Lab 1
Notes on Assignment One & Python Tutorial

Dr Jianhua He
University of Essex
Email: j.he@essex.ac.uk

Overview

1. Assignment One
2. Installing & Running Python
3. Python Basics
4. Sequences types: Lists, Tuples, and Strings
5. IF Statements
6. Iteration Statements
7. Functions
8. File Input/Output (I/O)

1. Assignment on Monoalphabetic Cipher

- Documents for Assignments 1 are now available at Moodle under folder *Assignment Documents*
  - with a sample Python program for Caesar cipher
  - Deadlines for Assignment 1: week 19, 12-Feb-2021
  - Example Python notebook program for Caesar cipher at Google Colab
- Task:
  - Design and develop a Python program with encryption and decryption functions for monoalphabetic cipher.
- How to submit
  - Submit one python program file to Faser called: `mc_registrationnumber.py`
- Go through the Assignment 1 document (pay attention to the annotation)

Python Program of Caesar Cipher

- Install and use programming software (Python), or use Google Colab
  - Your computer may have already installed Python.
  - You can check it by typing python in Terminal.
- Example Python notebook program for Caesar cipher
  - https://colab.research.google.com/drive/1l7UcF7EeAoONchZvTSXKm3ZQ-Q7rSScJ?usp=sharing
  - You should not change the shared program. Instead, you can download a copy and make changes to your own copy using Google colab
    - » Google colab: https://colab.research.google.com/notebooks/intro.ipynb
  - You can download the code as Python file, make changes and run it in Terminal or in a Python integrated development environment (IDE) such as PyCharm or default IDLE
    - » PyCharm CE: https://www.jetbrains.com/pycharm/ (for Windows, macOS, Linux)

2. Installing & Running Python

Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning
- Considered a scripting language, but is much more
- Scalable, object oriented and functional from the beginning
- Used by Google from the beginning
- Increasingly popular

Installing

- Python is pre-installed on Linux and MAC OS
- The pre-installed version may not be the most recent one (version 3.9 as of Jan 2021; ver 2.7, end of support)
- Download from http://python.org/download/
- Python comes with a large library of standard modules
- There are several options for an IDE, e.g.,
  - IDLE – works well with MAC OS, Windows
  - PyCharm: https://www.jetbrains.com/pycharm/ download
  - A youtube video explain installation of Python and Pycharm (and a little bit use of Pycharm to run python program): Install #Python 3.8 and #PyCharm on Windows 10

The Python Interpreter

- Typical Python implementations offer both an interpreter and compiler
- Interactive interface to Python with a read-eval-print loop

```
(base) jianhuahe@MBP ~ % python
Python 3.8.5 (default, Sep  4 2020, 02:22:02)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a=2
>>> a*5
10
```

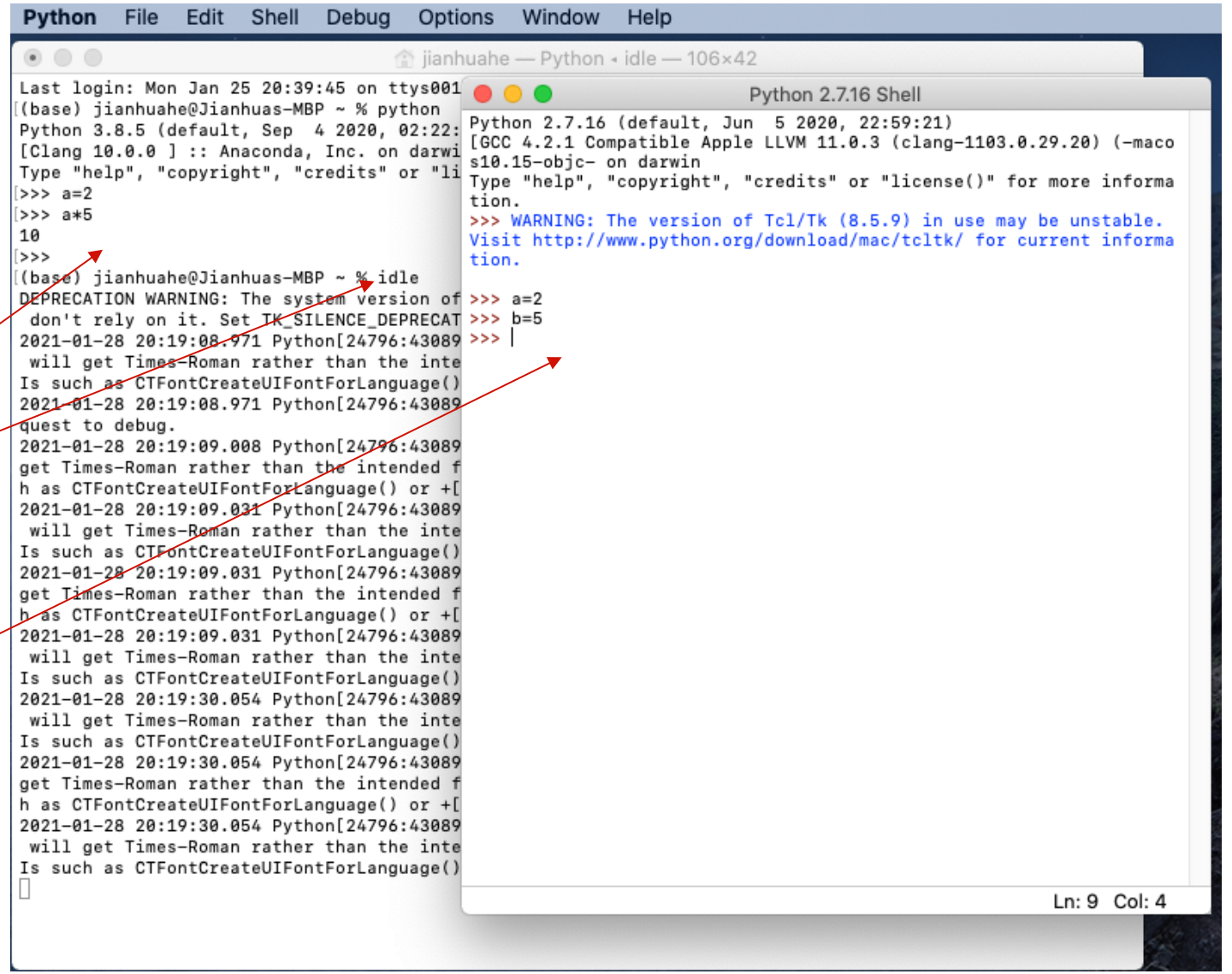
IDLE Development Environment

- IDLE is an Integrated DeveLopment Environment for Python
  - Open from Applications in Windows

For MAC OS

- Open `terminal`
- Type `idle`
- IDLE window appears

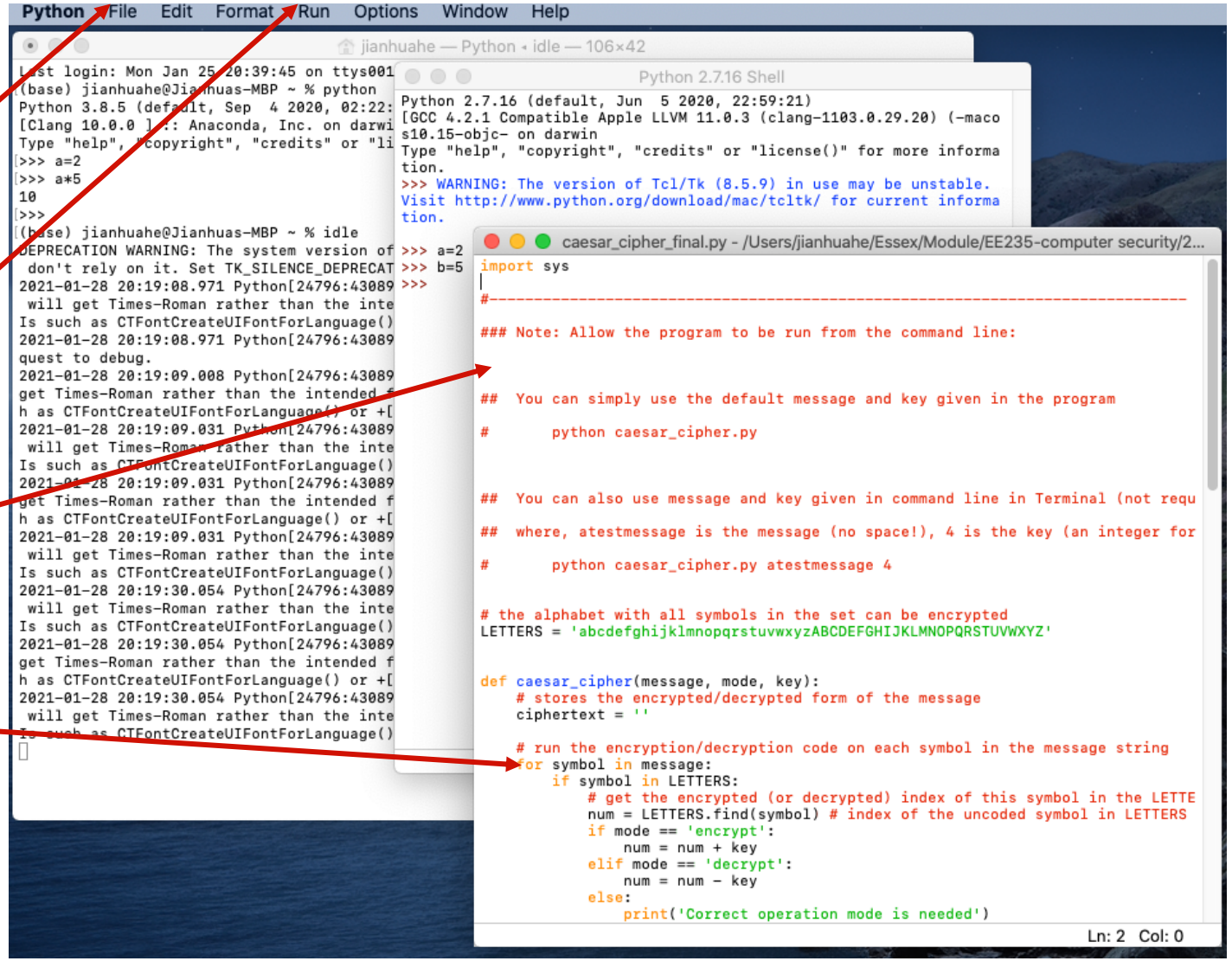
- Install python 3.9
- Type `idle3`



Editing Python in IDLE Editor

- Multi-window text editor with syntax highlighting, auto-completion, smart indent and other.
- Python shell with syntax highlighting.
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility

- Create/Open files
- Editor appears
- Edit file
- Run file



Running Interactively in Terminal or IDLE

- In terminal, type `python`

  You will enter a python sheel. Python prompts '>>>' will appear.

- Type commands following Python prompts with '>>>'.

```
>>> print("Hello, World!")
>>> 3+3
6
>>> a =3
3
```

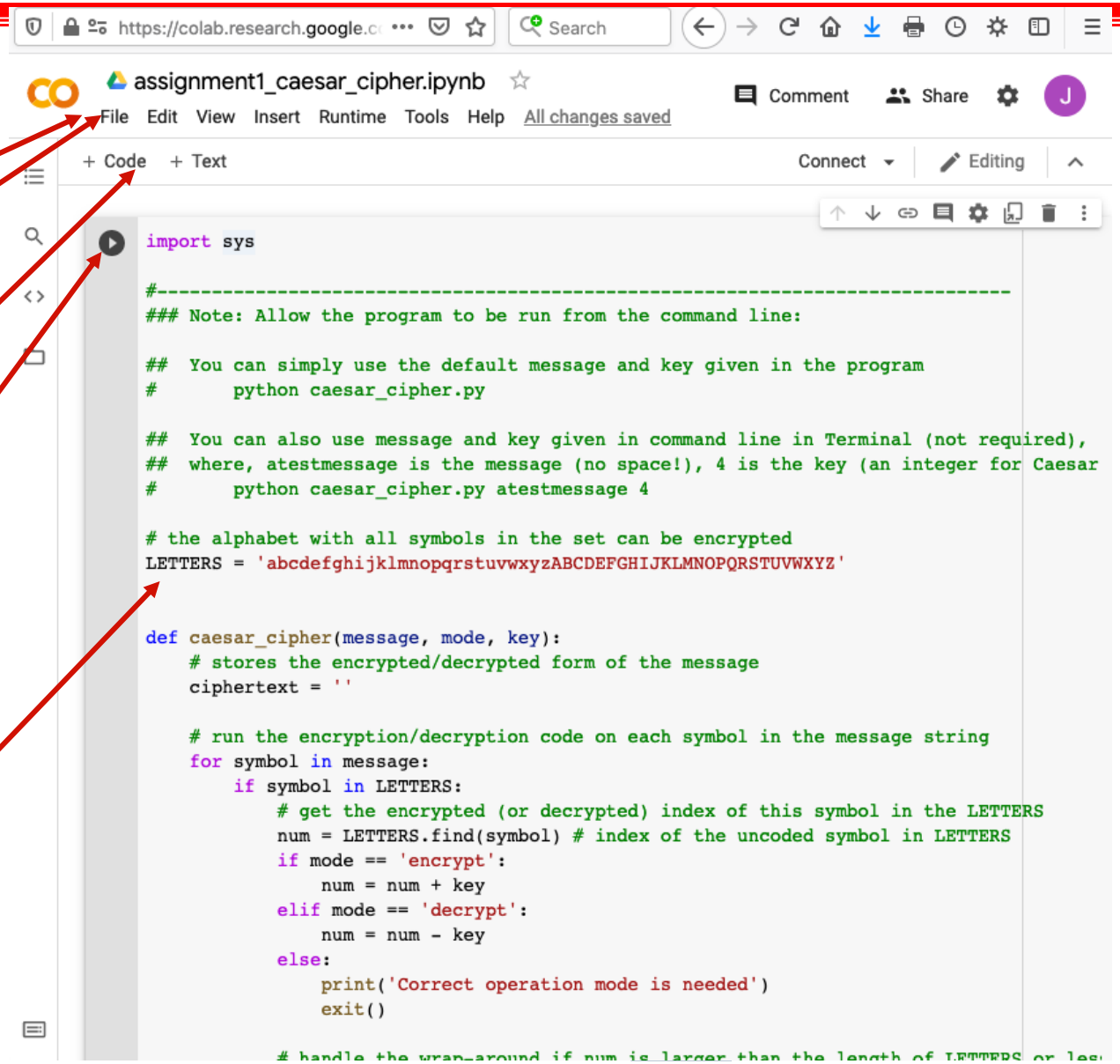
Running Programs in Terminal

- Call python program via the python interpreter in Terminal
- E.g.
  - Firstly, enter the directory where the python program to be run is stored

```
% cd /ce235/python
% python caesar_cipher.py
```

Running Python Program in Google Colab

- East way to run/test python programs
- Create a google account, visit colab.research.google.com
- Create or open .ipynb notebook
- Download as .py or .ipynb files
- Add code or text
- Run code/check output
- Edit/debug the code if needed



3. Python Basics

A Code Sample (in IDLE)

```
x = 34 - 23                    # A comment.
y = “Hello”                    # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”    # String concatenation
print x
print y
```

Enough to Understand the Code

- Indentation matters to code meaning
  - Block structure indicated by indentation
- First assignment to a variable creates it
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.
- Assignment is `=` ; comparison is `==`
- For numbers: `+ - * / %` are as expected
  - Special use of `+` for string concatenation and `%` for string formatting (as in C's printf)
- Logical operators are words (`and, or, not`), *not* symbols
- The basic printing command is `print`

Operators

- + addition
- - subtraction
- / division
- ** exponentiation
- % modulus (remainder after division)
- Comparison operators

Basic Datatypes

- Integers (default for numbers)

```
z = 5 / 2  # Answer 2, integer division
```

- Floats

```
x = 3.456
```

- Strings
  - Can use “” or ‘’ to specify with “abc” == ‘abc’
  - Unmatched can occur within the string: “matt’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:
    “””“a‘b“c”””

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines

- Use a newline to end a line of code
  - Use `\` when must go to next line prematurely
- No braces `{}` to mark blocks of code, use *consistent* indentation
  - First line with *less* indentation is outside of the block
  - First line with *more* indentation starts a nested block
- Colons start of a new block in many constructs, e.g. function definitions, then clauses

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob   Bob   _bob   _2_bob_   bob_2   BoB
```

- There are some reserved words:

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

Naming conventions

The Python community has these recommend-ed naming conventions

- joined_lower for functions, methods and, attributes
- joined_lower or ALL_CAPS for constants
- StudlyCaps for classes
- camelCase only to conform to pre-existing conventions
- Attributes: interface, _internal, __private

Assignment

- You can assign to multiple names at the same time

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

This makes it easy to swap values

```
>>> x, y = y, x
```

- Assignments can be chained

```
>>> a = b = x = 2
```

Type conversion

- int(), float(), str(), and bool() convert to integer, floating point, string, and boolean (True or False) types, respectively

- Example typeconv.py:

```
print 1.0/2.0
print 1/2
print float(1)/float(2)
print int(3.1415926)
print str(3.1415926)
print bool(1)
print bool(0)
```

- Output:

```
0.5
0
0.5
3
3.1415926
True
False
```

4. Sequence types: Tuples, Lists, and Strings

Sequence Types

1. Tuple: (‘john’, 32, [CMSC])
   - A simple *immutable* ordered sequence of items
   - Items can be of mixed types, including collection types
2. Strings: “John Smith”
   - *Immutable*
   - Conceptually very much like a tuple
3. List: [1, 2, ‘john’, (‘up’, ‘down’)]
   - *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
  - Tuples and strings are *immutable*
  - Lists are *mutable*
- The operations shown in this section can be applied to *all* sequence types
  - most examples will just show the operation performed on one

Sequence Types 1

- Define tuples using parentheses and commas

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Define lists are using square brackets and commas

```
>>> li = ["abc", 34, 4.34, 23]
```

- Define strings using quotes (", ', or """").

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Sequence Types 2

- Access individual members of a tuple, list, or string using square bracket "array" notation
- *Note that all are 0 based…*

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

Positive and negative indices

- Index is used to access elements in the sequences

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0

```
>>> t[1]
'abc'
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```

Slicing: return copy of a subset

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* second.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

Negative indices count from end

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Lists are mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
  ['abc', 45, 4.34, 23]
```

- We can change lists *in place*.
- Name *li* still points to the same memory reference when we're done.

The 'in' Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops*

The + Operator

The + operator can be used to produce a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

The * Operator

- The * operator produces a *new* tuple, list, or string that "repeats" the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> "Hello" * 3
'HelloHelloHello'
```

5. If Statements

Booleans: True and False

- >>> type (True)
  <type 'bool'>
  >>> 2+2==5
  False
- Note: True and False are of type bool. *The capitalization is required for the booleans!*

- A boolean expression can be evaluated as True or False. An expression evaluates to False if it is...
  the constant False, the object None, an empty sequence or collection, or a numerical item of value 0
- Everything else is considered True

Comparison operators

- == : is equal to?
- != : not equal to
- > : greater than
- < : less than
- >= : greater than or equal to
- <= : less than or equal to
- is : do two references refer to the same object? (See Chapter 6)

Logical operators

- and, or, not

```
>>> 2+2==5 or 1+1==2
True
>>> 2+2==5 and 1+1==2
False
>>> not(2+2==5) and 1+1==2
True
```

- **Note: We do NOT use &&, ||, !, as in C!**

If statements

- Example ifelse.py

```
if (1+1==2):
    print "1+1==2"
    print "I always thought so!"
else:
    print "My understanding of math is faulty!"
```

- Simple one-line if:

```
if (1+1==2): print “I can add!”
```

elif statement

- Equivalent of “else if” in C
- Example elif.py:

```
x = 3
if (x == 1):
    print "one"
elif (x == 2):
    print "two"
else:
    print "many"
```

6. Iteration Statements

- while loops
- for loops
- range function
- Flow control within loops: break, continue, pass, and the “loop else”

while

- Example whileloop.py

```
i=1
while i < 4:
    print i
    i += 1
```

Output:
1
2
3

for

- Example forloop.py

```
for i in range(3):
    print i,
```

output:
0, 1, 2

- range(n) returns a list of integers from 0 to n-1.
- range(0,10,2) returns a list 0, 2, 4, 6, 8

Flow control within loops

- General structure of a loop:

```
while <statement>      (or for <item> in <object>):
   <statements within loop>
   if <test1>: break          # exit loop now
   if <test2>: continue       # go to top of loop now
   if <test3>: pass           # does nothing!
else:
   <other statements>         # if exited loop without
                              # hitting a break
```

Using the “loop else”

- An else statement after a loop is useful for taking care of a case where an item isn't found in a list. Example: search_items.py:

```
for i in range(3):
    if i == 4:
        print "I found 4!"
        break
    else:
        print "Don't care about",i
else:
    print "I searched but never found 4!"
```

for ... in

- Used with collection data types which can be iterated through ("iterables"):

```
for name in ["Mutasem", "Micah", "Ryan"]:
    if name[0] == "M":
        print name, "starts with an M"
    else:
        print name, "doesn't start with M"
```

Parallel traversals

- If we want to go through 2 lists (more later) in parallel, can use zip:

```
A = [1, 2, 3]
B = [4, 5, 6]
for (a,b) in zip(A,B):
  print a, "*", b, "=", a*b

output:
1 * 4 = 4
2 * 5 = 10
3 * 6  = 18
```

7. Functions

- Defining functions
- Return values
- Local variables
- Built-in functions
- Functions of functions
- Passing lists, dictionaries, and keywords to functions

Functions

- Define them in the file above the point they're used
- Body of the function should be indented consistently (4 spaces is typical in Python)
- Example: square.py

```
def square(n):
    return n*n

print "The square of 3 is ",
print square(3)

Output:
The square of 3 is  9
```

The def statement

- The `def` statement is *executed* (that's why functions have to be defined before they're used)
- `def` creates an object and assigns a name to reference it; the function could be assigned another name, function names can be stored in a list, etc.
- Can put a `def` statement inside an if statement, etc!

More about functions

- Arguments are optional. Multiple arguments are separated by commas.
- If there's no return statement, then “None” is returned.
- Return values can be simple types or tuples.
- Return values may be ignored by the caller.
- Functions are “typeless.”
- They can call with arguments of any type, so long as the operations in the function can be applied to the arguments.
- This is considered a good thing in Python.

Function variables are local

- Variables declared in a function do not exist outside that function
- Example square2.py

```
def square(n):
    m = n*n
    return m

print "The square of 3 is ",
print square(3)
print m
```

Output:

```
  File "./square2.py", line 9, in <module>
    print m
NameError: name 'm' is not defined
```

Scope

- Variables assigned within a function are local to that function call
- Variables assigned at the top of a module are global to that module; there's only "global" within a module
- Within a function, Python will try to match a variable name to one assigned locally within the function;
- if that fails, it will try within enclosing function-defining (def) statements (if appropriate);
- if that fails, it will try to resolve the name in the global scope (but the variable must be declared global for the function to be able to change it).
- If none of these match, Python will look through the list of built-in names

Scope example

- scope.py

```
a = 5                          # global

def func(b):
    c = a + b
    return c

print func(4)                   # gives 4+5=9
print c                         # not defined
```

Scope example

- scope.py

```
a = 5

def func(b):
  global c              # global
  c = a + b
  return c

print func(4)           # gives 4+5=9
print c                 # now it's defined (9)
```

Multiple return values

- Can return multiple values by packaging them into a tuple

```
def onetwothree(x):
    return x*1, x*2, x*3

print onetwothree(3)

3, 6, 9
```

Built-in Functions

- Several useful built-in functions. Example math.py

```
print pow(2,3)
print abs(-14)
print max(1,-5,3,0)

Output:
8
14
3
```

8. File Input/Output (I/O)

Files: Input

fin = open('data', 'r')	Open the file (e.g. with name "data") for input
S = fin.read()	Read the whole file into one String
S = fin.read(N)	Reads N bytes (N >= 1)
L = fin.readlines()	Returns a list of line strings

Files: Output

fout = open(‘data’, ‘w’)	Open the file for writing
fout.write(S)	Writes the string S to file
fout.writelines(L)	Writes each of the strings in list L to file
fout.close()	Manual close

Example

```
import os, sys
fout = open("fileio.txt", "w")
txtLinesOut = ['write a message \n', 'to file']
fout.writelines(txtLinesOut)

# write out lines one by one
# for line in txtLinesOut:
#     fout.write(line)

fout.close()
```

```
fin=open("fileio.txt", "r")
txtLinesIn = fin.readlines()
for line in txtLinesIn:
    print(line)
fin.close()
```