 'Grade'))
print( '-'*30)
for line_str in grade_file:
    name_str,score_list = parse_line(line_str)
    grade_float = weighted_grade(score_list)
    print('{:>15s} {:.14.2f}'.format(name_str, grade_float))
```

# Arbitrary arguments

- it is also possible to pass an arbitrary number of arguments to a function
- the function simply collects all the arguments (no matter how few or many) into a tuple to be processed by the function
- tuple parameter preceded by a \* (which is not part of the param name, its part of the language)
- positional arguments only

# example

```
def aFunc(fixedParam,*tupleParam):  
    print("fixed =",fixedParam)  
    print("tuple=",tupleParam)  
aFunc(1,2,3,4)  
prints      fixed=1  
            tuple=(2,3,4)  
aFunc(1)  
prints      fixed=1  
            tuple=( )  
aFunc(fixedParam=4)  
prints      fixed=1  
            tuple=( )  
aFunc(tupleParam=(1,2,3),fixedParam=1)  
Error!
```

# Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.

GLOBAL  
EDITION



# chapter 8

## Dictionaries and Sets

PEARSON

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



P Pearson

ALWAYS LEARNING

# More Data Structures

- We have seen the list data structure and what it can be used for
- We will now examine two more advanced data structures, the Set and the Dictionary
- In particular, the dictionary is an important, very useful part of python, as well as generally useful to solve many problems.

# Dictionaries

# What is a dictionary?

- In data structure terms, a dictionary is better termed an *associative array*, *associative list* or a *map*.
- You can think of it as a list of pairs, where the first element of the pair, the **key**, is used to retrieve the second element, the **value**.
- Thus we **map a key to a value**

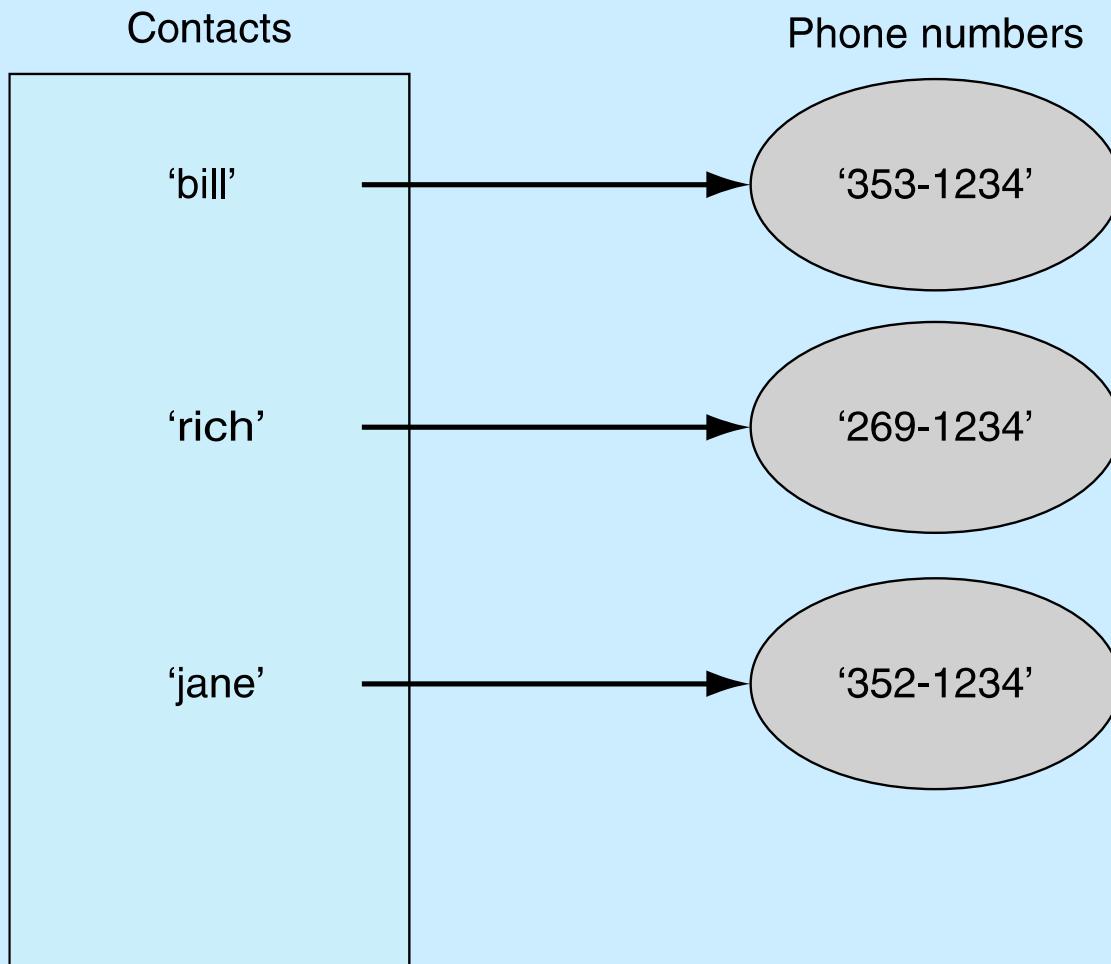
# Key Value pairs

- The key acts as an index to find the associated value.
- Just like a dictionary, you look up a word by its spelling to find the associated definition
- A dictionary can be searched to locate the value associated with a key

# Python Dictionary

- Use the { } marker to create a dictionary
- Use the : marker to indicate key:value pairs

```
contacts= {'bill': '353-1234',
'rich': '269-1234', 'jane': '352-1234'}
print contacts
{'jane': '352-1234',
'bill': '353-1234',
'rich': '369-1234'}
```



**FIGURE 9.1** Phone contact list: names and phone numbers.

# keys and values

- Key must be immutable
  - strings, integers, tuples are fine
  - lists are NOT
- Value can be anything

# collections but not a sequence

- dictionaries are collections but they are not sequences such as lists, strings or tuples
  - there is no order to the elements of a dictionary
  - in fact, the order (for example, when printed) might change as elements are added or deleted.
- So how to access dictionary elements?

# Access dictionary elements

Access requires [ ] , but the *key* is the index!

my\_dict={ }

– an empty dictionary

my\_dict['bill']=25

– added the pair 'bill':25

print(my\_dict['bill'])

– prints 25

# Dictionaries are mutable

- Like lists, dictionaries are a mutable data structure
  - you can change the object via various operations, such as index assignment

```
my_dict = {'bill':3, 'rich':10}  
print(my_dict['bill'])      # prints 2  
my_dict['bill'] = 100  
print(my_dict['bill'])      # prints 100
```

# again, common operators

Like others, dictionaries respond to these

- `len(my_dict)`
  - number of key:value **pairs** in the dictionary
- `element in my_dict`
  - boolean, is element **a key** in the dictionary
- `for key in my_dict:`
  - iterates through the **keys** of a dictionary

# fewer methods

Only 9 methods in total. Here are some

- `key in my_dict`  
does the key exist in the dictionary
- `my_dict.clear()` – empty the dictionary
- `my_dict.update(yourDict)` – for each key in yourDict, updates my\_dict with that key/value pair
- `my_dict.copy` - shallow copy
- `my_dict.pop(key)` – remove key, return value

# Dictionary content methods

- `my_dict.items()` – all the key/value pairs
- `my_dict.keys()` – all the keys
- `my_dict.values()` – all the values

These return what is called a *dictionary view*.

- the order of the views corresponds
- are dynamically updated with changes
- are iterable

# Views are iterable

```
for key in my_dict.keys():
    print key
```

- prints all the keys

```
for key,value in my_dict.items():
    print key,value
```

- prints all the key/value pairs

```
for value in my_dict.values():
    print value
```

- prints all the values

```
my_dict = {'a':2, 3:['x', 'y'], 'joe':'smith'}  
  
>>> dict_value_view = my_dict.values()  
>>> dict_value_view  
dict_values([2, ['x', 'y'], 'smith'])  
>>> type(dict_value_view)  
<class 'dict_values'>  
>>> for val in dict_value_view:  
    print(val)  
  
# view iteration
```

```
2  
['x', 'y']  
smith  
>>> my_dict['new_key'] = 'new_value'  
>>> dict_value_view  
dict_values([2, 'new_value', ['x', 'y'], 'smith'])  
>>> dict_key_view = my_dict.keys()  
dict_keys(['a', 'new_key', 3, 'joe'])  
>>> dict_value_view  
dict_values([2, 'new_value', ['x', 'y'], 'smith']) # same order  
>>>
```

# Frequency of words in list

## 3 ways

# membership test

```
count_dict = {}  
for word in word_list:  
    if word in count_dict:  
        count_dict[word] += 1  
    else:  
        count_dict[word] = 1
```

# exceptions

```
count_dict = {}  
for word in word_list:  
    try:  
        count_dict [word] += 1  
    except KeyError:  
        count_dict [word] = 1
```

# get method

get method returns the value associated with a dict key or a default value provided as second argument. Below, the default is 0

```
count_dict = {}  
for word in word_list:  
    count_dict[word] = count_dict.get(word, 0) + 1
```

# Word Frequency

## Gettysburg Address

### Code Listings 9.2-9.5

# 4 functions

- `add_word(word, word_dict)`. Add word to the dictionary. No return
- `process_line(line, word_dict)`. Process line and identify words. Calls `add_word`. No return.
- `pretty_print(word_dict)`. Nice printing of the dictionary contents. No return
- `main()`. Function to start the program.

# Passing mutables

- Because we are passing a mutable data structure, a dictionary, we do not have to return the dictionary when the function ends
- If all we do is update the dictionary (change the object) then the argument will be associated with the changed object.

```
1 def add_word(word, word_count_dict):
2     '''Update the word frequency: word is the key, frequency is the value.'''
3     if word in word_count_dict:
4         word_count_dict[word] += 1
5     else:
6         word_count_dict[word] = 1
```

```
1 import string
2 def process_line(line, word_count_dict):
3     '''Process the line to get lowercase words to add to the dictionary.'''
4     line = line.strip()
5     word_list = line.split()
6     for word in word_list:
7         # ignore the '--' that is in the file
8         if word != '--':
9             word = word.lower()
10            word = word.strip()
11            # get commas, periods and other punctuation out as well
12            word = word.strip(string.punctuation)
13            add_word(word, word_count_dict)
```

# sorting in pretty\_print

- the `sort` method works on lists, so if we sort we must sort a list
- for complex elements (like a tuple), the sort compares the first element of each complex element:

```
(1, 3) < (2, 1)      # True
```

```
(3, 0) < (1, 2, 3)  # False
```

- a list comprehension (commented out) is the equivalent of the code below it

```
1 def pretty_print(word_count_dict):
2     '''Print nicely from highest to lowest frequency.'''
3     # create a list of tuples, (value, key)
4     # value_key_list = [(val, key) for key, val in d.items()]
5     value_key_list = []
6     for key, val in word_count_dict.items():
7         value_key_list.append((val, key))
8     # sort method sorts on list's first element, the frequency.
9     # Reverse to get biggest first
10    value_key_list.sort(reverse=True)
11    # value_key_list = sorted([(v,k) for k,v in value_key_list.items()],
12    reverse=True)
12    print('{:11s}{:11s}'.format('Word', 'Count'))
13    print('_'*21)
14    for val, key in value_key_list:
15        print('{:12s}  {:<3d}'.format(key, val))
```

```
1 def main () :
2     word_count_dict={}
3     gba_file = open('gettysburg.txt', 'r')
4     for line in gba_file:
5         process_line(line, word_count_dict)
6     print('Length of the dictionary:', len(word_count_dict))
7     pretty_print(word_count_dict)
```

# Periodic Table example

# comma separated values (csv)

- **csv** files are a text format that are used by many applications (especially spreadsheets) to exchange data as text
- row oriented representation where each line is a row, and elements of the row (columns) are separated by a comma
- despite the simplicity, there are variations and we'd like Python to help

# csv module

- `csv.reader` takes an opened file object as an argument and reads one line at a time from that file
- Each line is formatted as a list with the elements (the columns, the comma separated elements) found in the file

# encodings other than UTF-8

- this example uses a csv file encoded with characters other than UTF-8 (our default)
  - in particular, the symbol ± occurs
- can solve by opening the file with the correct encoding, in this case windows-1252

# example

```
>>> import csv
>>> periodic_file = open("Periodic-Table.csv", "r", encoding="windows-1252")
>>> reader = csv.reader(periodic_file)
>>> for row in reader:
    print(row)

# some of the output data
['2', 'He', '18', 'VIII A', '1', 'helium', '4.003', '0', '', '', ...]
['3', 'Li', '1', 'I A', '2', 'lithium', '6.941', '+1', '', '', ...]
['4', 'Be', '2', 'II A', '2', 'beryllium', '9.012', '+2', '', '', ...]
['5', 'B', '13', 'III A', '2', 'boron', '10.81', '+3', '', '', ...]
# etc. etc.
```

# Sets

# Sets, as in Mathematical Sets

- in mathematics, a set is a collection of objects, potentially of many different types
- in a set, no two elements are identical. That is, a set consists of elements each of which is unique compared to the other elements
- there is no order to the elements of a set
- a set with no elements is the empty set

# Creating a set

Set can be created in one of two ways:

- constructor: `set(iterable)` where the argument is iterable

```
my_set = set('abc')  
my_set → {'a', 'b', 'c'}
```

- shortcut: `{ }`, braces where the elements have no colons (to distinguish them from dicts)

```
my_set = {'a', 'b', 'c'}
```

# Diverse elements

- A set can consist of a mixture of different types of elements

```
my_set = { 'a', 1, 3.14159, True }
```

- as long as the single argument can be iterated through, you can make a set of it

# no duplicates

- duplicates are automatically removed

```
my_set = set("aabbcudd")
print(my_set)
→ { 'a', 'c', 'b', 'd' }
```

# example

```
>>> null_set = set()                      # set() creates the empty set
>>> null_set
set()
>>> a_set = {1,2,3,4}                      # no colons means set
>>> a_set
{1, 2, 3, 4}
>>> b_set = {1,1,2,2,2}                    # duplicates are ignored
>>> b_set
{1, 2}
>>> c_set = {'a', 1, 2.5, (5,6)} # different types is OK
>>> c_set
{(5, 6), 1, 2.5, 'a'}
>>> a_set = set("abcd")                   # set constructed from iterable
>>> a_set
{'a', 'c', 'b', 'd'}                      # order not maintained!
```

# common operators

Most data structures respond to these:

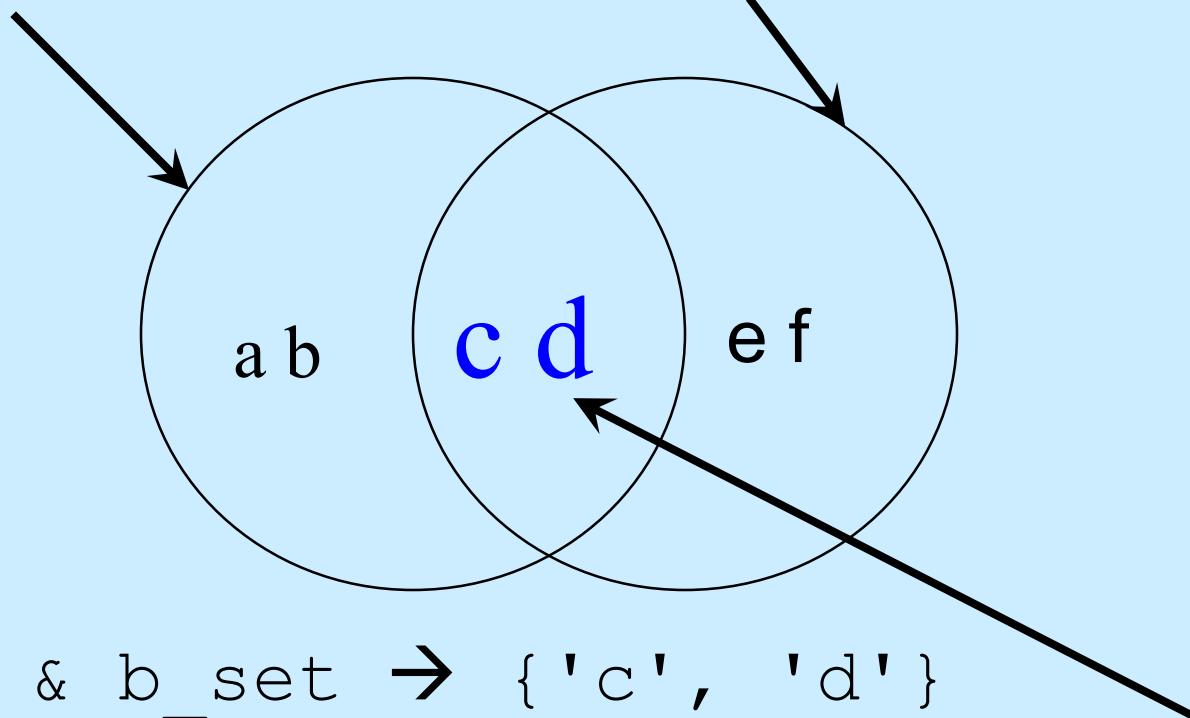
- `len(my_set)`
  - the number of elements in a set
- `element in my_set`
  - boolean indicating whether element is in the set
- `for element in my_set:`
  - iterate through the elements in `my_set`

# Set operators

- The set data structure provides some special operators that correspond to the operators you learned in middle school.
- These are various combinations of set contents
- These operations have both a method name and a shortcut binary operator

# method: intersection, op: &

```
a_set=set("abcd") b_set=set("cdef")
```

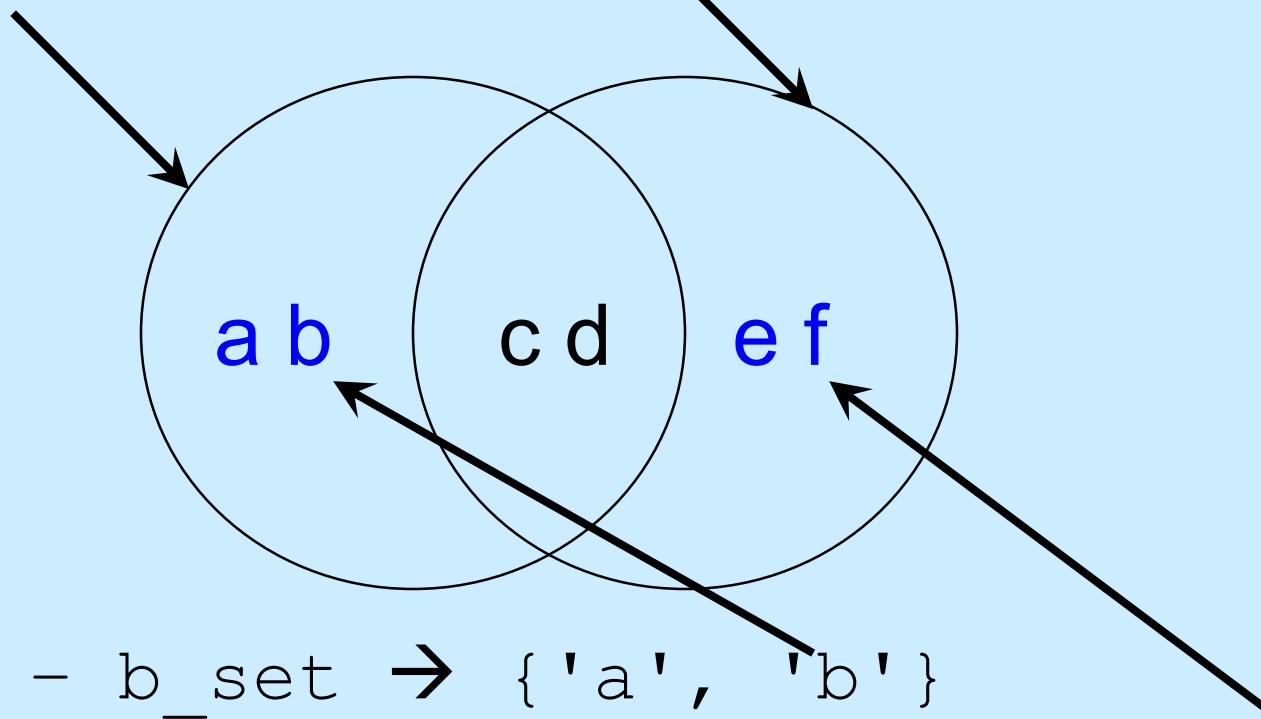


```
a_set & b_set → { 'c', 'd' }
```

```
b_set.intersection(a_set) → { 'c', 'd' }
```

# method:difference op: -

a\_set=set("abcd") b\_set=set("cdef")

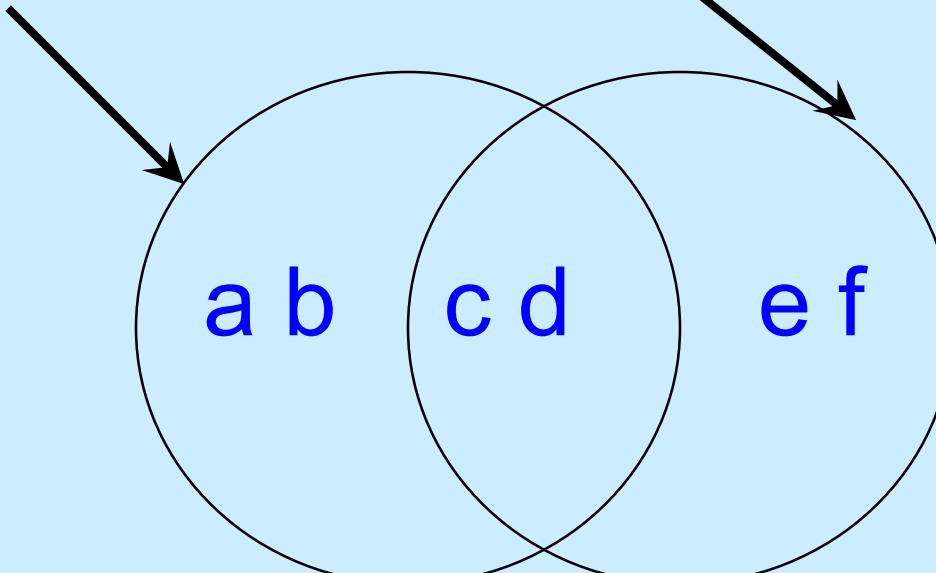


a\_set - b\_set → { 'a', 'b' }

b\_set.difference(a\_set) → { 'e', 'f' }

# method: union, op: |

```
a_set=set("abcd") b_set=set("cdef")
```

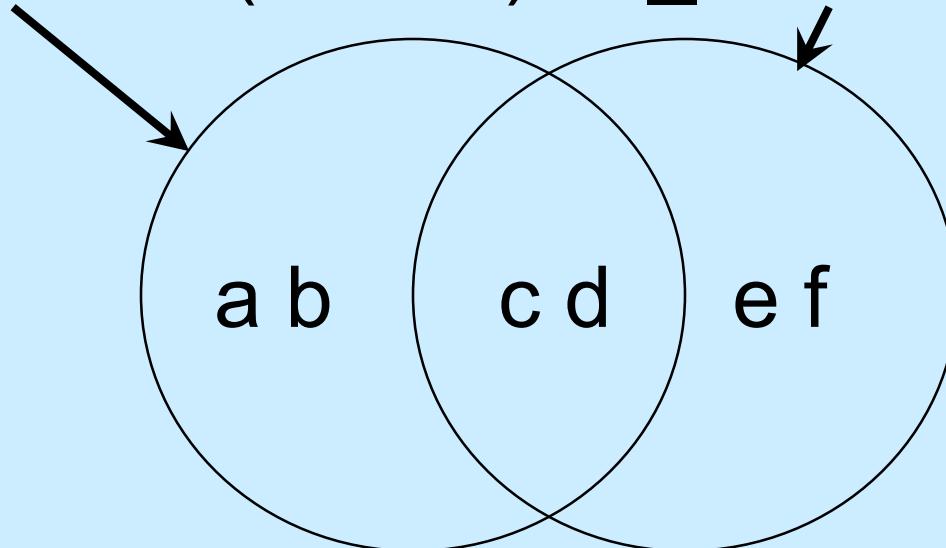


```
a_set | b_set → {'a', 'b', 'c', 'd', 'e', 'f'}  
b_set.union(a_set) → {'a', 'b', 'c', 'd', 'e',  
'f'}
```

# method:symmetric\_difference,

op: ^

a\_set=set("abcd"); b\_set=set("cdef")



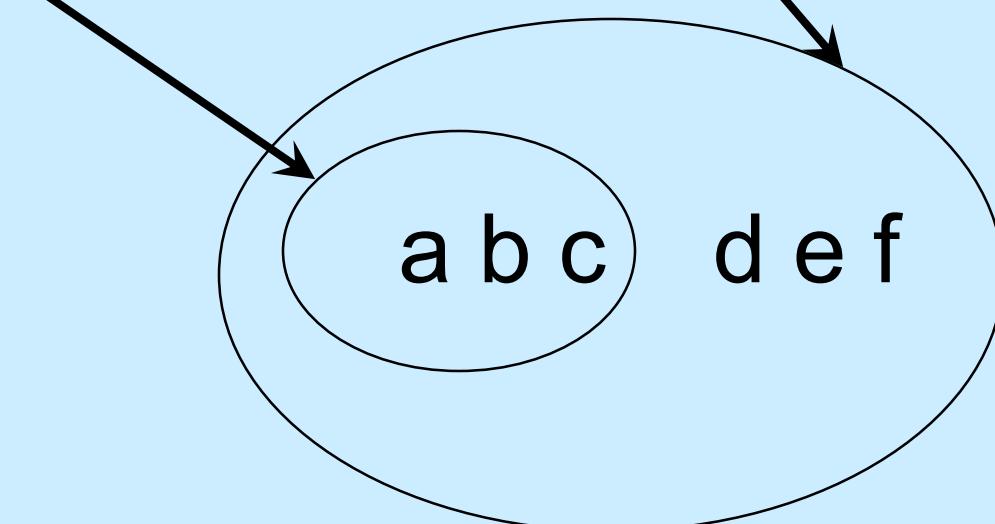
a\_set ^ b\_set → {'a', 'b', 'e', 'f'}

b\_set.symmetric\_difference(a\_set) → {'a', 'b', 'e', 'f'}

method: issubset, op: <=

method: issuperset, op: >=

small\_set=set("abc"); big\_set=set("abcdef")



small\_set <= big\_set → True

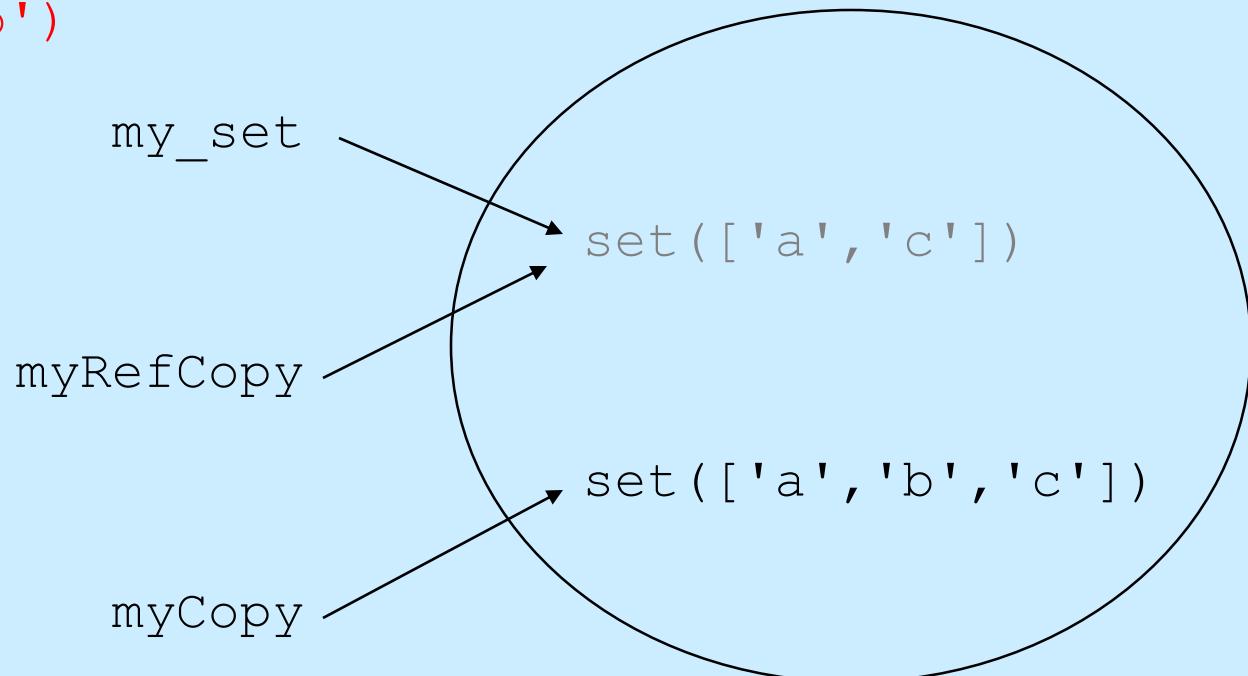
big\_set >= small\_set → True

# Other Set Ops

- `my_set.add("g")`
  - adds to the set, no effect if item is in set already
- `my_set.clear()`
  - emptys the set
- `my_set.remove("g")` versus  
`my_set.discard("g")`
  - remove throws an error if "g" isn't there. discard doesn't care. Both remove "g" from the set
- `my_set.copy()`
  - returns a shallow copy of my\_set

# Copy vs. assignment

```
my_set=set {'a', 'b', 'c'}  
my_copy=my_set.copy()  
my_ref_copy=my_set  
my_set.remove('b')
```



# Common/Unique words

## Code Listings 9.9-9.12

# Common words in Gettysburg Address and Declaration of Independence

- can reuse or only slightly modify much of the code for document frequency
- the overall outline remains much the same
- for clarity, we will ignore any word that has three characters or less (typically stop words)

# 4 functions

- `add_word(word, word_set)`. Add word to the set (instead of dict). No return.
- `process_line(line, word_set)`. Process line and identify words. Calls `add_word`. No return. (no change except for parameters)
- `pretty_print(word_set)`. Nice printing of the various set operations. No return
- `main()`. Function to start the program.

```
1 def add_word(word, word_set):  
2     '''Add the word to the set. No word smaller than length 3.'''  
3     if len(word) > 3:  
4         word_set.add(word)
```

```
1 import string
2 def process_line(line, word_set):
3     '''Process the line to get lowercase words to be added to the set.'''
4     line = line.strip()
5     word_list = line.split()
6     for word in word_list:
7         # ignore the '--' that is in the file
8         if word != '--':
9             word = word.strip()
10            # get commas, periods and other punctuation out as well
11            word = word.strip(string.punctuation)
12            word = word.lower()
13            add_word(word, word_set)
```

# more complicated pretty print

- the `pretty_print` function applies the various set operators to the two resulting sets
- prints, in particular, the intersection in a nice format
- should this have been broken up into two functions??

```

1 def pretty_print(ga_set, doi_set):
2     # print some stats about the two sets
3     print('Count of unique words of length 4 or greater')
4     print('Gettysburg Addr: {}, Decl of Ind: {}\\n'.format(len(ga_set),len(
5         doi_set)))
6     print('{:15s} {:15s}'.format('Operation', 'Count'))
7     print('-'*35)
8     print('{:15s} {:15d}'.format('Union', len(ga_set.union(doi_set))))
9     print('{:15s} {:15d}'.format('Intersection', len(ga_set.intersection(
10        doi_set))))
11    print('{:15s} {:15d}'.format('Sym Diff', len(ga_set.symmetric_difference(
12        doi_set))))
13    print('{:15s} {:15d}'.format('GA-DoI', len(ga_set.difference(doi_set))))
14    print('{:15s} {:15d}'.format('DoI-GA', len(doi_set.difference(ga_set))))
15
16
17    # list the intersection words, 5 to a line, alphabetical order
18    intersection_set = ga_set.intersection(doi_set)
19    word_list = list(intersection_set)
20    word_list.sort()
21    print('\\n Common words to both')
22    print('*'*20)
23    count = 0
24    for w in word_list:
25        if count % 5 == 0:
26            print()
27        print('{:13s}'.format(w), end=' ')
28        count += 1

```

# More on Scope

# OK, what is a namespace

- We've had this discussion, but let's review
- A namespace is an association of a name and a value
- It looks like a dictionary, and for the most part it is (at least for modules and classes)

# Scope

- What namespace you might be using is part of identifying the scope of the variables and function you are using
- by "scope", we mean the context, the part of the code, where we can make a reference to a variable or function

# Multiple scopes

- Often, there can be multiple scopes that are candidates for determining a reference.
- Knowing which one is the right one (or more importantly, knowing the order of scope) is important

# Two kinds

- ***Unqualified namespaces.*** This is what we have pretty much seen so far. Functions, assignments etc.
- ***Qualified namespaces.*** This is modules and classes (we'll talk more about this one later in the classes section)

# Unqualified

- this is the standard assignment and def we have seen so far
- Determining the scope of a reference identifies what its true 'value' is

# unqualified follow the LEGB rule

- *local*, inside the function in which it was defined
- if not there, ***enclosing/encomposing***. Is it defined in an enclosing function
- if not there, is it defined in the ***global*** namespace
- finally, check the ***built-in***, defined as part of the special builtin scope
- else ERROR

## Code Listing 9.13

# `locals()` function

Returns a dictionary of the current (presently in play) local namespace. Useful for looking at what is defined where.

# function local values

- if a reference is assigned in a function, then that reference is only available within that function
- if a reference with the same name is provided outside the function, the reference is reassigned

```
global_X = 27

def my_function(param1=123, param2='hi mom') :
    local_X = 654.321
    print('\n==== local namespace ===')
    for key, val in locals().items():
        print('key:{} , object:{}' .format(key, str(val)))
    print('local_X:', local_X)
    print('global_X:', global_X)

my_function()
```

```
==== local namespace ===
key:local_X, object:654.321
key:param1, object:123
key:param2, object:hi mom
local_X: 654.321
global_X: 27
```

**global is still found because of the sequence of namespace search**

## Code Listing 9.14

# globals() function

Like the `locals()` function, the `globals()` function will return as a dictionary the values in the global namespace

```
import math
global_X = 27

def my_function(param1=123, param2='hi mom'):
    local_X = 654.321
    print('\n==== local namespace ===')
    for key, val in locals().items():
        print('key: {}, object: {}'.format(key, str(val)))
    print('local_X:', local_X)
    print('global_X:', global_X)

my_function()

key, val = 0, 0 # add to the global namespace. Used below
print('\n--- global namespace ---')

for key, val in globals().items():
    print('key: {:15s} object: {}'.format(key, str(val)))

print('-----')
#print 'Local_X:', local_X
print('Global_X:', global_X)
print('Math.pi:', math.pi)
print('Pi:', pi)
```

```
== local namespace ==
key: local_X, object: 654.321
key: param1, object: 123
key: param2, object: hi mom
local_X: 654.321
global_X: 27

--- global namespace ---
key: my_function      object: <function my_function at 0xe15a30>
key: __builtins__      object: <module '__builtin__' (built-in)>
key: __package__       object: None
key: global_X          object: 27
key: __name__           object: __main__
key: __doc__             object: None
key: math                object: <module 'math' from '/Library/Frameworks/Python.
framework/Versions/3.2/lib/python3.2/lib-dynload/math.so'>
```

```
-----
Global_X: 27
Math.pi: 3.14159265359
Pi:
```

```
Traceback (most recent call last):
  File "/Volumes/Admin/Book/chapterDictionaries/localsAndGlobals.py", line 22, in
<module>
    print('Pi:',pi)
NameError: name 'pi' is not defined
```

# Global Assignment Rule

A quirk of Python.

If an assignment occurs ***anywhere*** in the suite of a function, Python adds that variable to the local namespace

- means that, even if the variable is assigned later in the suite, the variable is still local

## Code Listing 9.15

```
my_var = 27

def my_function(param1=123, param2='Python'):
    for key, val in locals().items():
        print('key {}: {}'.format(key, str(val)))
    my_var = my_var + 1      # causes an error!

my_function(123456, 765432.0)
```

```
key param2: 765432.0
key param1: 123456
Traceback (most recent call last):
  File "localAssignment1.py", line 9, in <module>
    my_function(123456, 765432.0)
  File "localAssignment1.py", line 7, in my_function
    my_var = my_var + 1      # causes an error!
UnboundLocalError: local variable 'my_var' referenced before assignment
```

my\_var is local (is in the local namespace)  
because it is assigned in the suite

# the global statement

You can tell Python that you want the object associated with the global, not local namespace, using the `global` statement

- avoids the local assignment rule
- should be used carefully as it is an override of normal behavior

## Code Listing 9.16

```
my_var = 27

def my_function(param1=123, param2='Python'):
    for key, val in locals().items():
        print('key {}: {}'.format(key, str(val)))
    my_var = my_var + 1      # causes an error!
```

```
def better_function(param1=123, param2='Python'):
    global my_var
    for key, val in locals().items():
        print('key {}: {}'.format(key, str(val)))
    my_var = my_var + 1
    print('my_var:', my_var)
```

```
# my_function(123456, 765432.0)
better_function()
```

```
key param2: Python
key param1: 123
my_var: 28
```

# my\_var is not in the local namespace

# Builtin

- This is just the standard library of Python.
- To see what is there, look at

```
import __builtin__  
dir(__builtin__)
```

# Enclosed

Functions which define other functions in a function suite are **enclosed**, defined only in the enclosing function

- the inner/enclosed function is then part of the local namespace of the outer/enclosing function
- remember, a function is an object too!

## Code Listing 9.18

```

global_var = 27

def outer_function(param_outer = 123):
    outer_var = global_var + param_outer

    def inner_function(param_inner = 0):
        # get inner, enclosed and global
        inner_var = param_inner + outer_var + global_var

        # print inner namespace
        print('\n--- inner local namespace ---')
        for key, val in locals().items():
            print('{}:{}' .format(key, str(val)))
        return inner_var

    result = inner_function(outer_var)
    # print outer namespace
    print('\n--- outer local namespace ---')
    for key, val in locals().items():
        print('{}:{}' .format(key, str(val)))
    return result

result = outer_function(7)
print('\n--- result ---')
print('Result:', result)

```

```
--- inner local namespace ---
outer_var:34
inner_var:95
param_Inner:34

--- outer local namespace ---
outer_var:34
param_Outer:7
result:95
inner_function:<function inner_function at 0xe2ba30>

--- result ---
Result: 95
```

# Building dictionaries faster

- `zip` creates pairs from two parallel lists
  - `zip("abc", [1, 2, 3])` yields  
`[('a', 1), ('b', 2), ('c', 3)]`
- That's good for building dictionaries. We call the `dict` function which takes a list of pairs to make a dictionary
  - `dict(zip("abc", [1, 2, 3]))` yields  
`{'a': 1, 'c': 3, 'b': 2}`

# dict and set comprehensions

Like list comprehensions, you can write shortcuts that generate either a dictionary or a set, with the same control you had with list comprehensions

- both are enclosed with { } (remember, list comprehensions were in [ ])
- difference is if the collected item is a : separated pair or not

# dict comprehension

```
>>> a_dict = {k:v for k,v in enumerate('abcdefg')}
```

```
>>> a_dict
```

```
{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g'}
```

```
>>> b_dict = {v:k for k,v in a_dict.items()} # reverse key-value pairs
```

```
>>> b_dict
```

```
{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'g': 6, 'f': 5}
```

```
>>> sorted(b_dict)      # only sorts keys
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
>>> b_list = [(v,k) for v,k in b_dict.items()]      # create list
```

```
>>> sorted(b_list)                      # then sort
```

```
[('a', 0), ('b', 1), ('c', 2), ('d', 3), ('e', 4), ('f', 5), ('g', 6)]
```

# set comprehension

```
>>> a_set = {ch for ch in 'to be or not to be'}
```

```
>>> a_set
```

```
{' ', 'b', 'e', 'o', 'n', 'r', 't'} # set of unique characters
```

```
>>> sorted(a_set)
```

```
[' ', 'b', 'e', 'n', 'o', 'r', 't']
```

# Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.



# chapter 9

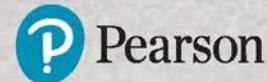
## Files and Exceptions II

PEARSON

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



Pearson

ALWAYS LEARNING

# What we already know

- Files are bytes on disk. Two types, text and binary (we are working with text)
- `open` creates a connection between the disk contents and the program
- different modes of opening a file, '`r`', '`w`', '`a`'
- files might have different encodings (default is `utf_8`)

# more of what we know

- all access, reading or writing, to a text file is by the use of strings
- iteration via a `for` loop gathers info from a file opened for reading one line at a time
- we write to a file opened for reading using the `print` function with an argument  
`file=`

# Code Listing 14.1

## Review

```

1 # Prompt for three values: input file, output file, search string.
2 # Search for the string in the input file, write results to the
3 # output file
4
5 import sys
6 def process_file(i_file, o_file, a_str):
7     ''' if the a_str is in a line of i_file, add stars
8         to the a_str in line, write it out with the
9         line number to o_file '''
10    line_count_int = 1
11    for line_str in i_file:
12        if a_str in line_str:
13            new_line_str = line_str.replace(a_str, '***'+a_str)
14            print('Line {}: {}'.format(line_count_int, new_line_str), \
15                  file=o_file)
16        line_count_int += 1
17
18 try:
19     in_file_str = input("File to search:")
20     in_file = open(in_file_str, 'r', encoding='utf_8')
21 except IOError:
22     print('{} is a bad file name'.format(in_file_str))
23     sys.exit()
24
25 out_file_str = input("File to write results to:")
26 out_file = open(out_file_str, 'w')
27 search_str = input("Search for what string:")
28 process_file(in_file, out_file, search_str)
29 in_file.close()
30 out_file.close()

```

# results, searching for "This"

| inFile.txt           | outFile.txt                    |
|----------------------|--------------------------------|
|                      |                                |
| This is a test       | Line 1: ***This is a test      |
| This is only a test  |                                |
| Do not pass go       | Line 2: ***This is only a test |
| Do not collect \$200 |                                |

# More ways to read

- `my_file.read()`
  - Reads the entire contents of the file as a string and returns it. It can take an optional argument integer to limit the read to N bytes, that is `my_file.read(N)`
- `my_file.readline()`
  - Delivers the next line as a string.
- `my_file.readlines() # note plural`
  - Returns a ***single list*** of all the lines from the file

# example file

We'll work with a file called `temp.txt` which has the following file contents

First Line

Second Line

Third Line

Fourth Line

```
>>> temp_file = open("temp.txt", "r")      # open file for reading
>>> first_line_str = temp_file.readline() # read exactly one line
>>> first_line_str
'First line\n'
>>> for line_str in temp_file:      # read remaining lines
    print(line_str)
```

Second line

Third line

Fourth line

```
>>> temp_file.readline()                  # file read, return empty str
''
>>> temp_file.close()
```

```
>>> temp_file = open("temp.txt", "r")      # open file for reading
>>> temp_file.read(1)                      # read 1 char
'F'
>>> temp_file.read(2)                      # read the next 2 chars
'ir'
>>> temp_file.read()                      # read remaining file
'st line\nSecond line\nThird line\nFourth line\n'
>>> temp_file.read(1)                      # file read, return empty string
''
>>> temp_file.close()
```

```
>>> temp_file = open("temp.txt", "r")           # open file for reading
>>> file_contents_list = temp_file.readlines() # read all file lines into a list
>>> file_contents_list
['First line\n', 'Second line\n', 'Third line\n', 'Fourth line\n']
>>>
```

# More ways to write

- Once opened, you can write to a file (if the mode is appropriate):
  - `my_file.write(s)`
    - Writes the string `s` to the file
  - `my_file.writelines(lst)`
    - write a ***list of strings*** (one at a time) to the file

```
>>> word_list = ['First', 'Second', 'Third', 'Fourth']
>>> out_file = open('outfile.txt', 'w')
>>> for word in word_list:
...     out_file.write(word + ' line\n')
...
>>> out_file.close()
>>>
```

# Universal New Line

# Different OS's, different format

- Each operating system (Windows, OS X, Linux) developed certain standards for representing text
- In particular, they chose different ways to represent the end of a file, the end of a line, etc.

| Operating System | Character Combination |
|------------------|-----------------------|
| Unix & Mac OS X  | '\n'                  |
| MS Windows       | '\r\n'                |
| Mac (pre-OS X)   | '\r'                  |

**TABLE 14.1** End-of-Line Characters

# Universal new line

- To get around this, Python provides **by default** a special file option to deal with variations of OS text encoding called Universal new line
- you can over-ride this with an option to `open` called `newline=`
  - look at the docs for what this entails.

# Working with a file

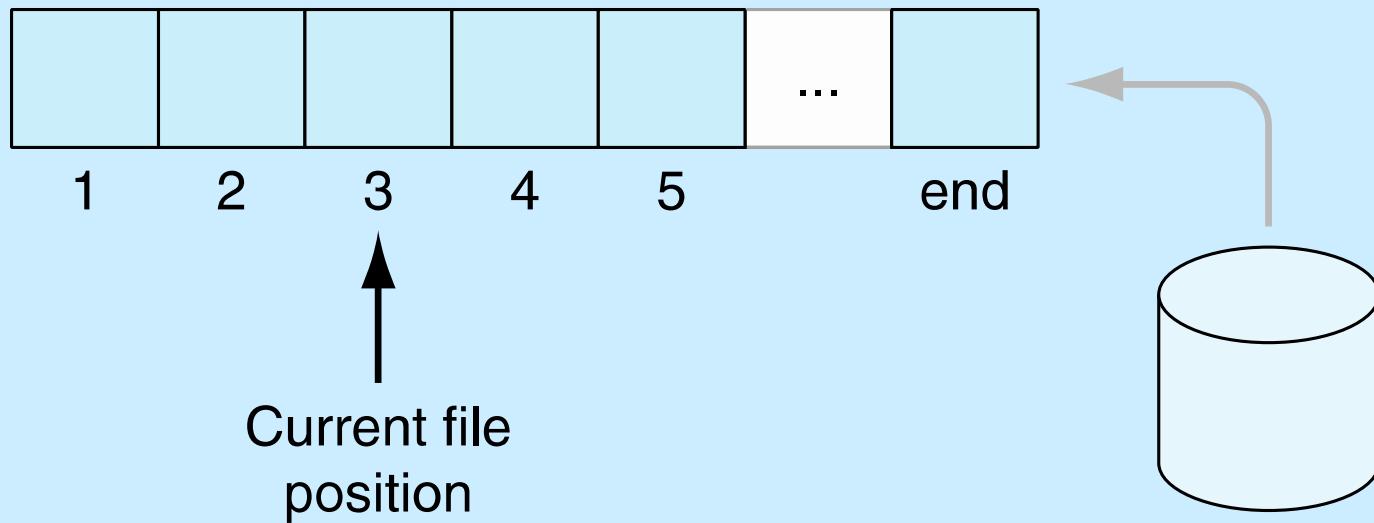
# The current file position

- Every file maintains a ***current file position***.
- It is the current position in the file, and indicates what the file will read next
- Is set by the mode table above

# Remember the file object buffer

- When the disk file is opened, the contents of the file are copied into the buffer of the file object
- Think of the file object as a very big list, where every index is one of the pieces of information of the file
- The current position is the present index in that list

## File object buffer



# the tell() method

- The `tell()` method tells you the current file position
- The positions are in bytes (think characters for UTF-8) from the beginning of the file

```
my_file.tell() => 42L
```

# the seek() method

- the `seek( )` method updates the current file position to a new file index (in bytes offset from the beginning of the file)
- `fd.seek(0) # to the beginning of the file`
- `fd.seek(100) # 100 bytes from beginning`

# counting bytes is a pain

- counting bytes is a pain
- `seek` has an optional argument set:
  - 0: count from the beginning
  - 1: count for the current file position
  - 2: count from the end (backwards)

# every read moves current forward

- every read/readline/readlines moves the current pos forward
- when you hit the end, every read will just yield '' (empty string), since you are at the end
  - no indication of end-of-file this way!
- you need to seek to the beginning to start again (or close and open, seek is easier)

```
>>> test_file = open('temp.txt', 'r')
>>> test_file.tell()                      # where is the current file position?
0
>>> test_file.readline()                  # read first line
'First Line\n'
>>> test_file.tell()                      # where are we now?
11
>>> test_file.seek(0)                     # go to beginning
0
>>> test_file.readline()                  # read first line again
'First Line\n'
>>> test_file.readline()                  # read second line
'Second Line\n'
>>> test_file.tell()                      # where are we now?
23
>>> test_file.seek(0, 2)                  # go to end
46
>>> test_file.tell()                      # where are we now?
46
>>> test_file.readline()                  # try readline at end of file: nothing there
''
>>> test_file.seek(11)                    # go to the end of the first line (see tell above)
11
>>> test_file.readline()                  # when we read now we get the second line
'Second Line\n'
>>> test_file.close()
>>> test_file.readline()                  # Error: reading after file is closed
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    test_file.readline()
ValueError: I/O operation on closed file.
>>>
```

# with statement

open and close occur in pairs (or should) so Python provides a shortcut, the `with` statement

- creates a context that includes an exit which is invoked automatically
- for files, the exit is to close the file

with expression as variable:

suite

# File is closed automatically when the suite ends

```
>>> with open('temp.txt') as temp_file:  
...     temp_file.readlines()  
...  
['First line\n', 'Second line\n', 'Third line\n', 'Fourth line\n']  
>>>
```

# read(size=1)

- you can use the `read( )` method to read just one byte at a time, and in combination with `seek` move around the file and “look for things”. Once current is set, you can begin reading again

# More on CSV files

# spreadsheets

- The spreadsheet is a very popular, and powerful, application for manipulating data
- Its popularity means there are many companies that provide their own version of the spreadsheet
- It would be nice if those different versions could share their data

# CSV, basic sharing

- A basic approach to share data is the comma separated value (CSV) format
  - it is a text format, accessible to all apps
  - each line (even if blank) is a row
  - in each row, each value is separated from the others by a comma (even if it is blank)
  - cannot capture complex things like formula

# Spread sheet and corresponding CSV file

| Name    | Exam1  | Exam2  | Final Exam | Overall Grade |
|---------|--------|--------|------------|---------------|
| Bill    | 75.00  | 100.00 | 50.00      | <b>75.00</b>  |
| Fred    | 50.00  | 50.00  | 50.00      | <b>50.00</b>  |
| Irving  | 0.00   | 0.00   | 0.00       | <b>0.00</b>   |
| Monty   | 100.00 | 100.00 | 100.00     | <b>100.00</b> |
| Average |        |        |            | <b>56.25</b>  |

Name,Exam1,Exam2,Final Exam,Overall Grade

Bill,75.00,100.00,50.00,75.00

Fred,50.00,50.00,50.00,50.00

Irving,0.00,0.00,0.00,0.00

Monty,100.00,100.00,100.00,100.00

Average,,,56.25

# Even CSV isn't universal

- As simple as that sounds, even CSV format is not completely universal
  - different apps have small variations
- Python provides a module to deal with these variations called the csv module
- This module allows you to read spreadsheet info into your program

# csv reader

- import the csv module
- open the file as normally, creating a file object.
- create an instance of a csv reader, used to iterate through the file just opened
  - you provide the file object as an argument to the constructor
- iterating with the reader object yields a row as a list of strings

## Code Listing 14.2 (and output)

```
        workbook_file)

for row in workbook_reader:
    print(row)

workbook_file.close()

>>>
['Name', 'Exam1', 'Exam2', 'Final Exam', 'Overall Grade']
['Bill', '75.00', '100.00', '50.00', '75.00']
['Fred', '50.00', '50.00', '50.00', '50.00']
['Irving', '0.00', '0.00', '0.00', '0.00']
['Monty', '100.00', '100.00', '100.00', '100.00']
[]
['Average', '', '', '', '56.25']

>>>
```

# things to note

- Universal new line is working by default
  - needed for this worksheet
- A blank line in the CSV shows up as an empty list
- empty column shows up as an empty string in the list

# CSV writer

much the same, except:

- the opened file must have write enabled
- the method is `writerow`, and it takes a *list of strings* to be written as a row

# os module

# What is the os module

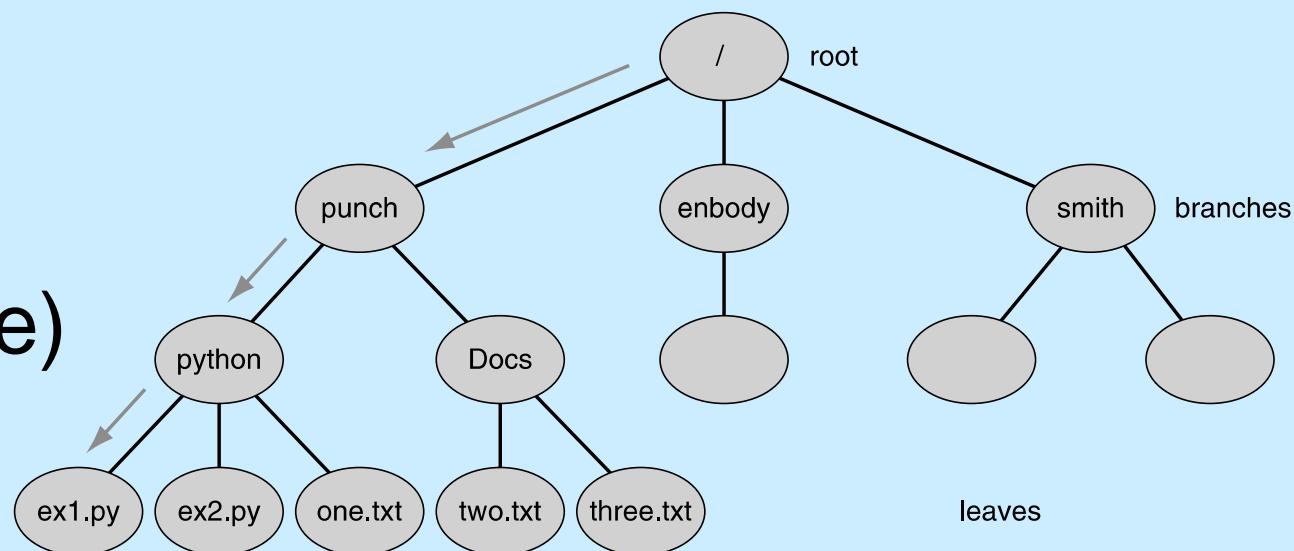
- The `os` module in Python is an interface between the operating system and the Python language.
- As such, it has many sub-functionalities dealing with various aspects.
- We will look mostly at the file related stuff

# What is a directory/folder?

- Whether in Windows, Linux or on OS X, all OS's maintain a ***directory structure***.
- A directory is a container of files or other directories
- These directories are arranged in a hierarchy or tree

# Computer Science *tree*

- it has a *root* node,  
with *branch*  
nodes, ends in  
*leaf* nodes
- the directory  
structure is  
hierarchy (tree)



# Directory tree

- Directories can be organized in a hierarchy, with the root directory and subsequent branch and leaf directories
- Each directory can hold files or other directories
- This allows for sub and super directories

# ***file path*** is a path through the tree

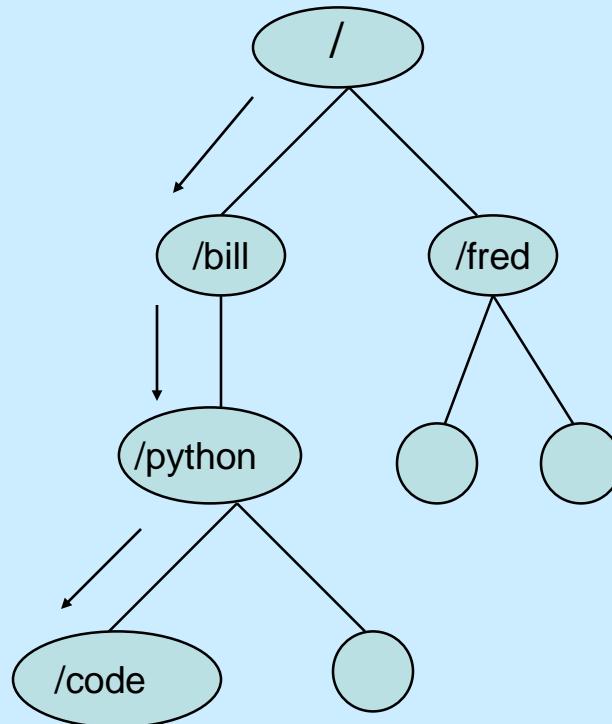
A path to a file is a path through the hierarchy to the node that contains a file

/bill/python/code/myCode.py

- path is from the root node /, to the bill directory, to the python directory, to the code directory where the file myCode.py resides

# the / in a path

- think of / as an operator, showing something is a directory
- follow the path, the leaf is either a directory or file



# a path String

- a valid path string for python is a string which indicates a valid path in the directory structure
- Thus '/Users/bill/python/code.py' is a valid path string

# different 'paths' for different os

- It turns out that each OS has its own way of specifying a path
  - C:\bill\python\myFile.py
  - /Users/bill/python/myFile.py
- Nicely, Python knows that and translates to the appropriate OS

# Two special directory names

- The directory name '`.`' is shortcut for the name of the current directory you are in as you traverse the directory tree
- The directory name '`..`' is a shortcut for the name of the parent directory of the current directory you are in

# Some os commands

- `os.getcwd()` Returns the full path of the current working directory
- `os.chdir(path_str)` Change the current directory to the path provided
- `os.listdir(path_str)` Return a list of the files and directories in the path (including '`.`' )

```
>>> import os                      # load the os package
>>> os.chdir("/punch/python")      # change to the example starting point
>>> os.getcwd()                  # check that we are there
'/punch/python'
>>> os.listdir(".") # list contents of current directory, indicated by "."
['ex1.py', 'ex2.py', 'one.txt']
>>> dir_list = os.listdir(".")      # we can give that list a name
>>> dir_list
['ex1.py', 'ex2.py', 'one.txt']
>>> os.listdir("/punch")          # list the contents at some path
['Docs', 'python']
```

# Some more os commands

- `os.rename(source_path_str, dest_path_str)` Renames a file or directory
- `os.mkdir(path_str)` make a new directory. So  
`os.mkdir(' /Users/bill/python/new' )` creates the directory `new` under the directory `python`.
- `os.remove(path_str)` Removes the file
- `os.rmdir(path_str)` Removes the directory, but the directory must be empty

# the walk function

- `os.walk(path_str)` Starts at the directory in `path_str`. It yields three values:
  - `dir_name`, name of the current directory
  - `dir_list`, list of subdirectories in the directory
  - `files`, list of files in the directory
- If you iterate through, walk will visit every directory in the tree. Default is top down

# Walk example

```
>>> os.getcwd()                                # check our starting point
'punch'
>>> for dir_name, dirs, files in os.walk("."):  # "walk" in the current directory
    print(dir_name, dirs, files)

. ['Docs', 'python'] []  # current directory, list of 2 subdirectories, no files
./Docs [] ['three.txt', 'two.txt']   # Does directory, no subdirectories, 3 files
./python [] ['ex1.py', 'ex2.py', 'one.txt'] # directory, no subdirectories, 3 files
>>>
```

# os.path module

# os.path module

allows you to gather some info on a path's existence

- `os.path.isfile(path_str)` is this a path to an existing file (T/F)
- `os.path.isdir(path_str)` is this a path to an existing directory (T/F)
- `os.path.exists(path_str)` the path (either as a file or directory) exists (T/F)

# os.path names

assume `p = '/Users/bill/python/myFile.py'`

- `os.path.basename(p)` returns `'myFile.py'`
- `os.path.dirname(p)` returns  
`'/Users/bill/python'`
- `os.path.split(p)` returns  
`[ '/Users/bill/python' , 'myFile.py' ]`
- `os.path.splitext(p)` returns  
`'/Users/bill/python/myFile' , '.py'`
- `os.path.join(os.path.split(p)[0] , 'other.py')` returns `'/Users/bill/python/other.py'`

## Code Listing 14.4

# Utility to find strings in files

- The main point of this function is to look through all the files in a directory structure and see if a particular string exists in any of those files
- Pretty useful for mining a set of files
- lots of comments so you can follow

```
def check(search_str, count, files_found_list, dirs_found_list):
    for dirname, dir_list, file_list in os.walk("."):      # walk the subtree
        for f in file_list:
            if os.path.splitext(f) [1] == ".txt": # if it is a text file
                count = count + 1           # add to count of files examined
                a_file = open(os.path.join(dirname, f), 'r') # open text file
                file_str = a_file.read()     # read whole file into string

                if search_str in file_str:      # is search_str in file?
                    filename = os.path.join(dirname, f) # if so, create path
                                            for file
                    files_found_list.append(filename) # and add to file list
                    if dirname not in dirs_found_list: # if directory is not
                        dirs_found_list.append(dirname) # and directory list
                a_file.close()

    return count
```

# More Exceptions

# What we already know

try-except suite to catch errors:

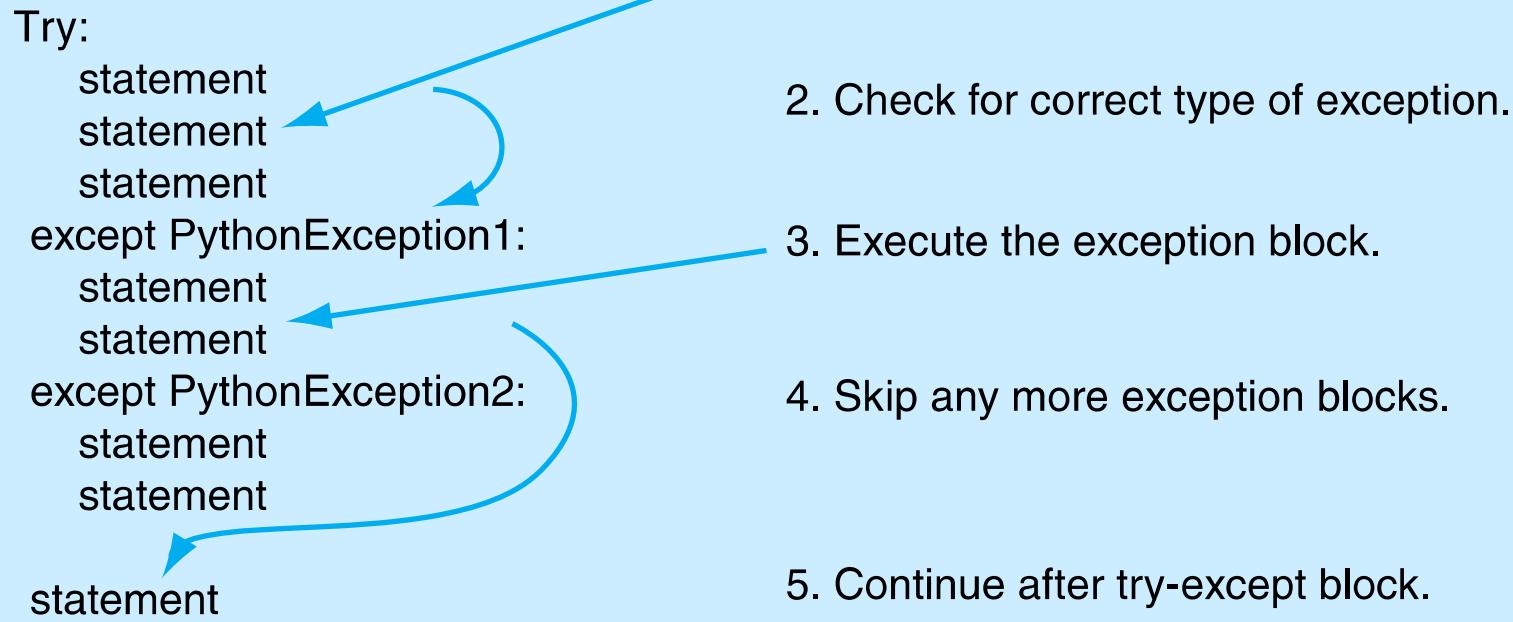
```
try:  
    suite to watch
```

```
except ParticularError  
    error suite
```

# more of what we know

- `try` suite contains code that we want to watch:
  - if an error occurs the `try` suite stops and looks for an `except` suite that can handle the error
- `except` suite has a particular error it can handle and a suite of code for handling that error

# Error Flow



**FIGURE 14.5** Exception flow.

## Code Listing 14.5

ing on with the rest of the program

# continuing

```
>>> ===== RESTART =====  
>>>  
Entering the try suite  
Provide a dividend to divide:10  
Provide a divisor to divide by:0  
Divide by 0 error  
Continuing on with the rest of the program  
>>> ===== RESTART =====  
>>>  
Entering the try suite  
Provide a dividend to divide:  
  
Traceback (most recent call last):  
  File "/Users/bill/book/v3.5/chapterExceptions/divide.py", line 3, in <module>  
    dividend = float(input("Provide a dividend to divide:"))  
KeyboardInterrupt  
>>>
```

# Check for specific exceptions

- Turns out that you don't have to check for an exception type. You can just have an exception without an particular error and it will catch anything
- That is a bad idea. How can you fix (or recover from) an error if you don't know the kind of exception
- Label your exceptions, all that you expect!

# What exceptions are there?

- In the present Python, there is a set of exceptions that are pre-labeled.
- To find the exception for a case you are interested it, easy enough to try it in the interpreter and see what comes up
- The interpreter tells you what the exception is for that case.

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        +-- FloatingPointError
        +-- OverflowError
        +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    +-- LookupError
        +-- IndexError
        +-- KeyError
    +-- MemoryError
    +-- NameError
        +-- UnboundLocalError
+-- OSError
    +-- BlockingIOError
    +-- ChildProcessError
    +-- ConnectionError
        +-- BrokenPipeError
        +-- ConnectionAbortedError
        +-- ConnectionRefusedError
        +-- ConnectionResetError
    +-- FileExistsError
    +-- FileNotFoundError
    +-- InterruptedError
    +-- IsADirectoryError
    +-- NotADirectoryError
    +-- PermissionError
    +-- ProcessLookupError
    +-- TimeoutError
```

# From Python docs webpage

```
+-- ReferenceError
+-- RuntimeError
    +-- NotImplementedError
    +-- RecursionError
+-- SyntaxError
    +-- IndentationError
    +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
    +-- UnicodeError
        +-- UnicodeDecodeError
        +-- UnicodeEncodeError
        +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

# Examples

In [1]: 1/0Out [1]:

Traceback (most recent call last): File "<pyshell#9>", line 1, in <module> 1/0

**ZeroDivisionError: integer division or modulo by zero**

error  
names  
CAPS  
matter!

In [2]: open("junk")

Out [2]: Traceback (most recent call last): File "<stdin>", line 1, in <module>

**FileNotFoundException: [Errno 2] No such file or directory: 'junk'**

# Philosophy of Exception Handling

# Dealing with problems

Two ways to deal with exceptions

- **LBYL:** Look Before you Leap
- **EAFP:** Easier to Ask Forgiveness than Permission (famous quote by Grace Hopper)

# Look Before You Leap

- By this we mean that before we execute a statement, we check all aspects to make sure it executes correctly:
  - if it requires a string, check that
  - if it requires a dictionary key, check that
- Tends to make code messy. The heart of the code (what you want it to do) is hidden by all the checking.

# Easier to Ask Forgiveness than Permission

- By this we mean, run any statement you want, no checking required
- However, be ready to “clean up any messes” by catching errors that occur
- The `try` suite code reflects what you want to do and the `except` code what you want to do on error. Cleaner separation!

# Python likes EAFP

- Code Python programmers support the **EAFP** approach:
  - run the code, let the `except` suites deal with the errors. Don't check first.

## Code Listing 14-6

```
# check whether int conversion will raise an error, two examples.  
# Python Idioms, http://jaynes.colorado.edu/PythonIdioms.html  
  
#LBYL, test for the problematic conditions  
def test_lbyl (a_str) :  
    if not isinstance(a_str, str) or not a_str.isdigit:  
        return None  
    elif len(a_str) > 10:      #too many digits for int conversion  
        return None  
    else:  
        return int(a_str)  
  
#EAFP, just try it, clean up any mess with handlers  
def test_eafp(a_str) :  
    try:  
        return int(a_str)  
    except (TypeError, ValueError, OverflowError): #int conversion failed  
        return None
```

# Extensions to the basic Exception Model

# Version 2, finally suite

- you can add a `finally` suite at the end of the `try/except` group
- the `finally` suite is run as you exit the `try/except` suite, ***no matter whether an error occurred or not.***
  - even if an exception raised in the `try` suite was not handled!
- Gives you an opportunity to clean up as you exit the `try/except` group

# `finally` and `with`

`finally` is related to a `with` statement:

- creates a context (the `try` suite)
- has an exit, namely execute the `finally` suite

# Version 3, else

- One way to think about things is to think of the `try` as a kind of condition (an exception condition) and the `excepts` as conditional clauses
- if an exception occurs then you match the exception
- the `else` clause covers the non-exception condition. It runs when the `try` suite does not encounter an error.

# The whole thing

```
try:  
    code to try  
except PythonError1:  
    exception code  
except PythonError2:  
    exception code  
except:  
    default except code  
else:  
    non exception case  
finally:  
    clean up code
```

## Code Listing 14-7

```
# all aspects of exceptions

def process_file(data_file):
    """Print each line of a file with its line number."""
    count = 1
    for line in data_file:
        print('Line ' + str(count) + ': ' + line.strip())
        count = count + 1

while True:      # loop forever: until "break" is encountered
    filename = input('Input a file to open: ')
    try:
        data_file = open(filename)
    except IOError:                      # we get here if file open failed
        print('Bad file name; try again')
    else:
        # no exception so let's process the file
        print('Processing file',filename)
        process_file(data_file)
        break      # exit "while" loop (but do "finally" block first)
    finally:      # we get here whether there was an exception or not
        try:
            data_file.close()
        except NameError:
            print('Going around again')

print('Line after the try-except group')
```

# Raising and creating your own exceptions

# invoking yourself, raise

- You can also choose to invoke the exception system anytime you like with the `raise` command

```
raise MyException
```

- you can check for odd conditions, `raise` them as an error, then catch them
- they must be part of the existing exception hierarchy in Python

# Non-local catch

- Interestingly, the `except` suite does not have to be right next to the `try` suite.
- In fact, the `except` that catches a `try` error can be in another function
- Python maintains a chain of function invocations. If an error occurs in a function and it cannot catch it, it looks to the function that called it to catch it

# Make your own exception

- You can make your own exception.
- Exceptions are classes, so you can make a new exception by making a new subclass:

```
class MyException (IOError):  
    pass
```

- When you make a new class, you can add your own exceptions.

## Code Listing 14.9 password manager

# part 1

```
1 import string
2
3 # define our own exceptions
4 class NameException (Exception):
5     ''' For malformed names '''
6     pass
7 class PasswordException (Exception):
8     ''' For bad password '''
9     pass
10 class UserException (Exception):
11     ''' Raised for existing or missing user '''
12     pass
13
14 def check_pass(pass_str, target_str):
15     """Return True, if password contains characters from target."""
16     for char in pass_str:
17         if char in target_str:
18             return True
19     return False
20
21 class PassManager(object):
22     """A class to manage a dictionary of passwords with error checking."""
23     def __init__(self, init_dict=None):
24         if init_dict==None:
25             self.pass_dict={}
```

# part 2

```
26     else:
27         self.pass_dict = init_dict.copy()
28
29     def dump_passwords(self):
30         return self.pass_dict.copy()
31
32     def add_user(self,user):
33         """Add good user name and strong password to password dictionary."""
34         if not isinstance(user,str) or not user.isalnum():
35             raise NameException
36         if user in self.pass_dict:
37             raise UserException
38         pass_str = input('New password:')
39         # strong password must have digits, uppercase and punctuation
40         if not (check_pass(pass_str, string.digits) and\
41                 check_pass(pass_str, string.ascii_uppercase) and\
42                 check_pass(pass_str, string.punctuation)):
43             raise PasswordException
44
45     def validate(self,user):
46         """Return True, if valid user and password."""
47         if not isinstance(user,str) or not user.isalnum():
48             raise NameException
49         if user not in self.pass_dict:
50             raise UserException
51         password = input('Passwd:')
52         return self.pass_dict[user]==password
```

GLOBAL  
EDITION



# chapter 10

## Recursion: Another Control Mechanism

PEARSON

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody



P Pearson

ALWAYS LEARNING

# a function that calls itself

- The very basic meaning of a recursive function is a ***function that calls itself***
- Leads to some funny definitions:
  - Def: recursion. see recursion
- However, when you first see it, it looks odd.
- Look at the recursive function that calculates factorial (code listing 16-2)

# It doesn't do anything!

- This is a common complaint when one first sees a recursive function. What exactly is it doing? It doesn't seem to do anything!
- Our goal is to understand what it means to write a recursive function from a programmers and computers view.

# Defining a recursive function

# 1) divide and conquer

- Recursion is a natural outcome of a divide and conquer approach to problem solving
- A recursive function defines how to break a problem down (divide) and how to reassemble (conquer) the sub-solutions into an overall solution

## 2) base case

- recursion is a process not unlike loop iteration.
  - You must define how long (how many iterations) recursion will proceed through until it stops
- The base case defines this limit
- Without the base case, recursion will continue infinitely (just like an infinite loop)

# Simple Example

Lao-Tzu: “A journey of 1000 miles begins with a single step”

```
def journey (steps):
```

- the first step is easy (base case)
- the  $n^{\text{th}}$  step is easy having complete the previous  $n-1$  steps (divide and conquer)

## Code Listing 15-1

```
def take_step(n):
    if n == 1:  # base case
        return "Easy"
    else:
        this_step = "step(" + str(n) + ")"
        previous_steps = take_step(n-1)      # recursive call
        return this_step + " + " + previous_steps
```

# Factorial

You remember factorial?

- $\text{factorial}(4) = 4! = 4 * 3 * 2 * 1$
- The result of the last step, multiply by 1, is defined based on all the previous results that have been calculated

## Code Listing 15-2

```
def factorial(n) :  
    """Recursive factorial."""  
    if n == 1:  
        return 1                                # base case  
    else:  
        return n * factorial(n-1)      # recursive case
```

# Trace the calls

$4! = 4 * 3!$  start first function invocation

$3! = 3 * 2!$  start second function invocation

$2! = 2 * 1!$  start third function invocation

$1! = 1$  fourth invocation, base case

$2! = 2 * 1 = 2$  third invocation finishes

$3! = 3 * 2 = 6$  second invocation finishes

$4! = 4 * 6 = 24$  first invocation finishes

# Another: Fibonacci

- Depends on the Fibonacci results for the two previous values in the sequence
- The base values are  $\text{Fibo}(0) == 1$  and  $\text{Fibo}(1) == 1$

$$\text{Fibo}(x) = \text{Fibo}(x-1) + \text{Fibo}(x-2)$$

## Code Listing 15-3

```
def fibonacci(n):
    """Recursive Fibonacci sequence."""
    if n == 0 or n == 1:      # base cases
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2) # recursive case
```

# Trace

- $\text{fibo}(4) = \text{fibo}(3) + \text{fibo}(2)$
- $\text{fibo}(3) = \text{fibo}(2) + \text{fibo}(1)$
- $\text{fibo}(2) = \text{fibo}(1) + \text{fibo}(0) = 2$  # base case
- $\text{fibo}(3) = 2 + \text{fibo}(1) = 3$  # base case
- $\text{fibo}(4) = 3 + 2 = 5$

# Reverse string

- We know Python has a very simple way to reverse a string, but let's see if we can write a recursive function that reverses a string without using slicing.

## Code Listing 15-4

### Template for reversal

```
# recursive function to reverse a string

def reverser (a_str):
    # base case
    # recursive step
        # divide into parts
        # conquer/reassemble

the_str = input("Reverse what string:")
result = reverser(the_str)
print("The reverse of {} is {}".format(the_str, result))
```

# Base Case

- What string do we know how to trivially reverse?
- Yes, a string with one character, when reversed, gives back exactly the same string
- We use this as our base case

## Code Listing 15-5

```
def reverser (aStr):  
    if len(aStr) == 1:          # base case  
        return aStr  
    # recursive step  
    # divide into parts  
    # conquer/reassemble  
theStr = raw_input("Reverse what string: ")  
result = reverser(theStr)  
print "Reverse of %s is %s\n" % (theStr, result)
```

# Recursive step

- We must base the recursive step on what came before, plus the extra step we are presently in.
- Thus the reverse of a string is the reverse of all but the first character of the string, which is placed at the end. We assume the rev function will reverse the rest

$$\text{rev}(s) = \text{rev}(s[1:]) + s[0]$$

## Code Listing 15-6

```
#recursive function to reverse a string

def reverse (a_str):
    #base case
    if len(a_str)==1:
        return a_str
    #recursive step
    else:
        return reverse(a_str[1:])+a_str[0]

the_str = input("Revrese what string: ")
result = reverse(the_str)
print("The reverse of {} is {}".format(the_str,result))
```

```
>python reverser.py
```

```
What string:abcde
```

```
Got as an argument: abcde
```

```
Got as an argument: bcde
```

```
Got as an argument: cde
```

```
Got as an argument: de
```

```
Got as an argument: e
```

```
Base Case!
```

```
Reassembling e and d into ed
```

```
Reassembling de and c into edc
```

```
Reassembling cde and b into edcb
```

```
Reassembling bcde and a into edcba
```

# How does Python keep track?

# The Stack

- A **Stack** is a data structure, like a List or a Dictionary, but with a few different characteristics
- A Stack is a sequence
- A stack only allows access to one end of its data, the **top** of the stack



Figure 15.1

# Operations

- `pop` : remove top of stack. Stack is one element smaller
- `push (val)` : add `val` to the stack. `Val` is now the top. Stack is one element larger
- `top` : Reveals the top of the stack. No modification to stack

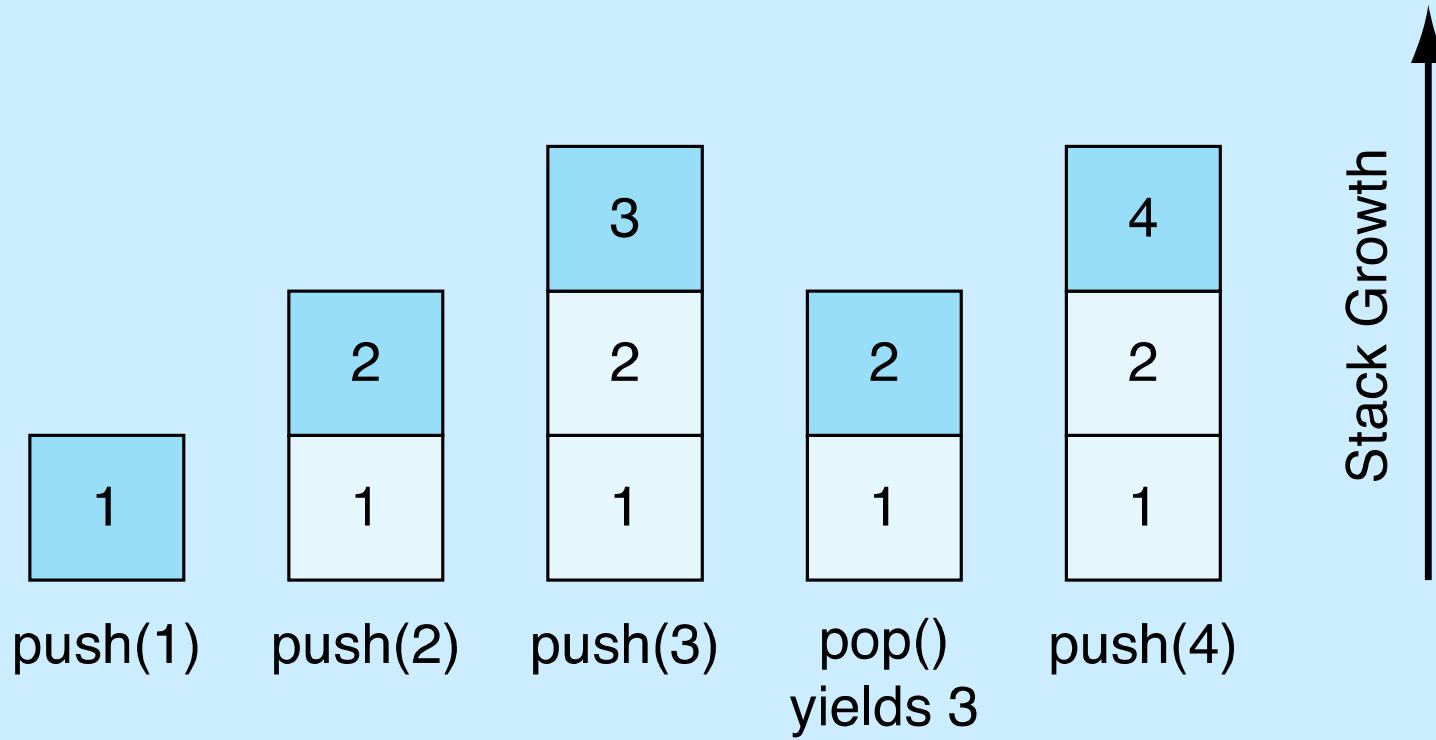


Figure 15.2 The operation of a stack data structure

# Stack of function calls

Python maintains a stack of function calls.

- If a function calls another function recursively, the new function is *pushed* onto the calling stack and the previous function waits
- The top is always the active function
- When a *pop* occurs, the function below becomes active

## Code Listing 15-7

```
def factorial(n) :
    """recursive factorial with print to show operation."""
    indent = 4*(6-n)*" " # more indent on deeper recursion
    print(indent + "Enter factorial n = ", n)
    if n == 1:           # base case
        print(indent + "Base case.")
        return 1
    else:              # recursive case
        print(indent + "Before recursive call f(" + str(n-1) + ")")
        # separate recursive call allows print after call
        rest = factorial(n-1)
        print(indent + "After recursive call f(" + str(n-1) + ") = ", rest)
    return n * rest
```

```
>>> factorial(4)
        Enter factorial n = 4
        Before recursive call f(3)
            Enter factorial n = 3
            Before recursive call f(2)
                Enter factorial n = 2
                Before recursive call f(1)
                    Enter factorial n = 1
                    Base case.
                    After recursive call f(1) = 1
                    After recursive call f(2) = 2
                    After recursive call f(3) = 6
```

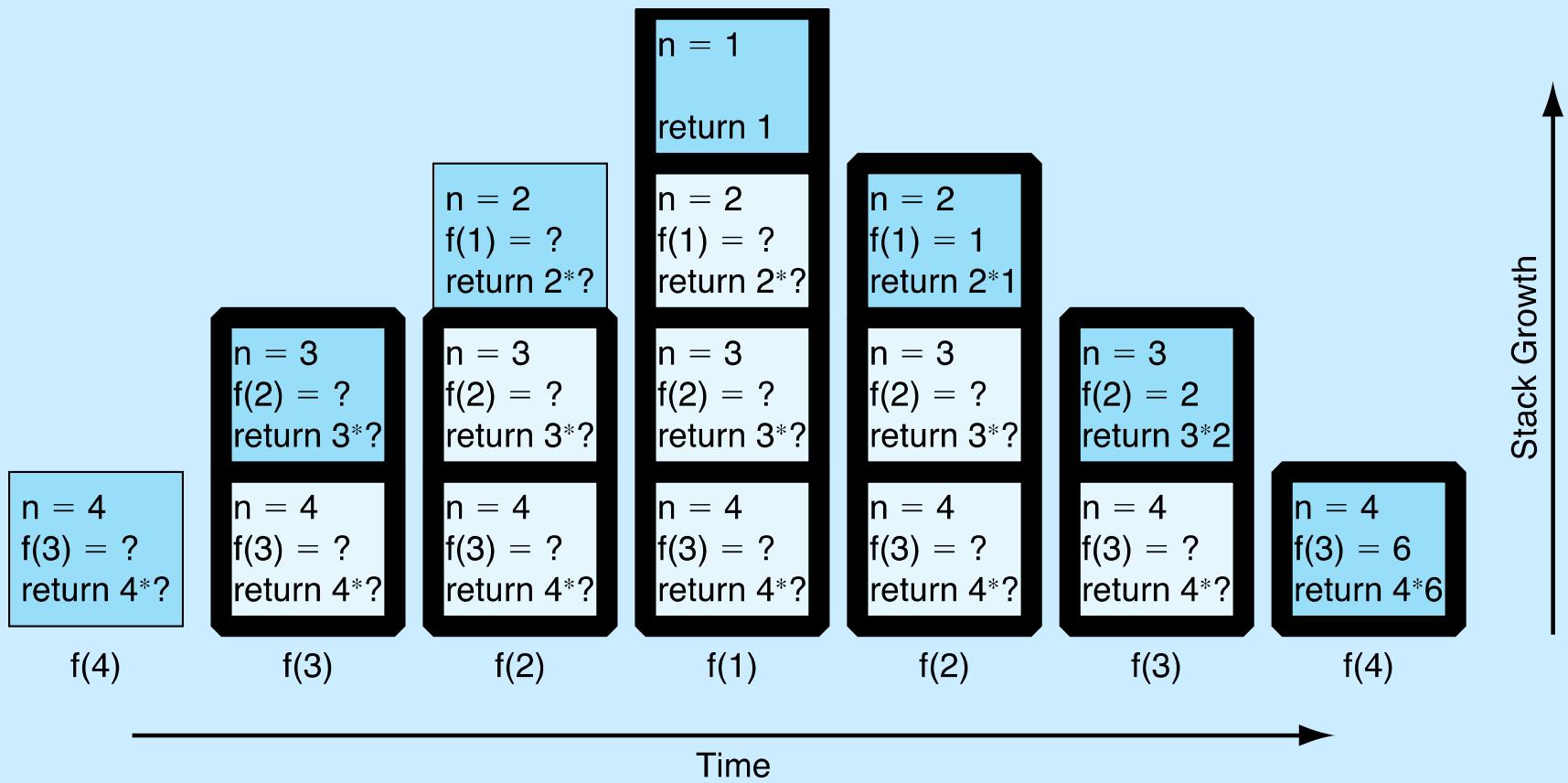
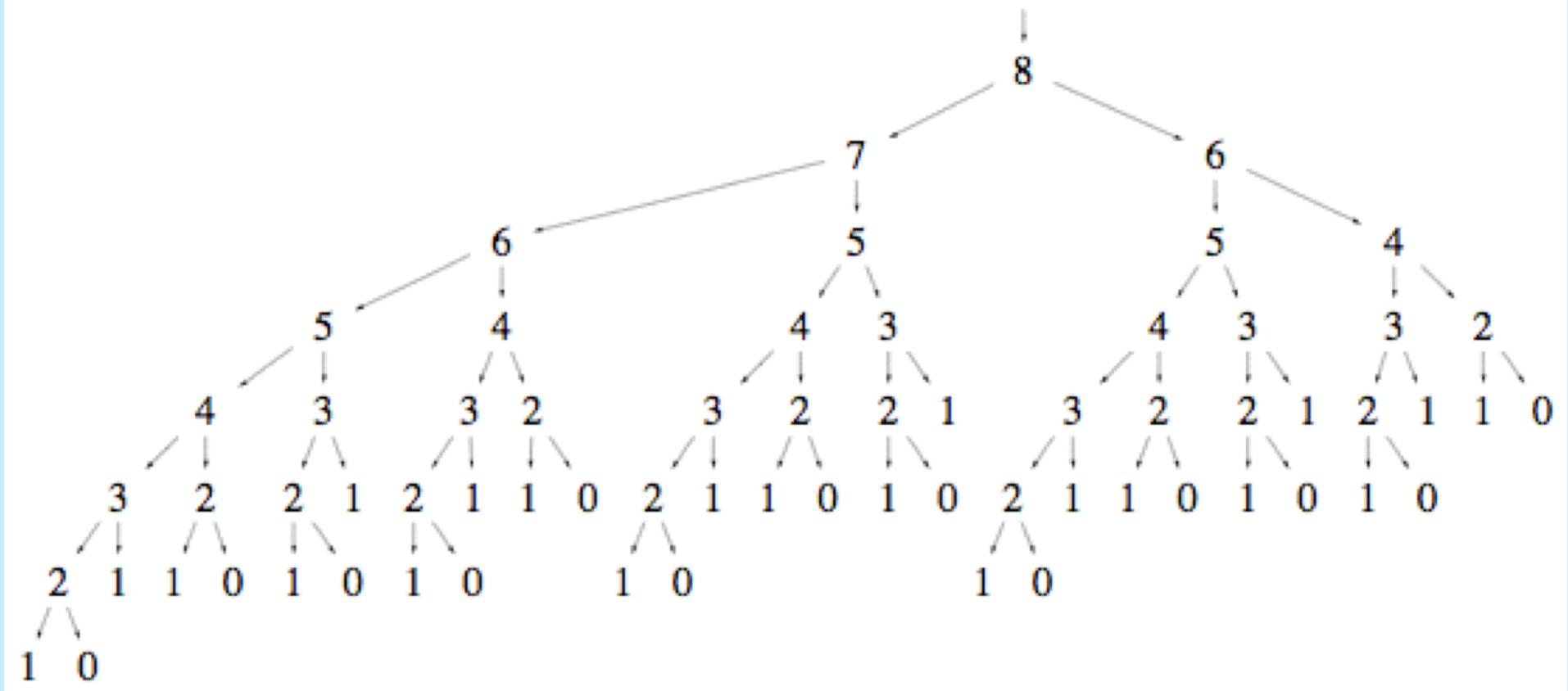


Figure 15.3 Call stack for  $\text{factorial}(4)$ . Note the question marks.

# A Better Fibonacci

The recursive function that we have written previously is very wasteful

It calls the function with the same argument many times, never "remembering" the previous result.



## Code Listin 15.8

### Fibonacci with memory

```
def fibonacci(n):
    """Recursive fibonacci that remembers previous values"""
    if n not in fibo_dict:
        # recursive case, store in the dict
        fibo_dict[n] = fibonacci(n-1) + fibonacci(n-2)
    return fibo_dict[n]

# global fibonacci dictionary.
fibo_dict = {}

# enter the base cases
fibo_dict[0] = 1
fibo_dict[1] = 1

fibo_val = input("Calculate what Fibonacci value:")
print("Fibonacci value of", fibo_val, "is",
      fibonacci(int(fibo_val)))
```

# Recursive Figures

# Fractal Objects

- You can use recursion to draw many real world figures
- Fractals are a class of drawing that has a couple of interesting properties:
  - upon magnification, the “shape” of the figure remains the same
  - the resulting figure has a floating point dimensionality value (2.35 D for example)

# Algorithm to draw a tree

1. Draw an edge
2. turn left
3. draw edge and left branch # recurse
4. turn right
5. draw edge and right branch #recurse
6. turn left
7. backup

## Code Listing 15-9

```
from turtle import *
def branch(length, level):
    if level <= 0:                      # base case
        return
    forward(length)
    left(45)
    branch(0.6*length, level-1)          # recursive case: left branch
    right(90)

    branch(0.6*length, level-1)          # recursive case: right branch
    left(45)
    backward(length)
    return

# turn to get started
left(90)
branch(100,5)
```

# Couple things

- note that length is reduced as we recurse down (making for shorter branches)
- the numbers on the right of the following picture show the order in which the branches are drawn

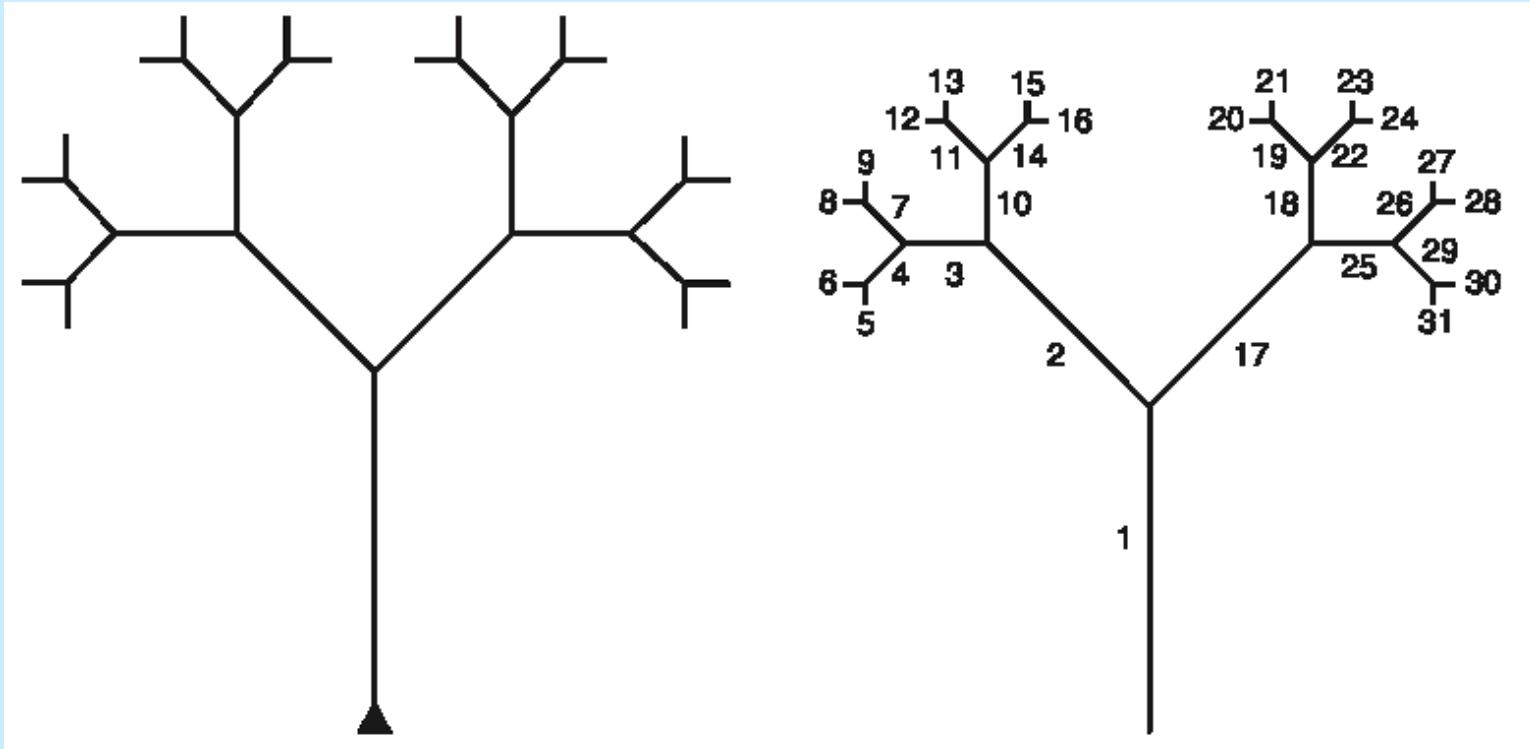


Figure 15.5 Recursive tree; (a) Python-drawn (left); (b) order-of-drawing on right.

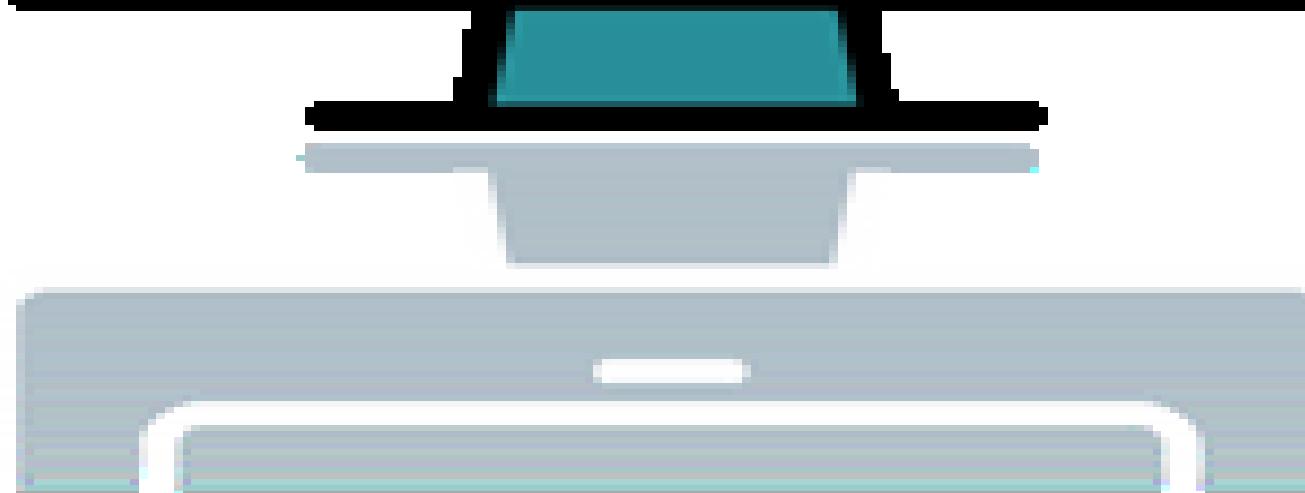
# Sierpinski Triangles

Little simpler than the tree:

1. draw edge
2. recurse
3. backward
4. turn 120 degrees

## Code Listing 15.10

### Sierpinski Triangles



```
# Draw Sierpinski figure

from turtle import *

def sierpinski(length, depth):
    if depth > 1: dot() # mark position to better see recursion
    if depth == 0:      # base case
        stamp() # stamp a triangular shape
    else:
        forward(length)
        sierpinski(length/2, depth-1) # recursive call
        backward(length)
        left(120)
        forward(length)
        sierpinski(length/2, depth-1) # recursive call
        backward(length)
        left(120)
        forward(length)
        sierpinski(length/2, depth-1) # recursive call
        backward(length)
        left(120)

sierpinski(200, 6)
```

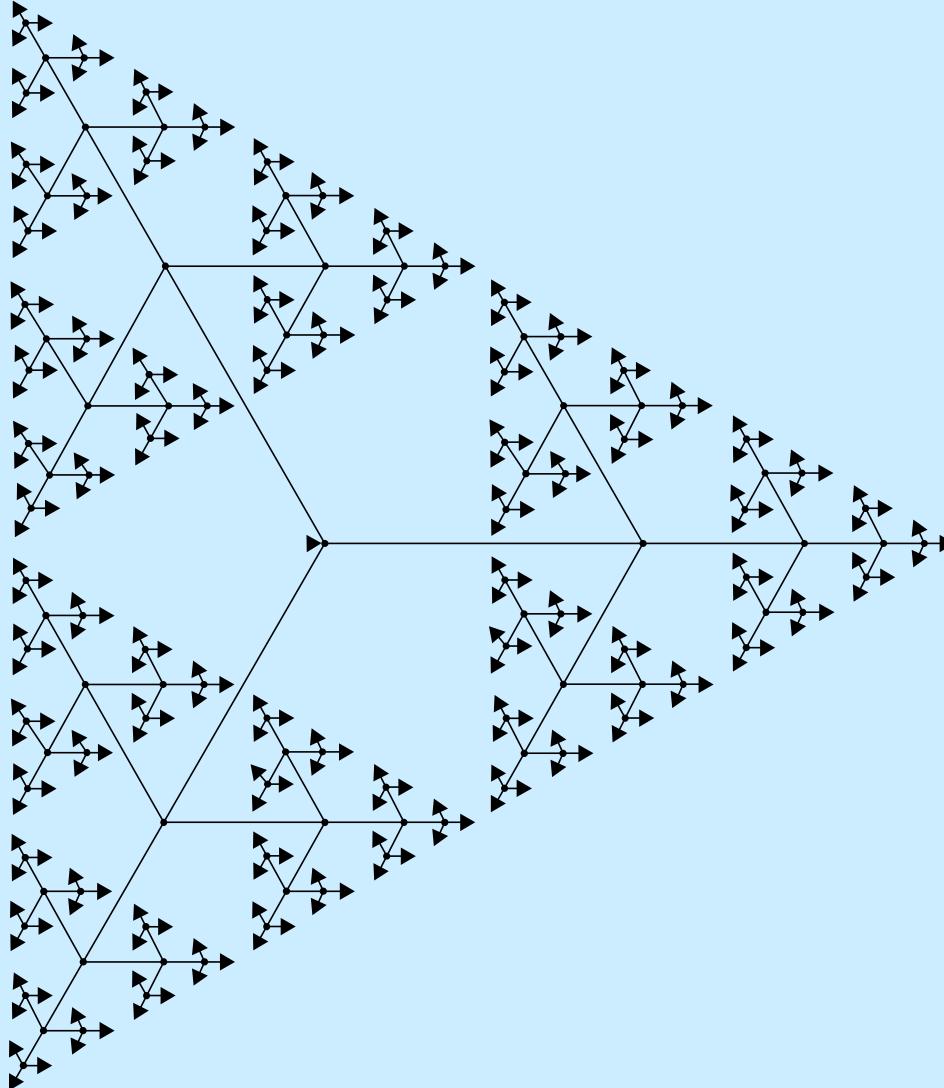


Figure 15.6 Sierpinski triangle

# Some recursive details

- Recursive functions are easy to write and lend themselves to divide and conquer
- They can be slow (all the pushing and popping on the stack)
- Can be converted from recursive to iterative but that can be hard depending on the problem.