# Financial Programming and Databases

WANG, Zigan (王自干)
Ph.D., Columbia University
Faculty of Business Economics, The University of Hong Kong

# Introduction of Python


python™

# 1. Python Overview

- Python is a high-level, interpreted, interactive and object oriented-scripting language.

- Python was designed to be highly readable which uses English keywords frequently where as other languages use punctuation and it has fewer syntactical constructions than other languages.

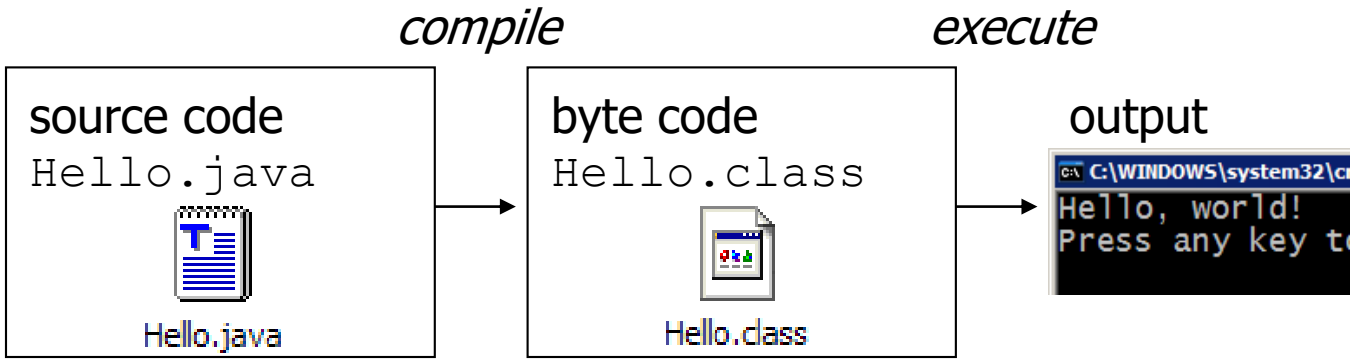
# Who uses Python?

- Yahoo Maps/Groups
- Google
- NASA
- ESRI
- Linux distros
- Me

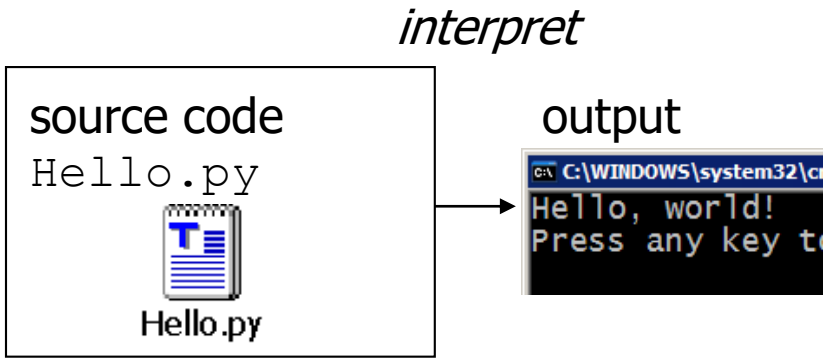
- **Python is Interpreted:** This means that it is processed at runtime by the interpreter and you do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** This means that you can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** This means that Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is Beginner's Language:** Python is a great language for the beginner programmers and supports the development of a wide range of applications, from simple text processing to WWW browsers to games.

# Compiling and interpreting

- Many languages require you to *compile* (translate) your program into a form that the machine understands.




- Python is instead directly *interpreted* into machine instructions.



# History of Python:

- Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.
- Python is copyrighted, Like Perl, Python source code is now available under the GNU General Public License (GPL).
- Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing it's progress.

# Python Features

- **Easy-to-learn:** Python has relatively few keywords, simple structure, and a clearly defined syntax.
- **Easy-to-read:** Python code is much more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's success is that its source code is fairly easy-to-maintain.
- **A broad standard library:** One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:** Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.

# Python Features (cont’d)

- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

# Python Environment

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX etc.)
- Win 9x/NT/2000
- Macintosh (PPC, 68K)
- OS/2
- DOS (multiple versions)
- PalmOS
- Nokia mobile phones
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python has also been ported to the Java and .NET virtual machines.

# 2. Python - Basic Syntax

- **Interactive Mode Programming:**

```
>>> print("Hello, Python!");
```

Hello, Python!

```
>>> 3+4*5;
```

23

# Script Mode Programming

- **Script Mode Programming :**

  Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

  For example, put the following in one test.py, and run,

```
print("Hello, Python!");
print("I love COMP3050!");
```

  The output will be:

```
Hello, Python!
I love COMP3050!
```

# Python Identifiers:

- A Python identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).
- Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus **Manpower** and **manpower** are two different identifiers in Python.

# Python Identifiers (cont’d)

- Here are following identifier naming convention for Python:
  - Class names start with an uppercase letter and all other identifiers with a lowercase letter.
  - Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private.
  - Starting an identifier with two leading underscores indicates a strongly private identifier.
  - If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

# Reserved Words:

**Keywords contain lowercase letters only.**

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

# Lines and Indentation:

- One of the first caveats programmers encounter when learning Python is the fact that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

```
if True:
    print("Answer");
    print("True");
else:
    print("Answer");
    print("False")
```

# Multi-Line Statements:

- Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

```
total = item_one + \
        item_two + \
        item_three
```

- Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example:

```
days = ['Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday']
```

# Quotation in Python:

- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes can be used to span the string across multiple lines. For example, all the following are legal:

```
word = 'word'

sentence = "This is a sentence."

paragraph = """This is a paragraph. It is made up
    of multiple lines and sentences."""
```

# Comments in Python:

- A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment, and the Python interpreter ignores them.

# Using Blank Lines:

- A line containing only whitespace, possibly with a comment, is known as a blank line, and Python totally ignores it.
- In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

# Multiple Statements on a Single Line:

- The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

# Multiple Statement Groups as Suites:

- Groups of individual statements making up a single code block are called **suites** in Python.

  Compound or complex statements, such as if, while, def, and class, are those which require a header line and a suite.

  Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines which make up the suite.

```
if expression :
  suite
elif expression :
  suite
else :
 suite
```

# 3. Python - Variable Types

- Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.
- Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

# Assigning Values to Variables:

- Python variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

```
counter = 100 # An integer assignment
miles = 1000.0 # A floating point
name = "John" # A string
print(counter)
print(miles)
print(name)
```

# Multiple Assignment:

- You can also assign a single value to several variables simultaneously. For example:

```
a = b = c = 1
a, b, c = 1, 2, "john"
```

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# Install programming language distribution

- What is a Python distribution?
- https://wiki.python.org/moin/PythonDistributions

- What is a Python version?
- https://en.wikipedia.org/wiki/History_of_Python#Version_release_dates

# Anaconda

- Where to download and install it?
- https://www.anaconda.com/download/


- Anaconda consists of the following packages:
- https://docs.anaconda.com/anaconda/packages/pkg-docs

# Conda commands

- Windows start -> anaconda prompt

- https://conda.io/docs/user-guide/tasks/manage-python.html

```
(D:\Python\Anaconda3) C:\Users\Zigan Wang>conda -V
conda 4.3.21
```

```
C:\Users\Zigan Wang>conda install spyder
```

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# Spyder

- What is an IDE?
- Why Spyder?
  - It’s light
- I don’t use Vim or Emacs

# Spyder installation

- https://pythonhosted.org/spyder/index.html

- No need to install separately if you have Anaconda installed

Spyder (Python 2.7)

File Edit Search Source Run Debug Consoles Projects Tools View Help

C:\Users\tmpuser

Editor - D:\Files\Documents\HKU Materials\HKU Research\Lin - Pollution and Brain Drain\2017.12\Table 7\20170803\20171203_parse.py

20171217_combine.py | 20171217_concat.py | 20171203_parse.py | createSubsiFolders_multipr_tar1.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  from html.parser import HTMLParser
5  from html.entities import name2codepoint
6  import pandas as pd
7
8
9
10
11     def __init__(self):
12         super(MyHTMLParser, self).__init__()
13         self.s = ''
14
15     def handle_data(self, data):
16         #print(data)
17         self.s = self.s + data + ',,,'
18
19     def show(self):
20         return self.s
21
22 """
23     def handle_starttag(self, tag, attrs):
24         print('<%s>' % tag)
25
26     def handle_endtag(self, tag):
27         print('</%s>' % tag)
28
29     def handle_startendtag(self, tag, attrs):
30         print('<%s/>' % tag)
31
32     def handle_comment(self, data):
33         print('<!--', data, '-->')
34
35     def handle_entityref(self, name):
36         print('&%s;' % name)
37
38     def handle_charref(self, name):
39         print('&#%s;' % name)
40 """
41
42
43 #currentPath = os.getcwd()
44 # Folder: ttest_results
45 #path = currentPath + '/ttest_results'
46
47 path = r'D:\Files\Documents\HKU Materials\HKU Research\Lin - Pollution and Brain Drain\2017.12\Table 7\20170803\20180116_tab
48 flag = True
49 for sample in ['_NmTrOp05m.xml', '_no_NmTrOp05m.xml']:
50     for item in ['ceo','chairman','executive','all']:
51         for i in os.listdir(path):
52             if item + sample in i:
53                 filename = path + '/' + i
54                 print(filename)
55                 with open(filename, 'r') as f:
56                     parser = MyHTMLParser()
57                     parser.feed(f.readline())
```

Code analysis

'html.entities.name2codepoint' imported but unused

File explorer

| Name | Size | Type | Date Modified |
|---|---|---|---|
| .astropy | | File Folder | 8/17/2017 1:46 PM |
| .cisco | | File Folder | 8/7/2017 11:43 AM |
| .eclipse | | File Folder | 8/17/2017 1:46 PM |
| .ipynb_checkpoints | | File Folder | 10/18/2017 3:55 PM |
| .ipython | | File Folder | 8/29/2017 3:48 PM |
| .jupyter | | File Folder | 9/5/2017 1:15 PM |
| .matplotlib | | File Folder | 1/16/2018 9:12 PM |
| .oracle_jre_usage | | File Folder | 8/17/2016 12:42 PM |
| .oss-browser | | File Folder | 10/31/2017 7:51 PM |
| .p2 | | File Folder | 9/10/2017 4:37 PM |
| .spyder | | File Folder | 1/16/2018 9:12 PM |
| .tooling | | File Folder | 8/17/2017 12:02 PM |
| Desktop | | File Folder | 10/30/2017 6:31 PM |

Variable explorer | File explorer | Help

IPython console

Console 1/A

```
Python 2.7.13 |Anaconda 4.3.1 (64-bit)| (default, Dec 19 2016, 13:29:36) [MSC v.1500 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: runfile('D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table
7/20170803/20171203_parse.py', wdir='D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table
7/20170803')
Traceback (most recent call last):

  File "<ipython-input-1-998bf94085c1>", line 1, in <module>
    runfile('D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803/20171203_parse.py',
wdir='D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803')

  File "D:\Programs\Anaconda2\lib\site-packages\spyder\utils\site\sitecustomize.py", line 866, in runfile
    execfile(filename, namespace)

  File "D:\Programs\Anaconda2\lib\site-packages\spyder\utils\site\sitecustomize.py", line 87, in execfile
    exec(compile(scripttext, filename, 'exec'), glob, loc)

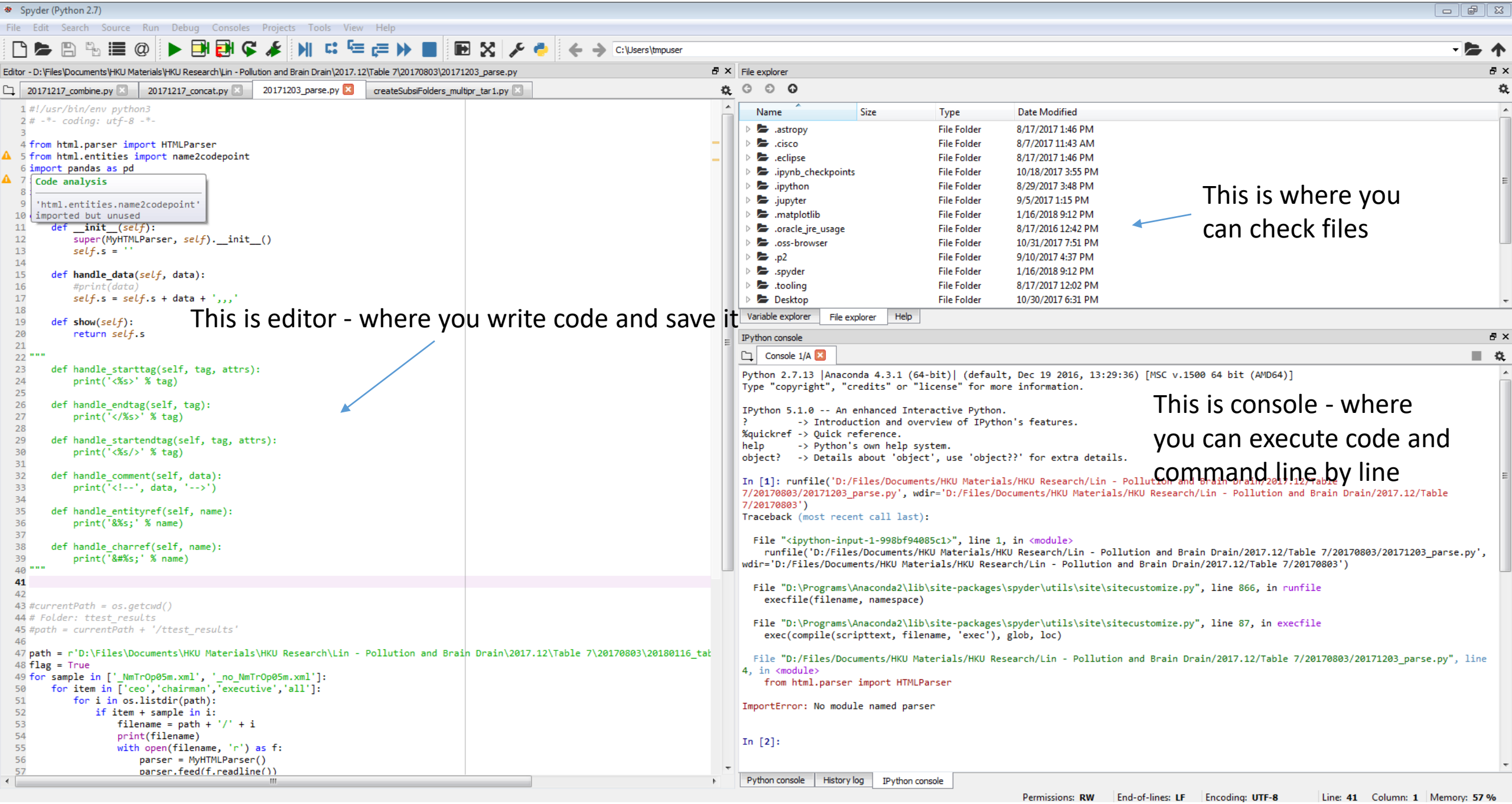
  File "D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803/20171203_parse.py", line
4, in <module>
    from html.parser import HTMLParser

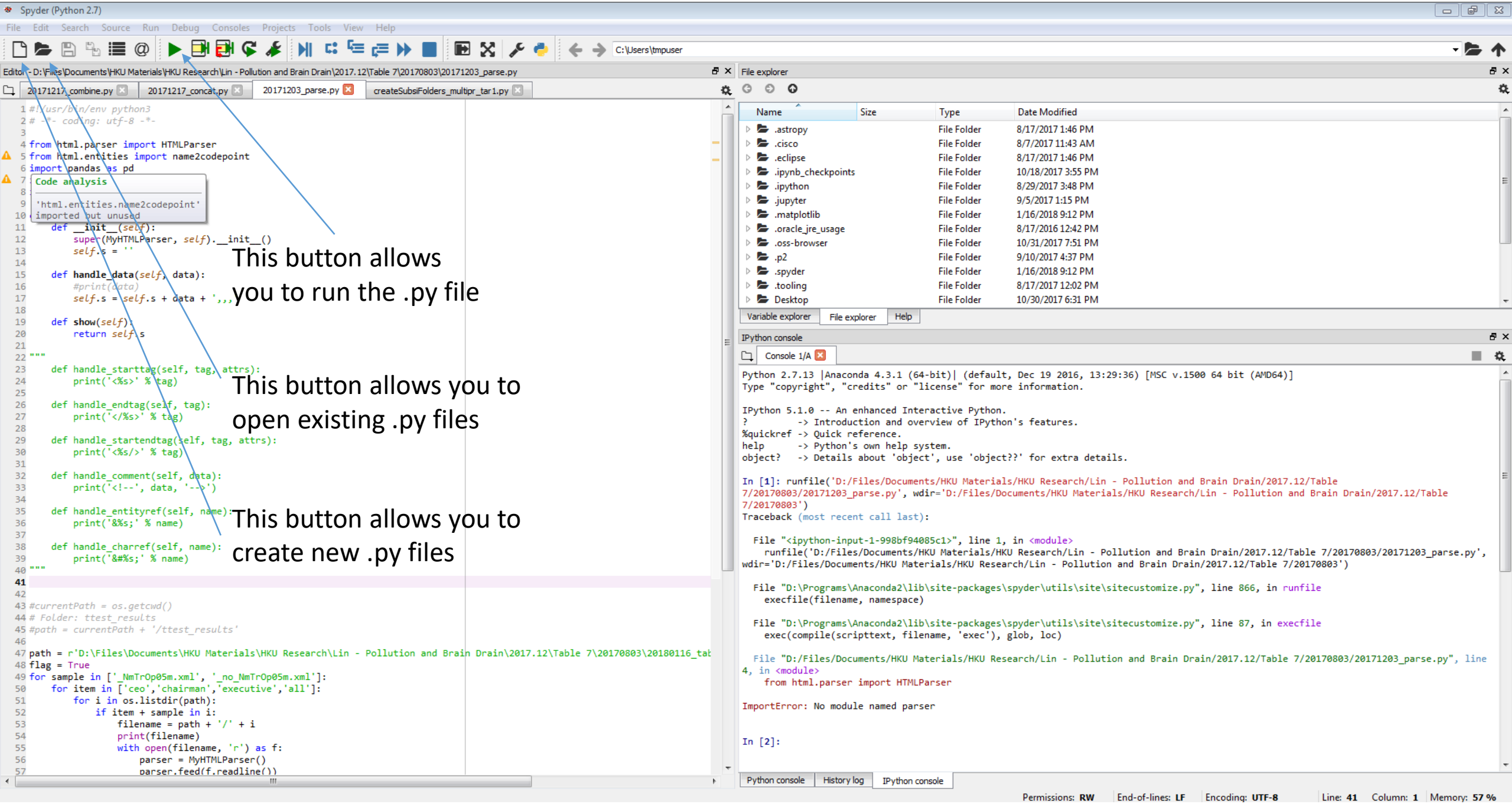
ImportError: No module named parser


In [2]:
```

Python console | History log | IPython console

Permissions: RW | End-of-lines: LF | Encoding: UTF-8 | Line: 41 | Column: 1 | Memory: 57 %


Spyder (Python 2.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\tmpuser
Editor - D:\Files\Documents\HKU Materials\HKU Research\Lin - Pollution and Brain Drain\2017.12\Table 7\20170803\20171203_parse.py
20171217_combine.py
20171217_concat.py
20171203_parse.py
createSubsiFolders_multipr_tar1.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from html.parser import HTMLParser
from html.entities import name2codepoint
import pandas as pd
Code analysis
'html.entities.name2codepoint' imported but unused
class MyHTMLParser(HTMLParser):
def __init__(self):
super(MyHTMLParser, self).__init__()
self.s = ''
def handle_data(self, data):
#print(data)
self.s = self.s + data + ',,,'
def show(self):
return self.s
"""
def handle_starttag(self, tag, attrs):
print('<%s>' % tag)
def handle_endtag(self, tag):
print('</%s>' % tag)
def handle_startendtag(self, tag, attrs):
print('<%s/>' % tag)
def handle_comment(self, data):
print('<!--', data, '-->')
def handle_entityref(self, name):
print('&%s;' % name)
def handle_charref(self, name):
print('&#%s;' % name)
"""
#currentPath = os.getcwd()
# Folder: ttest_results
#path = currentPath + '/ttest_results'
path = r'D:\Files\Documents\HKU Materials\HKU Research\Lin - Pollution and Brain Drain\2017.12\Table 7\20170803\20180116_tab
flag = True
for sample in ['_NmTrOp05m.xml', '_no_NmTrOp05m.xml']:
for item in ['ceo','chairman','executive','all']:
for i in os.listdir(path):
if item + sample in i:
filename = path + '/' + i
print(filename)
with open(filename, 'r') as f:
parser = MyHTMLParser()
parser.feed(f.readline())
This is editor - where you write code and save it
File explorer
Name
Size
Type
Date Modified
.astropy File Folder 8/17/2017 1:46 PM
.cisco File Folder 8/7/2017 11:43 AM
.eclipse File Folder 8/17/2017 1:46 PM
.ipynb_checkpoints File Folder 10/18/2017 3:55 PM
.ipython File Folder 8/29/2017 3:48 PM
.jupyter File Folder 9/5/2017 1:15 PM
.matplotlib File Folder 1/16/2018 9:12 PM
.oracle_jre_usage File Folder 8/17/2016 12:42 PM
.oss-browser File Folder 10/31/2017 7:51 PM
.p2 File Folder 9/10/2017 4:37 PM
.spyder File Folder 1/16/2018 9:12 PM
.tooling File Folder 8/17/2017 12:02 PM
Desktop File Folder 10/30/2017 6:31 PM
Variable explorer
File explorer
Help
This is where you can check files
IPython console
Console 1/A
Python 2.7.13 |Anaconda 4.3.1 (64-bit)| (default, Dec 19 2016, 13:29:36) [MSC v.1500 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
IPython 5.1.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
In [1]: runfile('D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803/20171203_parse.py', wdir='D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803')
Traceback (most recent call last):
File "<ipython-input-1-998bf94085c1>", line 1, in <module>
runfile('D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803/20171203_parse.py', wdir='D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803')
File "D:\Programs\Anaconda2\lib\site-packages\spyder\utils\site\sitecustomize.py", line 866, in runfile
execfile(filename, namespace)
File "D:\Programs\Anaconda2\lib\site-packages\spyder\utils\site\sitecustomize.py", line 87, in execfile
exec(compile(scripttext, filename, 'exec'), glob, loc)
File "D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803/20171203_parse.py", line 4, in <module>
from html.parser import HTMLParser
ImportError: No module named parser
In [2]:
This is console - where you can execute code and command line by line
Python console
History log
IPython console
Permissions: RW
End-of-lines: LF
Encoding: UTF-8
Line: 41
Column: 1
Memory: 57 %


Spyder (Python 2.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\tmpuser
Editor - D:\Files\Documents\HKU Materials\HKU Research\Lin - Pollution and Brain Drain\2017.12\Table 7\20170803\20171203_parse.py
20171217_combine.py
20171217_concat.py
20171203_parse.py
createSubsiFolders_multipr_tar1.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from html.parser import HTMLParser
from html.entities import name2codepoint
import pandas as pd
Code analysis
'html.entities.name2codepoint' imported but unused
class MyHTMLParser(HTMLParser):
def __init__(self):
super(MyHTMLParser, self).__init__()
self.s = ''
def handle_data(self, data):
#print(data)
self.s = self.s + data + ',,,'
def show(self):
return self.s
"""
def handle_starttag(self, tag, attrs):
print('<%s>' % tag)
def handle_endtag(self, tag):
print('</%s>' % tag)
def handle_startendtag(self, tag, attrs):
print('<%s/>' % tag)
def handle_comment(self, data):
print('<!--', data, '-->')
def handle_entityref(self, name):
print('&%s;' % name)
def handle_charref(self, name):
print('&#%s;' % name)
"""
#currentPath = os.getcwd()
# Folder: ttest_results
#path = currentPath + '/ttest_results'
path = r'D:\Files\Documents\HKU Materials\HKU Research\Lin - Pollution and Brain Drain\2017.12\Table 7\20170803\20180116_tab
flag = True
for sample in ['_NmTrOp05m.xml', '_no_NmTrOp05m.xml']:
for item in ['ceo','chairman','executive','all']:
for i in os.listdir(path):
if item + sample in i:
filename = path + '/' + i
print(filename)
with open(filename, 'r') as f:
parser = MyHTMLParser()
parser.feed(f.readline())
This button allows you to run the .py file
This button allows you to open existing .py files
This button allows you to create new .py files
File explorer
Name Size Type Date Modified
.astropy File Folder 8/17/2017 1:46 PM
.cisco File Folder 8/7/2017 11:43 AM
.eclipse File Folder 8/17/2017 1:46 PM
.ipynb_checkpoints File Folder 10/18/2017 3:55 PM
.ipython File Folder 8/29/2017 3:48 PM
.jupyter File Folder 9/5/2017 1:15 PM
.matplotlib File Folder 1/16/2018 9:12 PM
.oracle_jre_usage File Folder 8/17/2016 12:42 PM
.oss-browser File Folder 10/31/2017 7:51 PM
.p2 File Folder 9/10/2017 4:37 PM
.spyder File Folder 1/16/2018 9:12 PM
.tooling File Folder 8/17/2017 12:02 PM
Desktop File Folder 10/30/2017 6:31 PM
Variable explorer File explorer Help
IPython console
Console 1/A
Python 2.7.13 |Anaconda 4.3.1 (64-bit)| (default, Dec 19 2016, 13:29:36) [MSC v.1500 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
IPython 5.1.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
In [1]: runfile('D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803/20171203_parse.py', wdir='D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803')
Traceback (most recent call last):
File "<ipython-input-1-998bf94085c1>", line 1, in <module>
runfile('D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803/20171203_parse.py', wdir='D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803')
File "D:\Programs\Anaconda2\lib\site-packages\spyder\utils\site\sitecustomize.py", line 866, in runfile
execfile(filename, namespace)
File "D:\Programs\Anaconda2\lib\site-packages\spyder\utils\site\sitecustomize.py", line 87, in execfile
exec(compile(scripttext, filename, 'exec'), glob, loc)
File "D:/Files/Documents/HKU Materials/HKU Research/Lin - Pollution and Brain Drain/2017.12/Table 7/20170803/20171203_parse.py", line 4, in <module>
from html.parser import HTMLParser
ImportError: No module named parser
In [2]:
Python console History log IPython console
Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 41 Column: 1 Memory: 57 %

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# Hello World

- Hello World

```
8 print('Hello World!')
```

- String
- Boolean
- Number: float, integer
- No need to define a variable’s data type in Python

# Operators

- Python has many operators. Some examples are:

`+, -, *, /, %, >, <, ==`

- Operators perform an action on one or more operands. Some operators accept operands before and after themselves:

`operand1 + operand2`, or `3 + 5`

- When operators are evaluated, they perform action on their operands, and produce a new value.

# Example Expression Evaluations

- An expression is any set of values and operators that will produce a new value when evaluated. Here are some examples, along with the new value they produce when evaluated:

```
 5 + 10                    produces15
"Hi" + " " + "Jay!"        produces"Hi Jay!"
10 / (2+3)                 produces    2
10 > 5                     producesTrue
10 < 5                     producesFalse
10 / 3.5                   produces2.8571428571
10 / 3                     produces3.3333333333
10 // 3                    produces3
10 % 3                     produces1
```

# List of Operators: +, -, *, /, //,<, >, <=, >=, ==, %

- Some operators should be familiar from the world of mathematics such as Addition (+), Subtraction (-), Multiplication (*), and Division (/).
- Python also has comparison operators, such as Less-Than (<), Greater-Than (>), Less-Than-or-Equal(<=), Greater-Than-or-Equal (>=), and Equality-Test (==). These operators produce a True or False value.
- A less common operator is the Modulo operator (%), which gives the remainder of an integer division. 10 floor divided (//) by 3 is 3 with a remainder of 1:

`10 / 3` produces 3.33333, `10 // 3` produces 3

while `10 % 3` produces 1

# DANGER! Operator Overloading!

- NOTE! Some operators will work in a different way depending upon what their operands are. For example, when you add two numbers you get the expected result: `3 + 3` produces 6.
- But if you "add" two or more strings, the + operator produces a concatenated version of the strings: `"Hi" + "Jay"` produces "HiJay"
- Multiplying strings by a number repeats the string!

`"Hi Jay" * 3` produces "Hi JayHi JayHiJay"

- The modulo operator also works differently with strings:

`"test %f" % 34` produces "test 34.000"

# Data Types

- In Python, all data has an associated data "Type".
- You can find the "Type" of any piece of data by using the type() function:

`type( "Hi!")` produces <type 'str'>

`type( True )` produces <type 'bool'>

`type( 5)` produces <type 'int'>

`type(5.0)` produces <type 'float'>

- Note that python supports two different types of numbers, Integers (int) and Floating point numbers (float). Floating Point numbers have a fractional part (digits after the decimal place), while Integers do not!

# Effect of Data Types on Operator Results

- Math operators work differently on Floats and Ints:
  - int + int produces an int
  - int + float or float + int produces a float
- For division, both int/int and float/float produce a float in Python 3

  `10 / 3` and `10.0 / 3.0` produce 3.3333333
- For floor division, fractional part of result is truncated and int//int produces an int

  `10 // 3` produces 3
- Other operators work differently on different data types: + (addition) will add two numbers, but concatenate strings.

# Simple Data types in Python

- The simple data types in Python are:
- Numbers
  - int – Integer: -5, 10, 77
  - float – Floating Point numbers: 3.1457, 0.34
- bool – Booleans (True or False)
- Strings are a more complicated data type (called Sequences) that we will discuss more later. They are made up of individual letters (strings of length 1)

# Type Conversion

- Data can sometimes be converted from one type to another. For example, the string "3.0" is equivalent to the floating point number 3.0, which is equivalent to the integer number 3
- Functions exist which will take data in one type and return data in another type.
  - `int()` - Converts compatible data into an integer. This function will truncate floating point numbers
  - `float()` - Converts compatible data into a float.
  - `str()` - Converts compatible data into a string.
- Examples:

`int(3.3)` produces 3

`str(3.3)` produces "3.3"

`float(3)` produces 3.0

`float("3.5")` produces 3.5

`int("7")` produces 7

`int("7.1")` produces 7

`float("Test")` Throws an ERROR!

# Variables

- Variables are names that can point to data.
- They are useful for saving intermediate results and keeping data organized.
- The assignment operator (=) assigns data to variables.
  - Don't confuse the assignment operator (single equal sign, =) with the Equality-Test operator (double equal sign, ==)
- Variable names can be made up of letters, numbers and underscores (_), and must start with a letter.

# Variables

- When a variable is evaluated, it produces the value of the data that it points to.
- For example:

`myVariable = 5`

`myVariable` produces 5

`myVariable + 10` produces 15

- You MUST assign something to a variable (to create the variable name) before you try to use (evaluate) it.

# Program Example

- Find the area of a circle given the radius:

- `Radius = 10`
- `pi = 3.14159`
- `area = pi * Radius * Radius`
- `print(area)`

- will print(314.15) to the screen.

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# Create a file

- Create one .txt file

```
10 f = open('helloworld.txt','w')
11 f.write('Hello World!')
12 f.close()
```

- open is a function with two parameters
- write is a function with one parameter
- close is also a function
- write and close are both attribute of a class (dog.bark, dog.walk, dog.bite)

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# Outline

- Compound Data Types:
  Strings, Tuples, Lists & Dictionaries
- Immutable types:
  - Strings
  - Tuples
- Accessing Elements
- Cloning Slices
- Mutable Types:
  - Lists
  - Dictionaries
- Aliasing vs Cloning

Thanks to Jay Summet for the materials

# Data Types

- Strings – Enclosed in Quotes, holds characters, (immutable):
  "This is a String "
- Tuples – Values separated by commas, (usually enclosed by parenthesis) and can hold any data type (immutable):
  (4 , True, "Test", 34.8)
- Lists – Enclosed in square brackets, separated by commas, holds any data type (mutable):
  [4, True, "Test", 34.8]
- Dictionaries – Enclosed in curly brackets, elements separated by commas, key : value pairs separated by colons, keys can be any immutable data type, values can be any data type:
  { 1 : "I", 2 : "II", 3 : "III", 4 : "IV", 5 : "V", }

# Immutable Types

- Strings and Tuples are immutable, which means that once you create them, you can not change them.
- The assignment operator (=) can make a variable point to strings or tuples just like simple data types:
  myString = "This is a test"
  myTuple = (1,45.8, True, "String Cheese")
- Strings & Tuples are both *sequences*, which mean that they consist of an ordered set of elements, with each element identified by an index.

# STRING

# Accessing Elements

- `myString = "This is a test."`
- You can access a specific element using an integer *index* which counts from the front of the sequence (starting at ZERO!)

```
myString[0] produces 'T'
myString[1] produces 'h'
myString[2] produces 'i'
myString[3] produces 's'
```

- The `len()` function can be used to find the length of a sequence. Remember, the last element is the length minus 1, because counting starts at zero!

```
myString[ len(myString) - 1] produces '.'
```

# Counting Backwards

- `myString = "This is a test."`
- As a short cut (to avoid writing `len(myString)` ) you can simply count from the end of the string using negative numbers:

`myString[ len(myString) - 1 ]` produces '.'
`myString[-1]` also produces '.'
`myString[-2]` produces 't'
`myString[-3]` produces 's'
`myString[-4]` produces 'e'

# Index out of Range!

- `myString = "This is a test."`
- Warning! If you try to access an element that does not exist, Python will throw an error!
  `myString[5000]` produces An ERROR!
  `myString[-100]` produces An ERROR!

# Traversals (Go over from the beginning to the end)

- Many times you need to do something to every element in a sequence. (e.g. Check each letter in a string to see if it contains a specific character.)
- Doing something to every element in a sequence is so common that it has a name: a Traversal.
- You can do a traversal using a while loop as follows:

```
index = 0
while ( index < len (myString) ):
   if myString[index] == 'J':
     print("Found a J!")
   index = index + 1
```

# Easy Traversals – The FOR Loop

- Python makes traversals easy with a FOR loop:

for ELEMENT_VAR in SEQUENCE:

    STATEMENT

    STATEMENT

- A for loop automatically assigns each element in the sequence to the (programmer named) ELEMENT_VAR, and then executes the statements in the block of code following the for loop once for each element.
- Here is an example equivalent to the previous WHILE loop:

```
for myVar in myString:
        if myVar == 'J':
            print("I found a J!")
```

- Note that it takes care of the indexing variable for us.

# Grabbing Slices from a Sequence

- The slice operator will clip out part of a sequence. It looks a lot like the bracket operator, but with a colon that separates the “start” and “end” points.

```
SEQUENCE_VAR [ START : END ]
myString = "This is a test."
```

`myString[0:2]` produces 'Th' (0, 1, but NOT 2)

`myString[3:6]` produces 's i” (3-5, but NOT 6)

`myString[ 1 : 4 ]` produces ?

`myString[ 5 : 7 ]` produces ?

# Slices – Default values for blanks

- `SEQUENCE_VAR [ START : END ]`
- `myString = "This is a test."`

- If you leave the “start” part blank, it assumes you want zero.

`myString[ : 2]` produces 'Th' (0,1, but NOT 2)

`myString[ : 6]` produces 'This i' (0-5, but NOT 6)

- If you leave the “end” blank, it assumes you want until the end of the string

`myString[ 5 : ]` produces 'is a test.' (5 – end)

`myString [ : ]` produces 'This is a test.' (0-end)

# Using Slices to make clones (copies)

- You can assign a slice from a sequence to a variable.
- The variable points at a copy of the data.

```
myString = "This is a test."

hisString = myString [ 1 : 4 ]

isString = myString [ 5 : 7 ]

testString = myString [10 : ]
```

# TUPLE

# Tuples

- Tuples can hold any type of data!
- You can make one by separating some values with comas. Convention says that we enclose them with parenthesis to make the beginning and end of the tuple explicit, as follows:

```
(4, True, "Test", 14.8)
```

- NOTE: Parenthesis are being overloaded here, which make the commas very important!
  - (4) is NOT a tuple (it's a 4 in parenthesis)
  - (4,) IS a tuple of length 1 (note the trailing comma)
  - A tuple with a single element ***must*** have a comma inside the parentheses:
    - **a = (11,)**
- Tuples are good when you want to group a few variables together (firstName, lastName) (x,y)

# Examples

- `>>> mytuple = (11, 22, 33)`
- `>>> mytuple[0]`
  `11`
- `>>> mytuple[-1]`
  `33`
- `>>> mytuple[0:1]`
  `(11,)`
- **The comma is required!**

# Tuples are immutable

- `>>> mytuple = (11, 22, 33)`
- `>>> saved = mytuple`
- `>>> mytuple += (44,)`
- `>>> mytuple`
  `(11, 22, 33, 44)`
- `>>> saved`
  `(11, 22, 33)`

# Sorting tuples

- `>>> atuple = (33, 22, 11)`
- `>>> atuple.sort()`

```
Traceback (most recent call last):
…
AttributeError:
'tuple' object has no attribute 'sort'
```

- `>>> atuple = sorted(atuple)`
- `>>> atuple`

```
[11, 22, 33]
```

**Tuples are immutable!**

**sorted( ) returns a list!**

# Most other things work!

- **>>> atuple = (11, 22, 33)**
- **>>> len(atuple)**
  **3**
- **>>> 44 in atuple**
  **False**
- **Tuple -> list:**
- **>>> [i for i in atuple]**
  **[11, 22, 33]**

# The reverse does not work

- **`>>> alist = [11, 22, 33]`**
- **`>>> (i for i in alist)`**
  **`<generator object <genexpr> at 0x02855DA0>`**
  - ***Does not work!***

# Converting sequences into tuples

- `>>> alist = [11, 22, 33]`
- `>>> atuple = tuple(alist)`
- `>>> atuple`
  `(11, 22, 33)`
- `>>> newtuple = tuple('Hello World!')`
- `>>> newtuple`
  `('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!')`

# Using Tuples to return multiple pieces of data!

- Tuples can also allow you to return multiple pieces of data from a function:

```
def area_volume_of_cube(sideLength):
    area = 6 * sideLength * sideLength
    volume = sideLength * sideLength * sideLength
    return  area, volume

myArea, myVolume = area_volume_of_cube( 6 )
```

- Note that in this example we left out the (optional) parenthesis from the tuple `(area,volume)`!
- You can also use tuples to swap the contents of two variables: `a,b = b,a`

# Tuples are sequences!

- Because tuples are sequences, we can access them using the bracket operator just like strings.

```
myTuple = ( 4,48.8,True, "Test")
```

```
myTuple[ 0 ]      produces  4
myTuple[ 1 ]      produces  48.8
myTuple[ 2 ]      produces  True
myTuple[ -1 ]     produces  'Test'
```

# “Changing” Immutable data types

- Immutable data types can not be changed after they are created. Examples include Strings and Tuples.
- Although you can not change an immutable data type, you can create a new variable that has the changes you want to make.
- For example, to capitalize the first letter in this string:

```
myString = "all lowercase."
myNewString = "A" + myString[1:]
```

- We have concatenated two strings (a string with the capital letter A of length 1, and a clone of myString missing the first, lowercase, 'a').

# Tuples are Comparable

- The comparison operators work with tuples and other sequences If the first item is equal, Python goes on to the next element,  and so on, until it finds elements that differ.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True
>>> ( 'Jones', 'Sally') > ('Adams', 'Sam')
True
```

# LIST

# Changing mutable data types

- Mutable data types can  be changed after they are created. Examples include Lists and Dictionaries.
- These changes happen “in place”.

```
myList = ['l', 'o', 'w', 'e', 'r', 'c', 'a', 's', 'e']
myList[0] = 'L'
```

`print(myList)` produces:

```
[ 'L', 'o', 'w', 'e', 'r', 'c', 'a', 's', 'e']
```

- Note that we are using the indexing operator [brackets] in the same way that we always have, but when combined with the assignment operator (=) it replaces the element at index 0 with the 'L'.

# Lists – like strings & tuples, but mutable!

- Lists are a mutable data type that you can create by enclosing a series of elements in square brackets separated by commas. The elements do not have to be of the same data type:

```
myList = [ 23, True, 'Cheese', 3.1459 ]
```

- Unlike Strings and tuples, individual elements in a list can be modified using the assignment operator. After the following commands:

```
myList[0] = True
myList[1] = 24
myList[3] = 'Boo'
```

myList contains: `[ True, 24, 'Cheese', 'Boo' ]`

# Different Elements, Different Types

- Unlike strings, which contain nothing but letters, list elements can be of any data type.

```
myList = [ True, 24, 'Cheese', 'Boo' ]

for eachElement in myList:
    print(type(eachElement))
```

produces:

```
<type 'bool'>
<type 'int'>
<type 'str'>
<type 'str'>
```

# List Restrictions and the append method

- Just like a string, you can't access an element that doesn't exist. You also can not assign a value to an element that doesn't exist.
- Lists do have some helper methods. Perhaps the most useful is the append method, which adds a new value to the end of the list:

```
myList = [ True, 'Boo', 3]
myList.append(5)
print(myList)
[ True, 'Boo', 3, 5]
```

# Difference between append and extend

```
In [11]: myList = [ True, 'Boo', 3]

In [12]: myList.append([5,6])

In [13]: myList
Out[13]: [True, 'Boo', 3, [5, 6]]

In [14]: myList = [ True, 'Boo', 3]

In [15]: myList.extend([5,6])

In [16]: myList
Out[16]: [True, 'Boo', 3, 5, 6]
```

# DICTIONARY

# And now, for something completely different!

- Dictionaries are an *associative* data type.
- Unlike sequences, they can use ANY immutable data type as an index.
- Dictionaries will associate a key (any immutable data), with a value (any data).
- For example, we can use a dictionary to associate integer keys (the numbers 1-5) with strings (their Roman numeral representation) like this:
- ```
  arabic2roman = { 1 : "I", 2 : "II", 3 : "III", 4 : "IV", 5 : "V" }
  ```
  ```

# Dictionary example: Accessing

- `arabic2roman = { 1 : "I", 2 : "II", 3 : "III", 4 : "IV", 5 : "V" }`
- You can retrieve a value from a dictionary by indexing with the associated key:

  `arabic2roman[1]` produces 'I'
  `arabic2roman[5]` produces 'V'
  `arabic2roman[4]` produces 'IV'

# Reassigning & Creating new Key/Value associations

- `arabic2roman = { 1 : "I", 2 : "II", 3 : "III", 4 : "IV", 5 : "V" }`

- You can assign new values to existing keys:
  `arabic2roman[1] = 'one'`
  now:
  `arabic2roman[1]` produces 'one'

- You can create new key/value pairs:
  `arabic2roman[10]= 'X'`
  now
  `arabic2roman[10]` produces 'X'

# Dictionaries and default values with the get method

- If you use an index that does not exist in a dictionary, it will raise an exception (ERROR!)
- But, you can use the get( INDEX, DEFAULT) method to return a default value if the index does not exist. For example:

```
arabic2roman.get(1, "None")
        produces    'I'
arabic2roman.get(5, "None")
        produces    'V'
arabic2roman.get(500, "None")
        produces    'None'
arabic2roman.get("test", "None")
        produces    'None'
```

# Dictionaries (I)

- Store ***pairs*** of entries called ***items***
  **{ 'CS' : '743-713-3350', 'UHPD' : '713-743-3333'}**
- Each pair of entries contains
  - A ***key***
  - ***A value***
- Key and values are separated by a colon
- Paris of entries are separated by commas
- Dictionary is enclosed within curly braces

# Usage

- Keys must be ***unique*** within a dictionary
  - No ***duplicates***
- If we have

```
age = {'Alice' : 25, 'Bob' :28}
```

then

```
age['Alice'] is 25
```

and

```
age[Bob'] is  28
```

# Dictionaries are mutable

- `>>> age = {'Alice' : 25, 'Bob' : 28}`
- `>>> saved = age`
- `>>> age['Bob'] = 29`
- `>>> age`
  `{'Bob': 29, 'Alice': 25}`
- `>>> saved`
  `{'Bob': 29, 'Alice': 25}`

# Keys must be unique

- **>>> age = {'Alice' : 25, 'Bob' : 28, 'Alice' : 26}**
- **>>> age**
  **{'Bob': 28, 'Alice': 26}**

# Displaying contents

- **>>> age = {'Alice' : 25, 'Carol': 'twenty-two'}**
- **>>> age.items()**
  **dict_items([ ('Alice', 25), ('Carol', 'twenty-two')])**
- **>>> age.keys()**
  **dict_keys([ 'Alice', 'Carol'])**
- **age.values()**
  **dict_values([28, 25, 'twenty-two'])**

# Updating directories

- `>>> age = {'Alice': 26 , 'Carol' : 22}`
- `>>> age.update({'Bob' : 29})`
- `>>> age`
  `{'Bob': 29, 'Carol': 22, 'Alice': 26}`
- `>>> age.update({'Carol' : 23})`
- `>>> age`
  `{'Bob': 29, 'Carol': 23, 'Alice': 26}`

# Returning a value

- `>>> age = {'Bob': 29, 'Carol': 23, 'Alice': 26}`
- `>>> age.get('Bob')`
  `29`
- `>>> age['Bob']`
  `29`

# Removing a specific item (I)

- `>>> a = {'Alice' : 26, 'Carol' : 'twenty-two'}`
- `>>> a`
  `{'Carol': 'twenty-two', 'Alice': 26}`
- `>>> a.pop('Carol')`
  `'twenty-two'`
- `>>> a`
  `{'Alice': 26}`

# Removing a specific item (II)

- `>>> a.pop('Alice')`
  `26`
- `>>> a`
  `{}`
- `>>>`

# Remove a random item

```
>>> age = {'Bob': 29, 'Carol': 23, 'Alice': 26}
>>> age.popitem()
('Bob', 29)
>>> age
{'Carol': 23, 'Alice': 26}
>>> age.popitem()
('Carol', 23)
>>> age
{'Alice': 26}
```

# Aliases & Clones

# Difference Between Aliases & Clones

- More than one variable can point to the same data. This is called an Alias.
- For example:
```
a = [ 5, 10, 50, 100]
b = a
```
Now, a and b both point to the same list.

- If you make a change to the data that a points to, you are also changing the data that b points to:
```
a[0] = 'Changed'
```

- a points to the list: ["Changed", 10, 50, 100]
- But because b points to the same data, b also points to the list ['Changed', 10, 50, 100]

# Difference Between Aliases & Clones

```
In [17]: a = [ 5, 10, 50, 100]

In [18]: b = a

In [19]: b
Out[19]: [5, 10, 50, 100]

In [20]: a[0] = 'Changed'

In [21]: b
Out[21]: ['Changed', 10, 50, 100]
```

# Cloning Data with the Slice operator

- If you want to make a clone (copy) of a sequence, you can use the slice operator ( [:] )
- For example:
  ```
  a = [ 5, 10, 50, 100]
  b = a[0:2]
  ```
  Now, b points to a cloned slice of a that is [ 5, 10]

- If you make a change to the data that a points to, you do NOT change the slice that b points to.
  ```
  a[0] = 'Changed'
  ```

- a points to the list: ["Changed", 10, 50, 100]
- b still points to the (different) list [5, 10]

# Cloning an entire list

- You can use the slice operator with a default START and END to clone an entire list (make a copy of it)
- For example:
  ```
  a = [ 5, 10, 50, 100]
  b = a[:]
  ```
  Now, a and b point to different copies of the list with the same data values.

- If you make a change to the data that a points to, nothing happens the the list that b points to.
  ```
  a[0] = 'Changed'
  ```

- a points to the list: ["Changed", 10, 50, 100]
- b points to the the copy of the old a: [ 5, 10, 50, 100]

```
In [27]: a = [ 5, 10, 50, 100]

In [28]: b = a[0:2]

In [29]: c = a[:]

In [30]: d = a

In [31]: a[0] = 'Changed'

In [32]: a
Out[32]: ['Changed', 10, 50, 100]
```

```
In [27]: a = [ 5, 10, 50, 100]

In [28]: b = a[0:2]

In [29]: c = a[:]

In [30]: d = a

In [31]: a[0] = 'Changed'

In [32]: a
Out[32]: ['Changed', 10, 50, 100]

In [33]: b
Out[33]: [5, 10]

In [34]: c
Out[34]: [5, 10, 50, 100]

In [35]: d
Out[35]: ['Changed', 10, 50, 100]
```

# Be very careful!

- The only difference in the above examples was the use of the slice operator in addition to the assignment operator:
- `b = a`
  vs
  `b = a[:]`
- This is a small difference in syntax, but it has a very big semantic meaning!
- Without the slice operator, a and b point to the same list. Changes to one affect the other.
- With it, they point to different copies of the list that can be changed independently.

# SET

# Sets

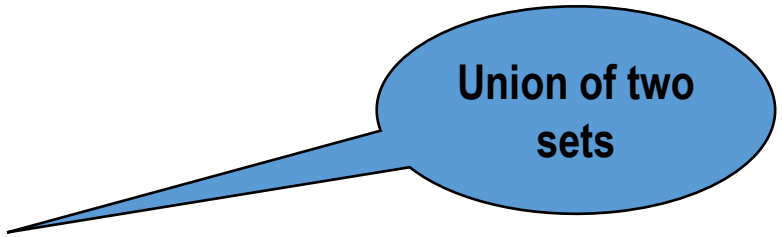
- Indentified by ***curly braces***
  - **{'Alice', 'Bob', 'Carol'}**
  - **{'Dean'}** is a ***singleton***
- Can only contain ***unique elements***
  - ***Duplicates are eliminated***
- ***Immutable*** like tuples and strings

# Sets do not contain duplicates

- **>>> cset = {11, 11, 22}**
- **>>> cset**
  **{11, 22}**

# Sets are immutable

- **>>> aset = {11, 22, 33}**
- **>>> bset = aset**
- **>>> aset = aset | {55}**
- **>>> aset**
  **{33, 11, 22, 55}**
- **>>> bset**
  **{33, 11, 22}**



# Sets have no order

- **>>> {1, 2, 3, 4, 5, 6, 7}**
  **{1, 2, 3, 4, 5, 6, 7}**
- **>>> {11, 22, 33}**
  **{33, 11, 22}**

# Sets do not support indexing

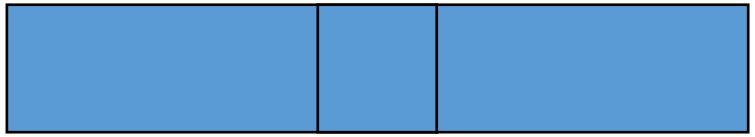
- **>>> myset = {'Apples', 'Bananas', 'Oranges'}**
- **>>> myset**
  **{'Bananas', 'Oranges', 'Apples'}**
- **>>> myset[0]**
  **Traceback (most recent call last):**
  **File "<pyshell#2>", line 1, in <module>**
  **myset[0]**
  **TypeError: 'set' object does not support indexing**

# Examples

- **>>> alist = [11, 22, 33, 22, 44]**
- **>>> aset = set(alist)**
- **>>> aset**
  **{33, 11, 44, 22}**
- **>>> aset = aset + {55}**
  **SyntaxError: invalid syntax**

# Boolean operations on sets (I)

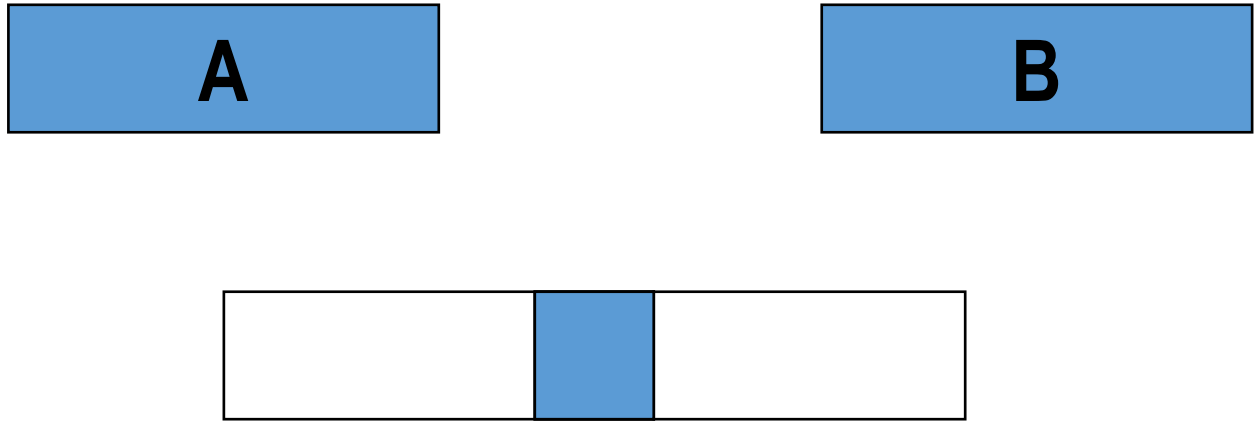
- **Union of two sets**

- Contains all elements that are in set **A or** in set **B**

# Boolean operations on sets (II)

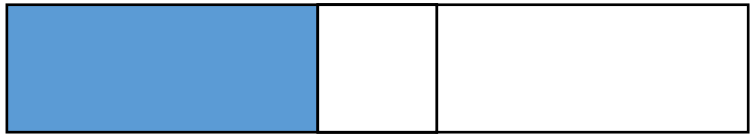
- **<u>Intersection of two sets</u>**




- Contains all elements that are in both sets **A and B**

# Boolean operations on sets (III)

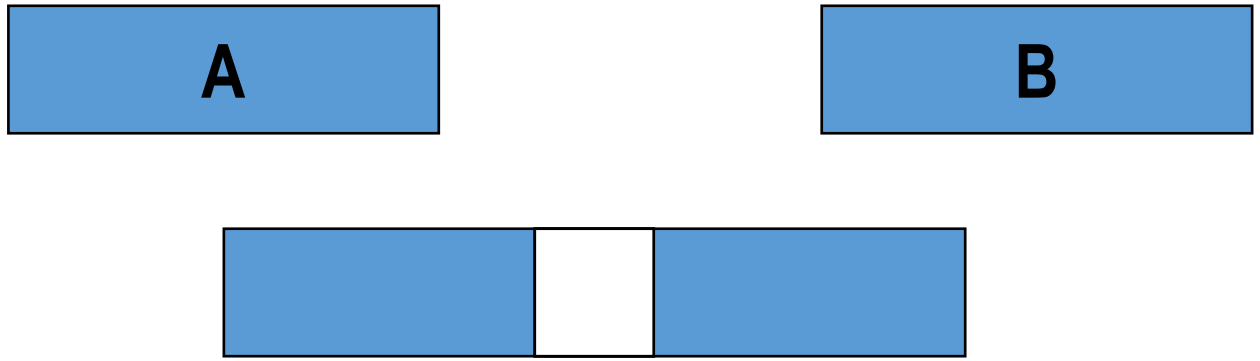
- **<u>Difference  of two sets</u>**

- Contains all elements that are **<u>in</u> A** but **<u>not in</u> B**

# Boolean operations on sets (IV)

- **Symmetric difference  of two sets**




- Contains all elements that are **either**
  - **in** set **A**  but **not in** set **B or**
  - **in** set **B**  but **not in** set **A**

# Boolean operations on sets (V)

- **>>> aset = {11, 22, 33}**
- **>>> bset = {12, 23, 33}**
- **<u>Union of two sets</u>**
  - **>>> aset | bset**
    **{33, 22, 23, 11, 12}**
- **<u>Intersection of two sets:</u>**
  - **>>> aset & bset**
    **{33}**

# Boolean operations on sets (VI)

- **>>> aset = {11, 22, 33}**
- **>>> bset = {12, 23, 33}**
- **<u>Difference:</u>**
  - **>>> aset – bset**
    **{11, 22}**
- **<u>Symmetric difference:</u>**
  - **>>> aset ^ bset**
    **{11, 12, 22, 23}**

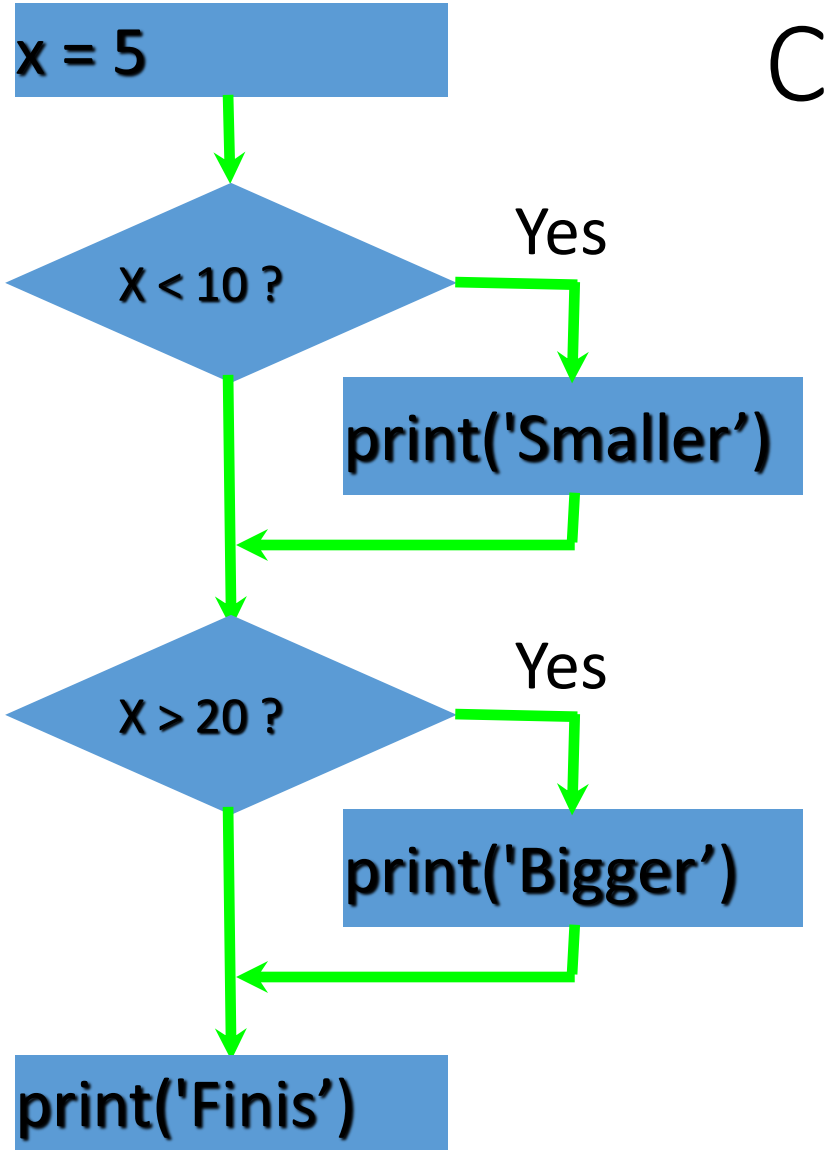
# Data Type Conversion:

| Function | Description |
|---|---|
| int(x [,base]) | Converts x to an integer. base specifies the base if x is a string. |
| float(x) | Converts x to a floating-point number. |
| complex(real [,imag]) | Creates a complex number. |
| str(x) | Converts object x to a string representation. |
| repr(x) | Converts object x to an expression string. |
| eval(str) | Evaluates a string and returns an object. |
| tuple(s) | Converts s to a tuple. |
| list(s) | Converts s to a list. |
| set(s) | Converts s to a set. |
| dict(d) | Creates a dictionary. d must be a sequence of (key,value) tuples. |
| frozenset(s) | Converts s to a frozen set. |
| chr(x) | Converts an integer to a character. |
| ord(x) | Converts a single character to its integer value. |
| hex(x) | Converts an integer to a hexadecimal string. |
| oct(x) | Converts an integer to an octal string. |

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# Conditional Steps




Program:

```
x = 5
if x < 10:
   print('Smaller')

if x > 20:
    print('Bigger')

print('Finis')
```

Output:

```
Smaller
Finis
```

# Comparison Operators

- Boolean expressions ask a question and produce a Yes or No result which we use to control program flow
- Boolean expressions using comparison operators evaluate to - True / False - Yes / No
- Comparison operators look at variables but do not change the variables

| Python | Meaning |
|---|---|
| < | Less than |
| <= | Less than or Equal |
| == | Equal to |
| >= | Greater than or Equal |
| > | Greater than |
| != | Not equal |

Remember: "=" is used for assignment.

http://en.wikipedia.org/wiki/George_Boole

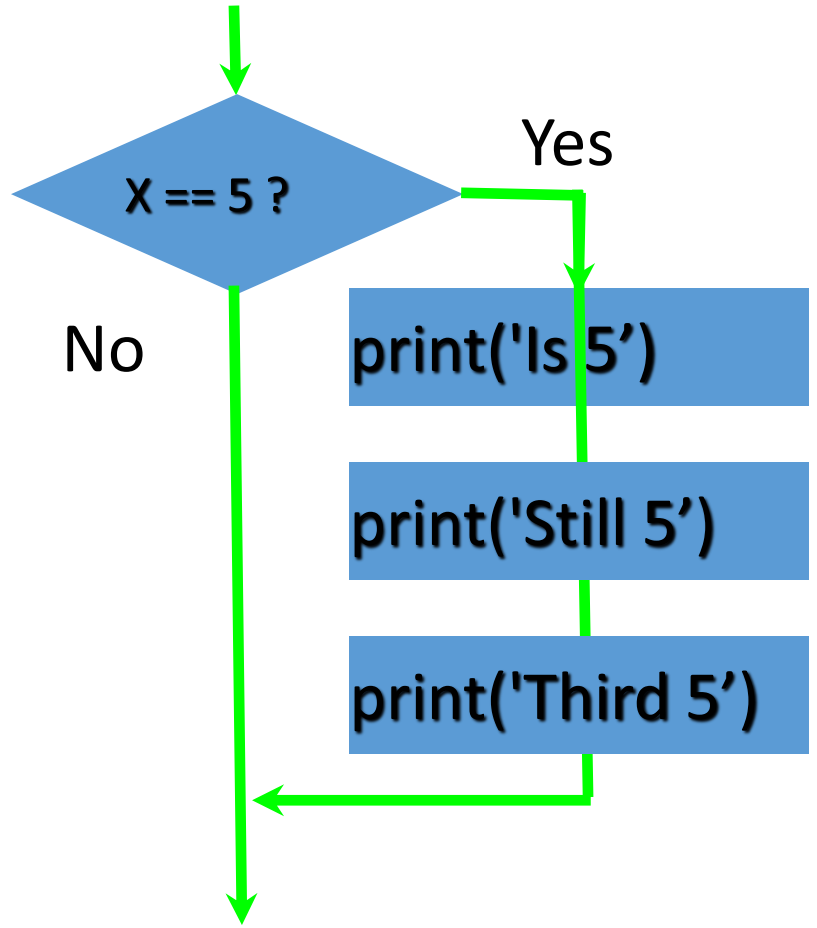
# Comparison Operators

```
x = 5
if x == 5 :
    print('Equals 5’)
if x > 4 :
   print('Greater than 4’)
if  x >= 5 :
    print('Greater than or Equal 5’)
if x < 6 : print('Less than 6’)
if x <= 5 :
    print('Less than or Equal 5’)
if x != 6 :
    print('Not equal 6’)
```

```
Equals 5
Greater than 4
Greater than or Equal 5
Less than 6
Less than or Equal 5
Not equal 6
```

```
x = 5
Print('Before 5’)
if  x == 5 :
    print('Is 5’)
    print('Is Still 5’)
    print('Third 5’)
print('Afterwards 5’)
print('Before 6’)
if x == 6 :
    print('Is 6’)
    print('Is Still 6’)
    print('Third 6’)
print('Afterwards 6’)
```

```
Before 5
Is 5
Is Still 5
Third 5
Afterwards 5
Before 6
Afterwards 6
```




# One-Way Decisions

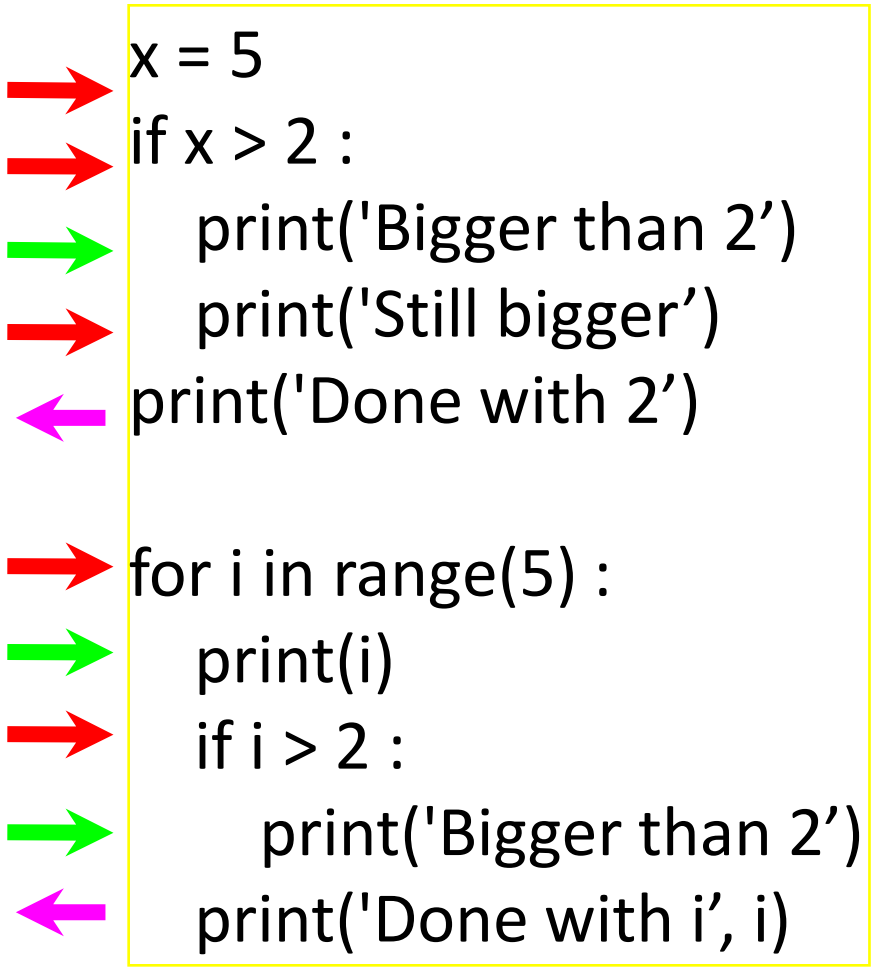
# Indentation

- Increase indent indent after an if statement or for statement (after : )
- Maintain indent to indicate the scope of the block (which lines are affected by the if/for)
- Reduce indent to *back to* the level of the if statement or for statement to indicate the end of the block
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored w.r.t. indentation

increase / maintain after if or for

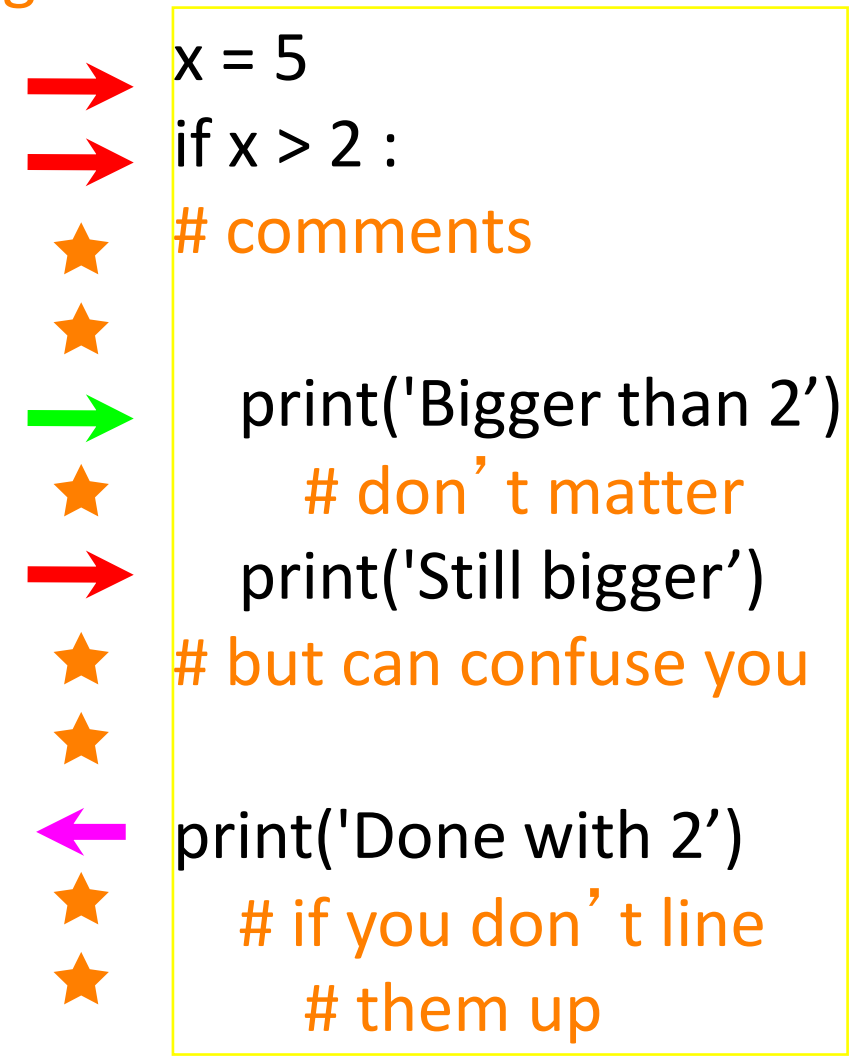
decrease to indicate end of block

blank lines and comment lines ignored



```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5) :
    print(i)
    if i > 2 :
        print('Bigger than 2')
    print('Done with i', i)
```



```
x = 5
if x > 2 :
# comments


    print('Bigger than 2')
        # don't matter
    print('Still bigger')
# but can confuse you


print('Done with 2')
    # if you don't line
        # them up
```

# Mental begin/end squares

```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')
```

```
for i in range(5) :
    print(i
    f i > 2 :
      print('Bigger than 2')
    print('Done with i', i)
```

```
x = 5
if x > 2 :
# comments

    print('Bigger than 2')
      # don't matter
    print('Still bigger')
# but can confuse you

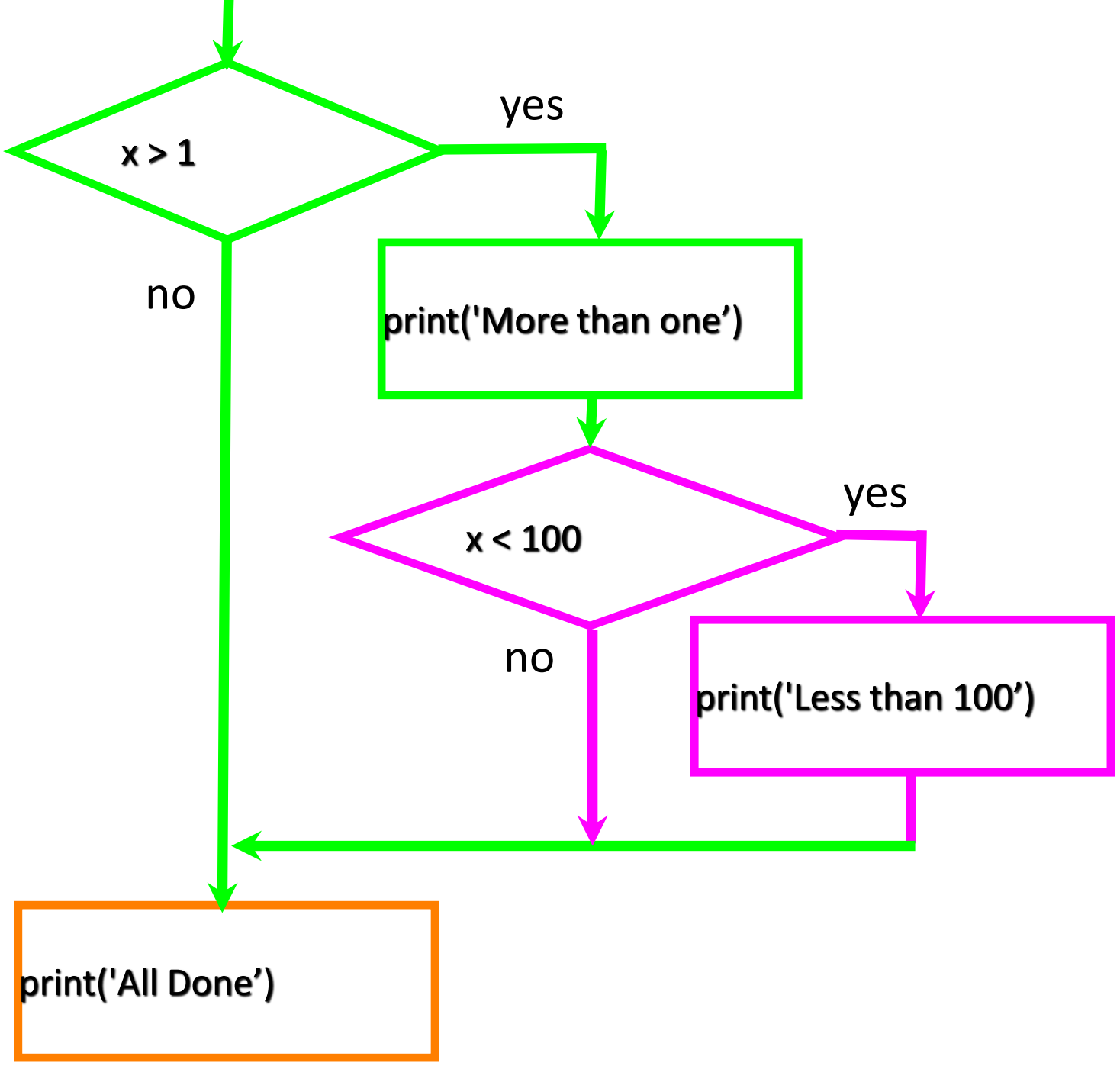
print('Done with 2')
  # if you don't line
    # them up
```

# Nested Decisions

```
x = 42

if x > 1 :
    print('More than one’)
    if x < 100 :
        print('Less than 100’)

print('All done’)
```

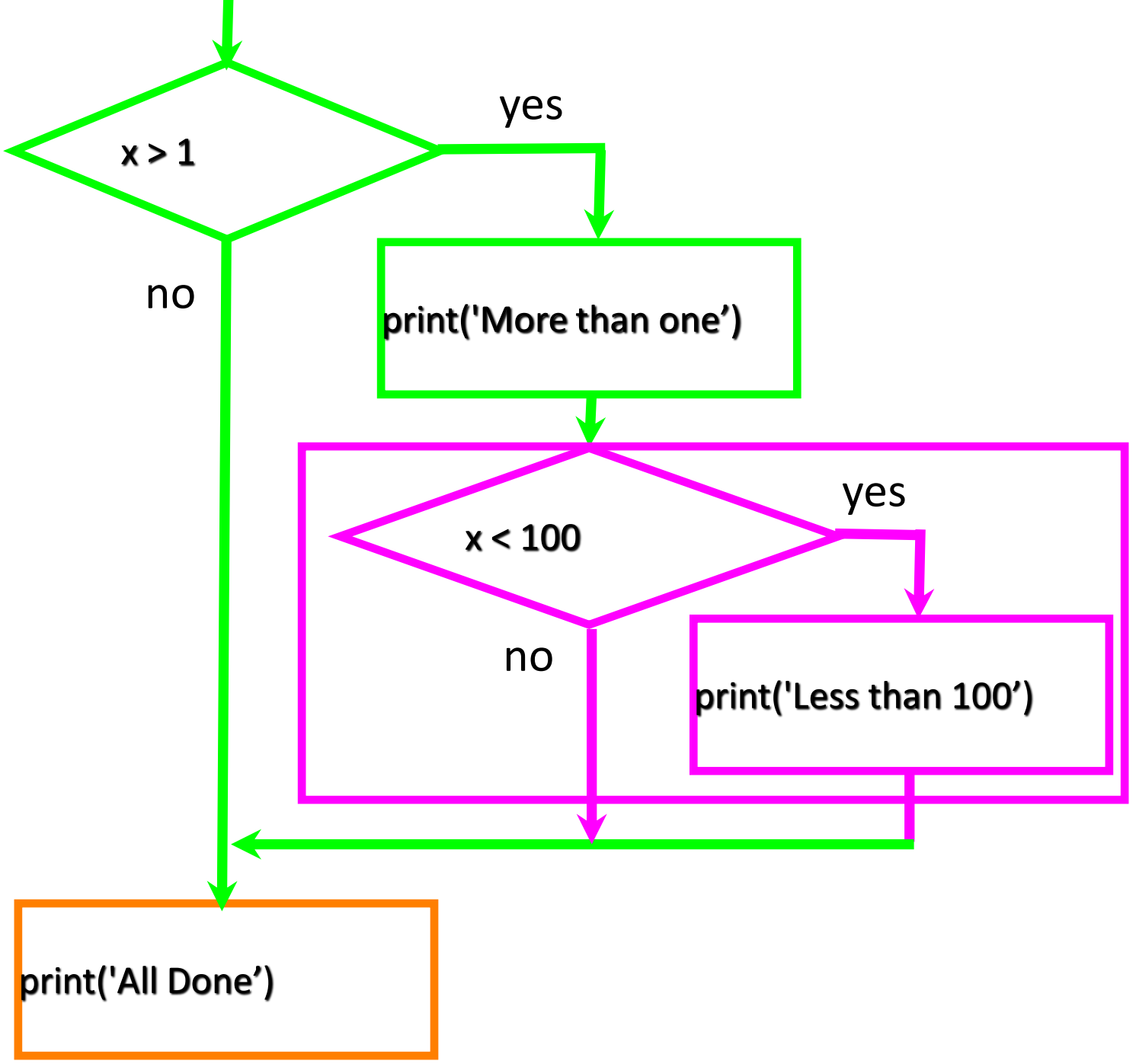


# Nested Decisions

```
x = 42

if x > 1 :
    print('More than one’)
    if x < 100 :
        print('Less than 100’)

print('All done’)
```

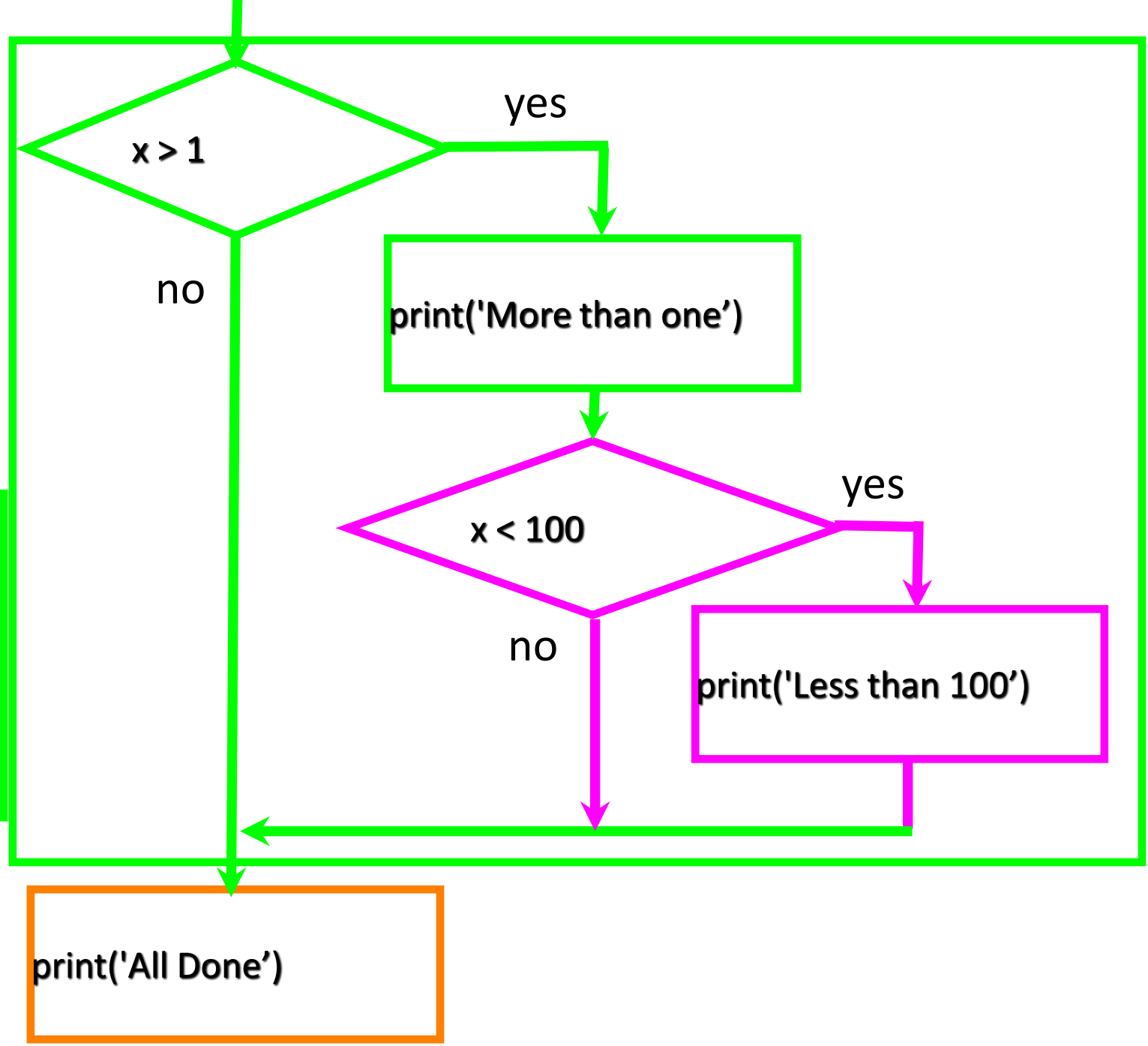


# Nested Decisions

x = 42

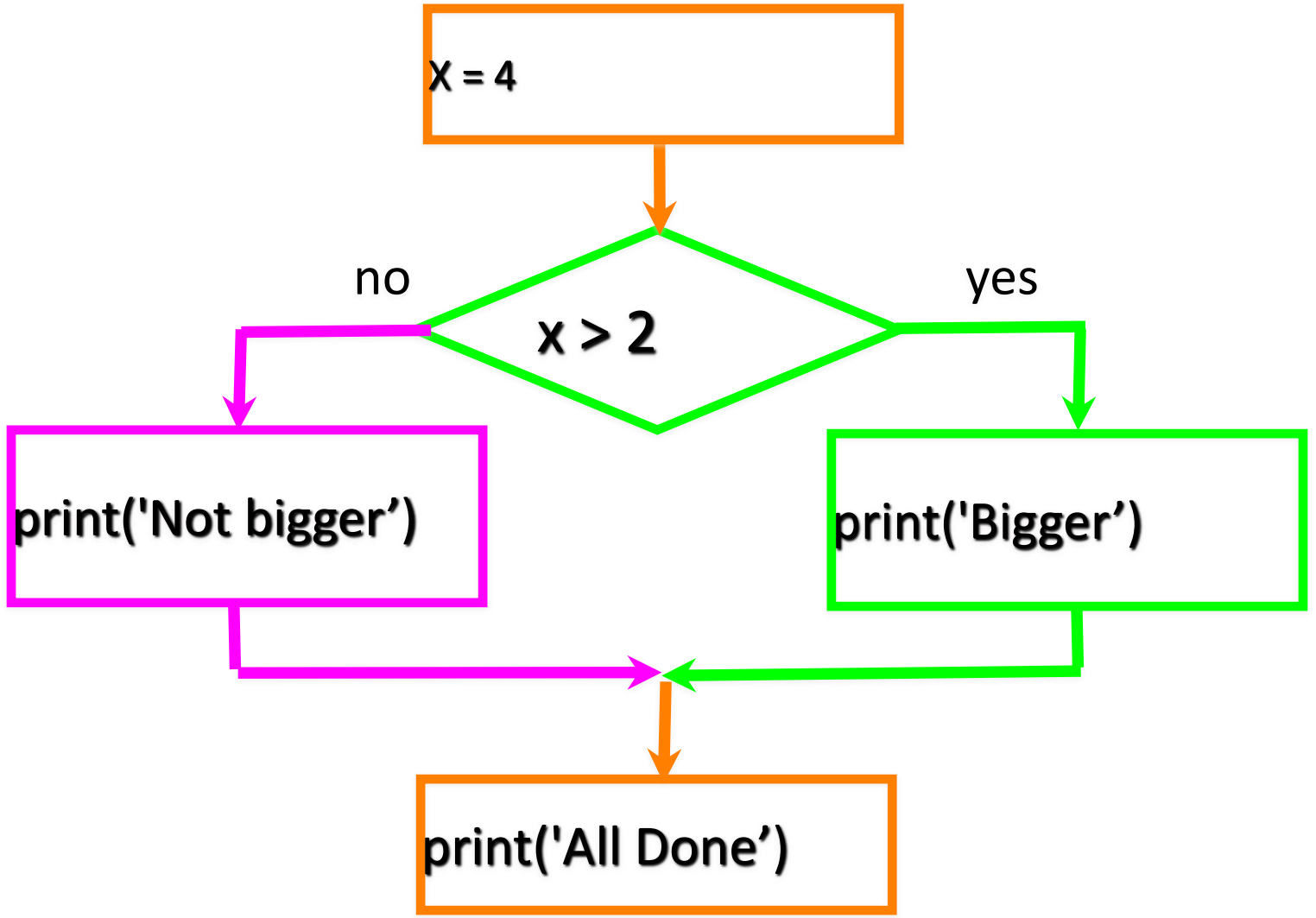
```
if x > 1 :
    print('More than one’)
    if x < 100 :
        print('Less than 100’)

print('All done’)
```



# Two Way Decisions

- Sometimes we want to do one thing if a logical expression is true and something else if the expression is false
- It is like a fork in the road - we must choose one or the other path but not both

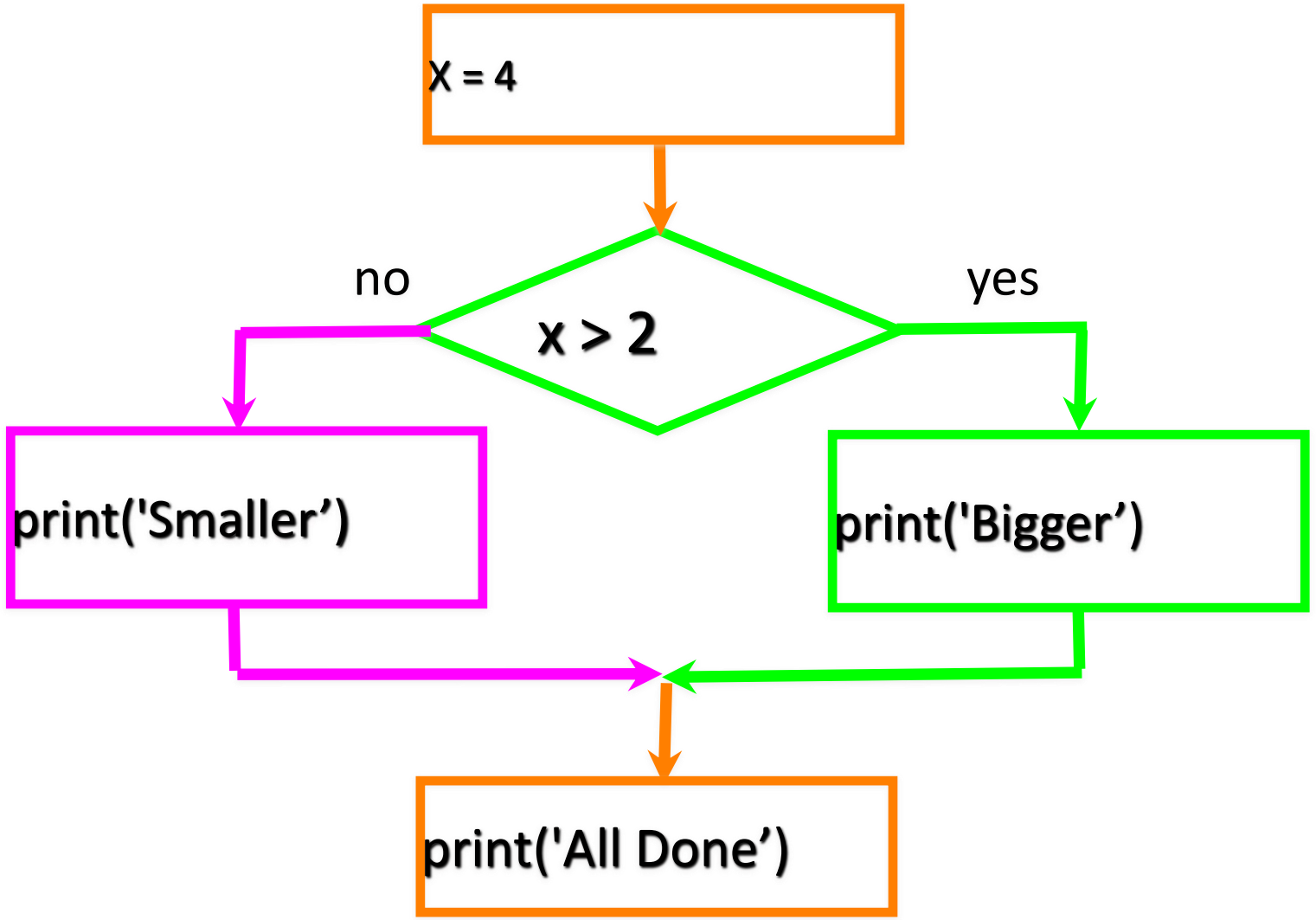


# Two-way using else :

```
x = 4

if x > 2 :
    print('Bigger’)
else :
    print('Smaller’)

print('All done’)
```

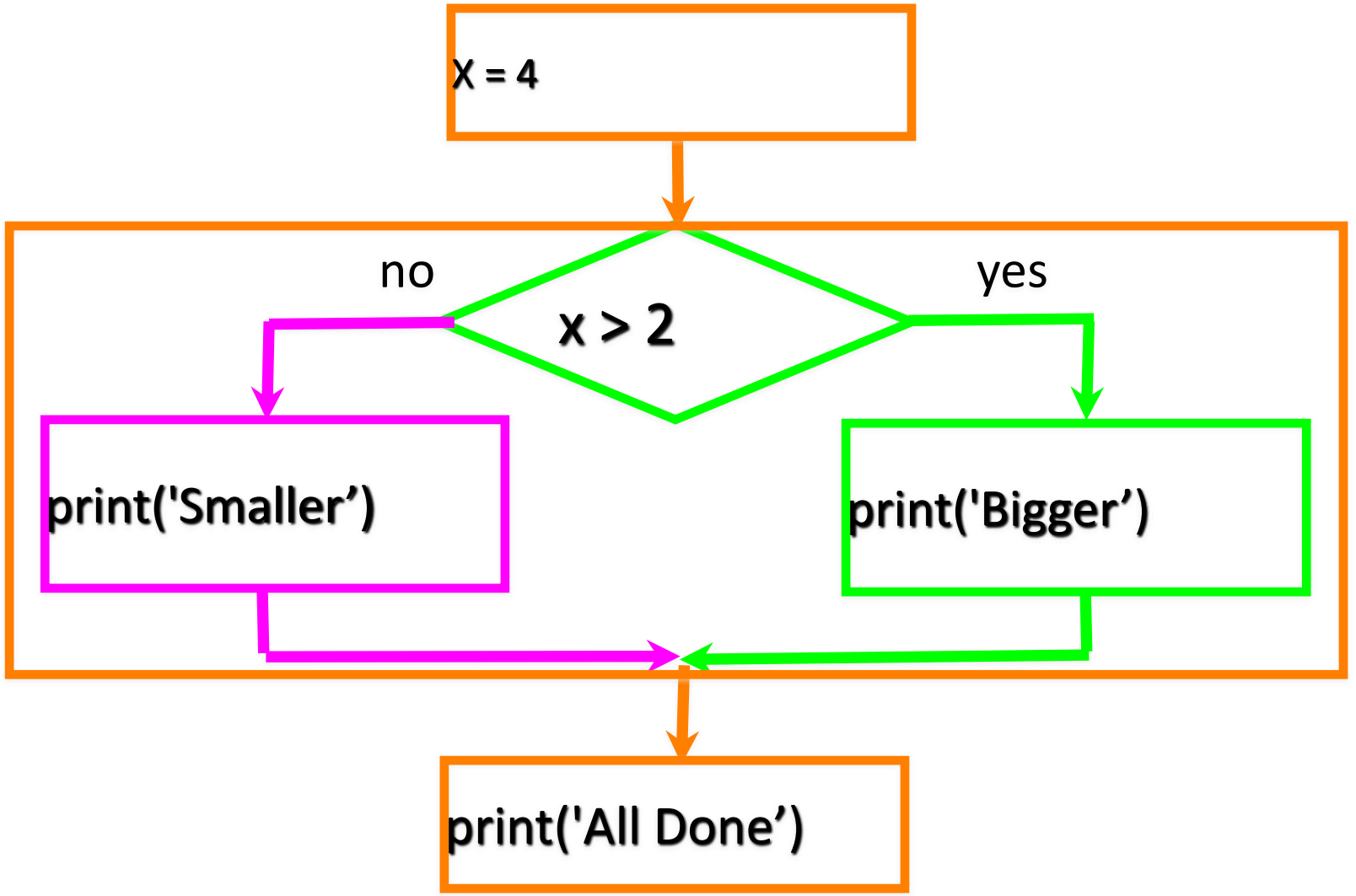


# Two-way using else :

```
x = 4

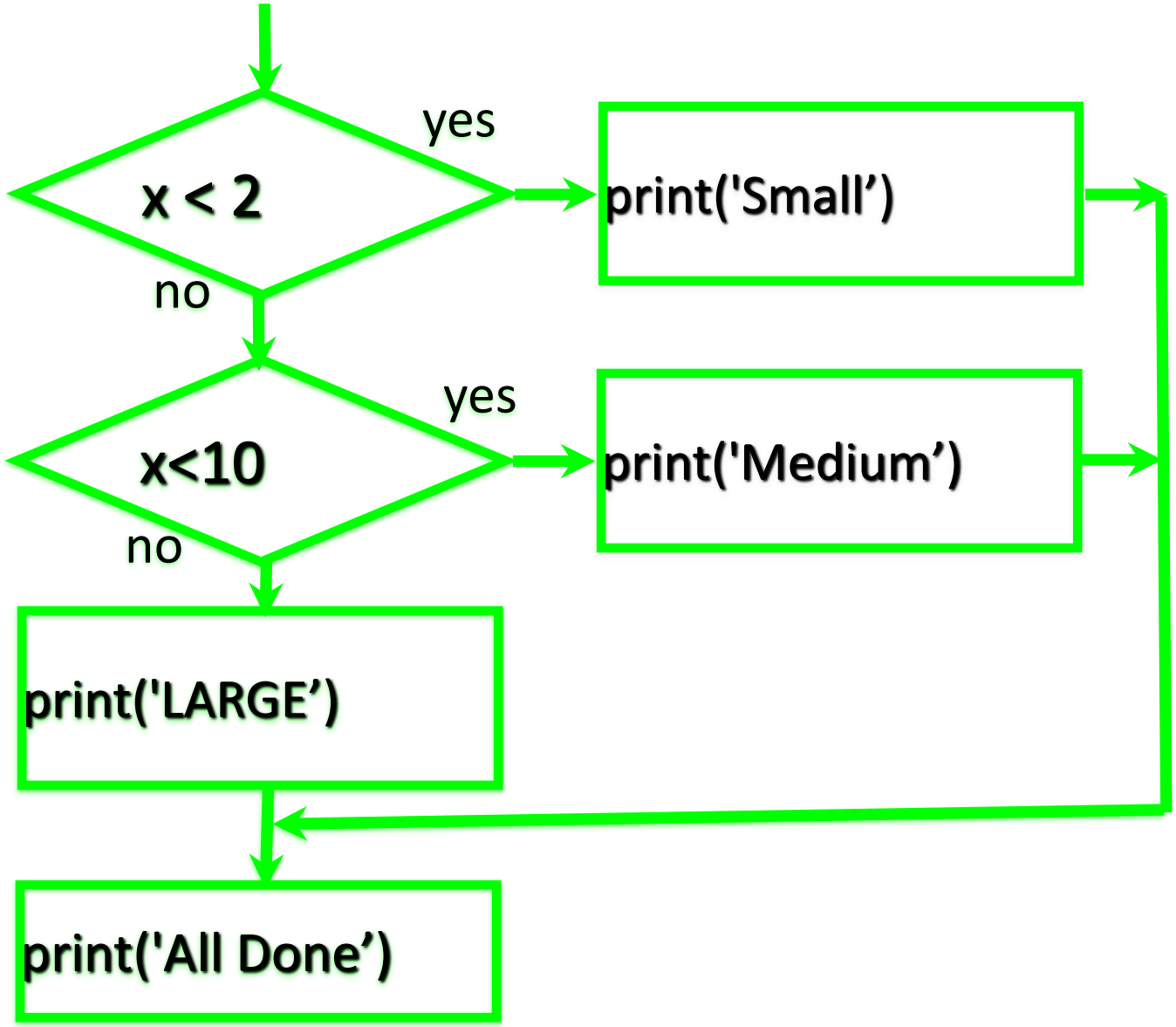
if x > 2 :
    print('Bigger')
else :
    print('Smaller')

print('All done')
```



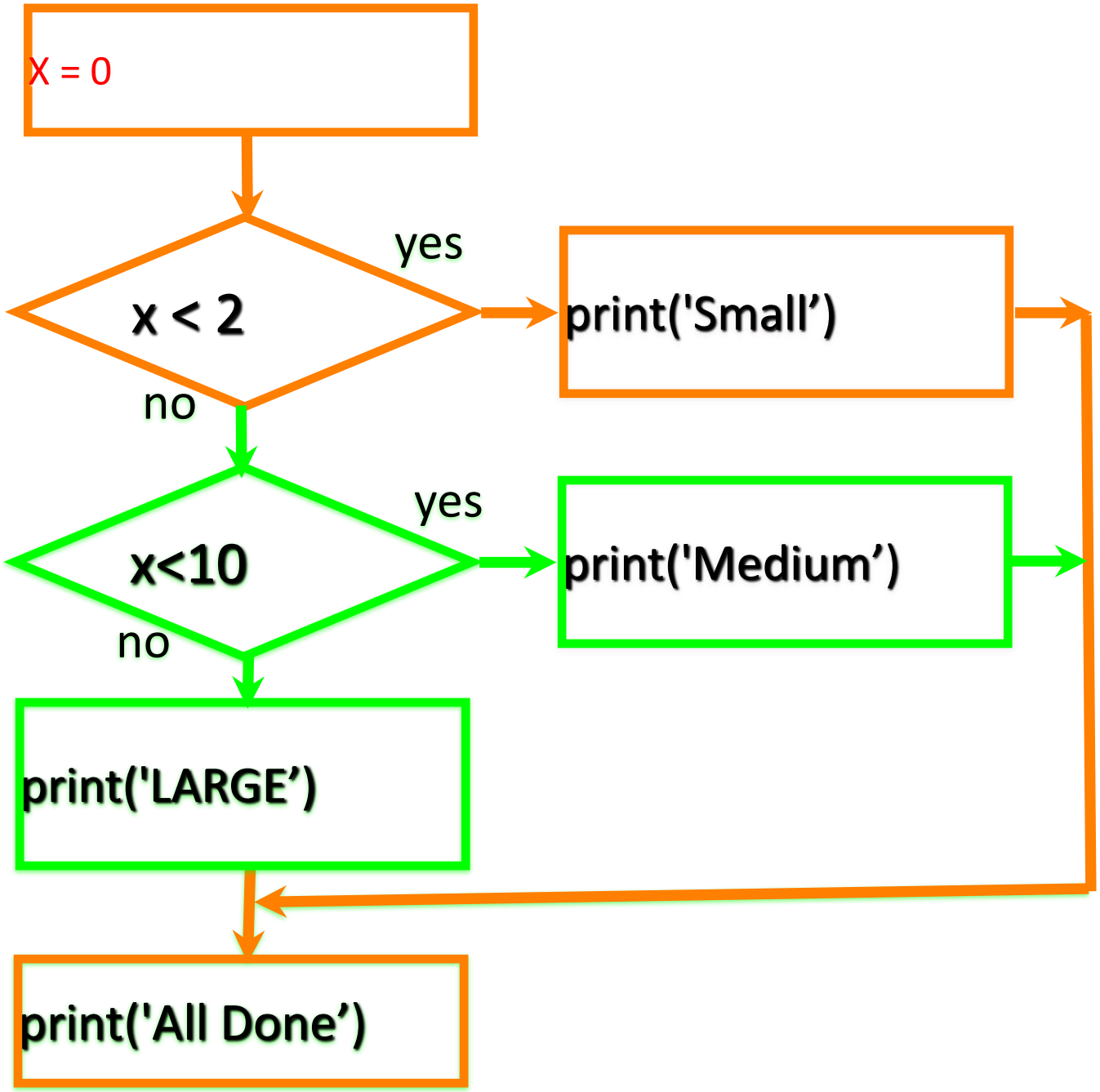
# Multi-way

```
if x < 2 :
    print('Small’)
elif x < 10 :
    print('Medium’)
else :
    print('LARGE’)
print('All done’)
```



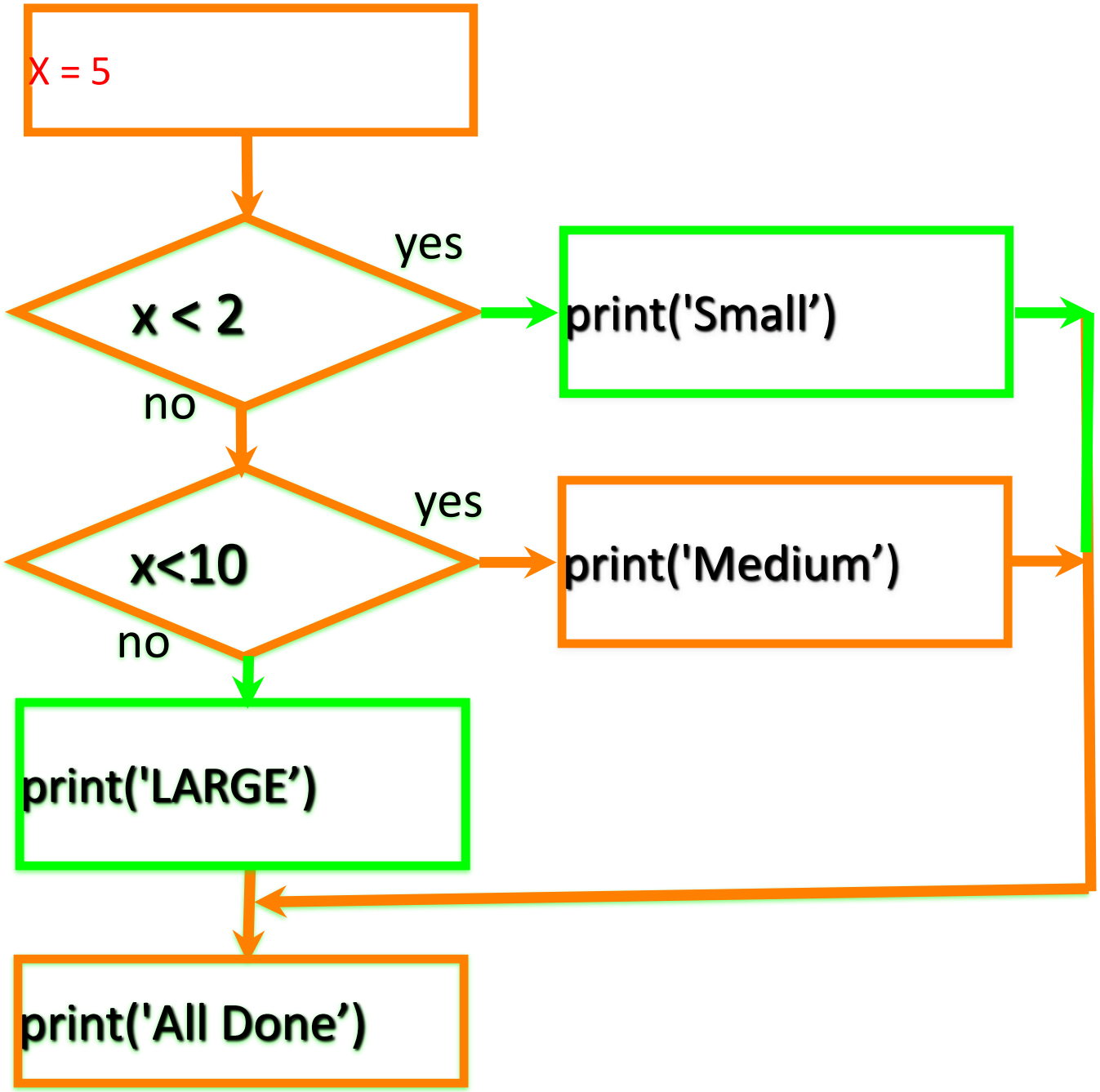
# Multi-way

```
x = 0
if x < 2 :
    print('Small’)
elif x < 10 :
    print('Medium’)
else :
    print('LARGE’)
print('All done’)
```



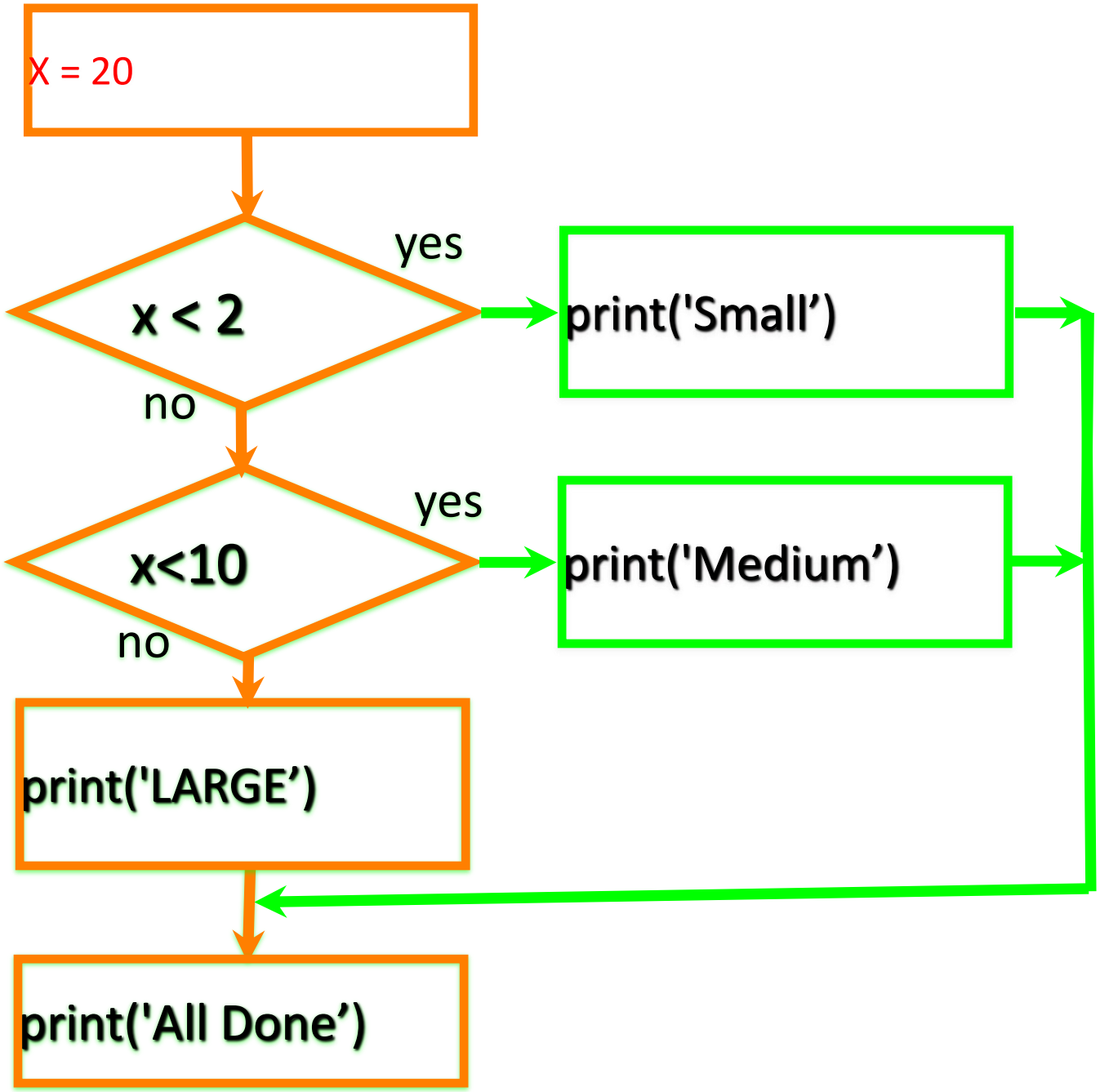
# Multi-way

```
x = 5
if x < 2 :
    print('Small’)
elif x < 10 :
    print('Medium’)
else :
    print('LARGE’)
Print('All done’)
```



# Multi-way

```
x = 20
if x < 2 :
    print('Small’)
elif x < 10 :
    print('Medium’)
else :
    print('LARGE’)
print('All done’)
```



# Multi-way

```
# No Else
x = 5
if x < 2 :
    print('Small’)
elif x < 10 :
    print('Medium’)

print('All done’)
```

```
if x < 2 :
    print('Small’)
elif x < 10 :
    print('Medium’)
elif x < 20 :
    print('Big’)
elif x< 40 :
    print('Large’)
elif x < 100:
    print('Huge’)
else :
    print('Ginormous’)
```

# Multi-way Puzzles

Which will never print?

```
if x < 2 :
    print('Below 2’)
elif x >= 2 :
    print('Two or more’)
else :
    print('Something else’)
```

```
if x < 2 :
    print('Below 2’)
elif x < 20 :
    print('Below 20’)
elif x < 10 :
    print('Below 10’)
else :
    print('Something else’)
```

# Summary

- Comparison operators == <= >= > < !=
- Logical operators: and or not
- Indentation
- One Way Decisions
- Two way Decisions  if :  and else :
- Nested Decisions
- Multiway decisions using elif

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# For Loops 1

- A for-loop steps through each of the items in a collection type, or any other type of object which is “iterable”

```
for <item> in <collection>:
    <statements>
```

- If <collection> is a list or a tuple, then the loop steps through each element of the sequence
- If <collection> is a string, then the loop steps through each character of the string

```
for someChar in “Hello World”:
    print(someChar)
```

# For Loops 2

```
for <item> in <collection>:
  <statements>
```

- <item> can be more than a single variable name
- When the <collection> elements are themselves sequences, then <item> can match the structure of the elements.
- This multiple assignment can make it easier to access the individual parts of each element

```
for (x,y) in [(a,1),(b,2),(c,3),(d,4)]:
  print(x)
```

# *For* loops & the *range()* function

- Since a variable often ranges over some sequence of numbers, the *range()* function returns a range of numbers from 0 up to but not including the number we pass to it.
- range(5) returns range(0,5) (a range object)
- list(range(5)) returns [0,1,2,3,4] (a list)
- So we could say:

```
for x in range(5):
    print(x)
```

- (There are more complex forms of *range()* that provide richer functionality…)

# For Loops and Dictionaries

```
>>> ages = { "Sam" : 4, "Mary" : 3, "Bill" : 2 }
>>> ages
{'Bill': 2, 'Mary': 3, 'Sam': 4}
>>> for name in ages.keys():
        print(name, ages[name])
Bill 2
Mary 3
Sam 4
```

# Looping, a better form of repetition.

- Repetition is OK for small numbers, but when you have to do something many, many times, it takes a very long time to type all those commands.
- We can use a loop to make the computer do the work for us.
- One type of loop is the "while" loop. The while loop repeats a block of code until a boolean expression is no longer true.
- Syntax:

```
while (boolean expression) :
    STATEMENT
    STATEMENT
    STATEMENT
```

# How to STOP looping!

- It is very easy to loop forever:

```
while ( True) :
   print("again, and again, and again"
```

- The hard part is to stop the loop!
- Two ways to do that is by using a loop counter, or a termination test.
  - A loop counter is a variable that keeps track of how many times you have gone through the loop, and the boolean expression is designed to stop the loop when a specific number of times have gone bye.
  - A termination test checks for a specific condition, and when it happens, ends the loop. (But does not guarantee that the loop will end.)

# Simple While Loop Example

- Let's write a while loop that prints the numbers from 1 to 10

```
i = 1          #initialize i 1
while i<=10: #execute the code in the loop
  print(i)          until i >10
  i = i +1    #increment i
print("all done!")
```

# Another While Loop Example

```
count =1
num =3
while count<5:
 print(num*count)
 count = count+1
```

*this prints 3, 6, 9, 12

# Variation on Last While Loop

```
num =3
while num<=12:
 print(num)
 num = num+3
```

*this loop also prints 3, 6, 9, 12

# Loop Counter

```
timesThroughLoop = 0

while (timesThroughLoop < 10):
    print("This is time", timesThroughLoop, "in the loop.")
    timesThroughLoop = timesThroughLoop + 1
```

- Notice that we:
  - Initialize the loop counter (to zero)
  - Test the loop counter in the boolean expression (is it smaller than 10, if yes, keep looping)
  - Increment the loop counter (add one to it) every time we go through the loop
- If we miss any of the three, the loop will NEVER stop!

# *while* Loops

```
>>> x = 3
>>> while x < 5:
      print(x, "still in the loop")
      x = x + 1
3 still in the loop
4 still in the loop
>>> x = 6
>>> while x < 5:
      print(x, "still in the loop")

>>>
```

# *break* and *continue*

- You can use the keyword *break* inside a loop to leave the *while* loop entirely.

- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# The try / except Structure

- You surround a dangerous section of code with try and except.
- If the code in the try works - the except is skipped
- If the code in the try fails - it jumps to the except section

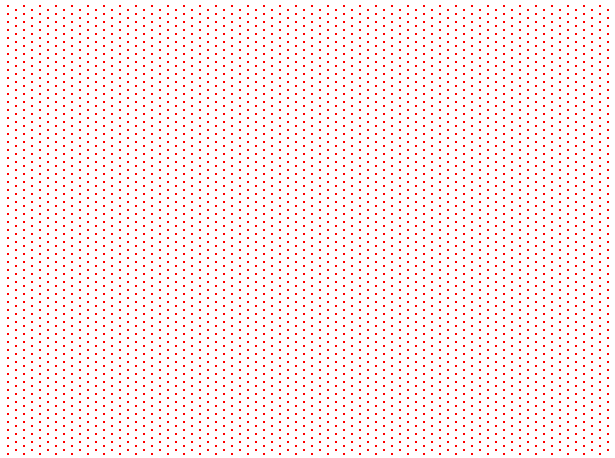
```
$ cat notry.py
astr = 'Hello Bob'
istr = int(astr)
print('First', istr)
astr = '123’
istr = int(astr)
print('Second', istr)
```

```
$ python notry.py Traceback (most
recent call last):  File "notry.py", line 2,
in <module>    istr =
int(astr)ValueError: invalid literal for
int() with base 10: 'Hello Bob'
```

All
Done

```
$ cat notry.py
astr = 'Hello Bob'
istr = int(astr)
```

The program stops here



```
$ python notry.py
Traceback (most recent call last):
File "notry.py", line 2, in <module>
istr = int(astr)ValueError: invalid literal
for int() with base 10: 'Hello Bob'
```

All Done

```
$ cat tryexcept.py
astr = 'Hello Bob'
try:
    istr = int(astr)
except:
    istr = -1

print('First', istr)

astr = '123'
try:
    istr = int(astr)
except:
    istr = -1

print('Second', istr)
```

When the first conversion fails - it just drops into the except: clause and the program continues.

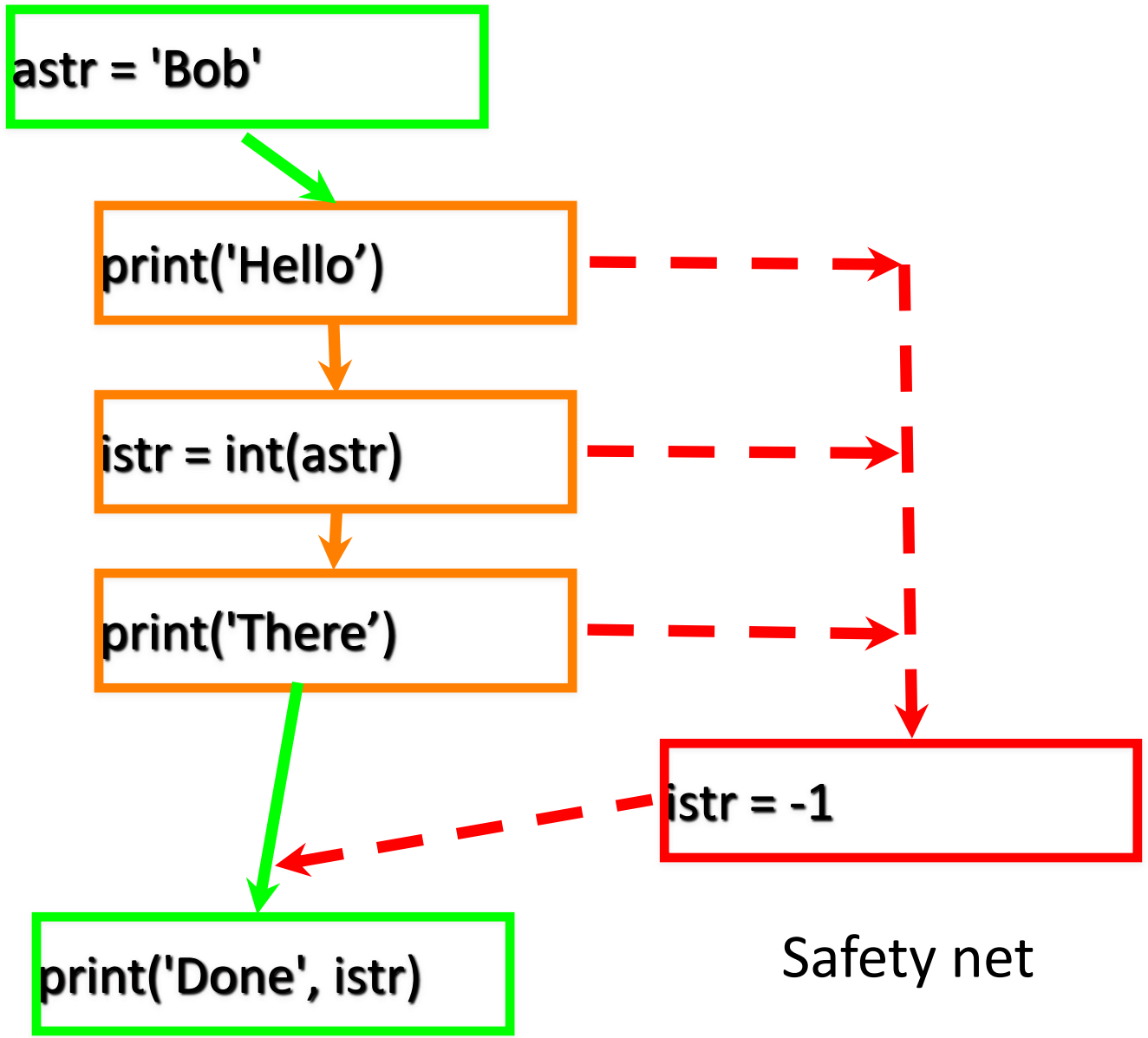
```
$ python tryexcept.py
First -1
Second 123
```

When the second conversion succeeds - it just skips the except: clause and the program continues.

# try / except

```
astr = 'Bob'
try:
    print('Hello’)
    istr = int(astr)
    print('There’)
except:
    istr = -1

print('Done', istr)
```



# Sample try / except

```
rawstr = input('Enter a positive number:')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print('Nice work‘)
else:
    print('Not a positive number’)
```

```
$ python trynum.py
Enter a number: 42
Nice work
$ python trynum.py
Enter a number: fourtytwo
Not a positive number
$
```

# Course Outline

- Introduction of Python (2 weeks)
  - Anaconda
  - IDE – spyder
  - Print
  - Open and write
  - String, list, dictionary, tuple
  - If … then
  - For loop
    - While loop
  - Try … except (exception handling)
  - Function

# FUNCTION

# Functions

- We’ve been some functions so far
  - int, float, str
- So what is a function?
  - Sequence of code which performs a specific task

# Functions, cont.

- Functions often take arguments
  - This is what you have to put inside the parentheses
  - For example, `input` “takes in” a string
    - `input ("enter your name")`
  - “Takes in” is another way of saying it takes an argument, in this case a string
- Functions often return a value
  - This is what you get back when you “call” the function
  - For example, `input` returns a string
  - int() returns an integer
  - float() returns a float

# Creating New Functions

- We don’t have to rely on only the functions in the Python library
- We can have functions that do almost anything
  - Can be as simple as adding 2 to a number
- Why use functions?
  - Makes our code easier to read
  - Lets us repeat code more efficiently

# Creating a New Function

- The syntax for creating a new function is:

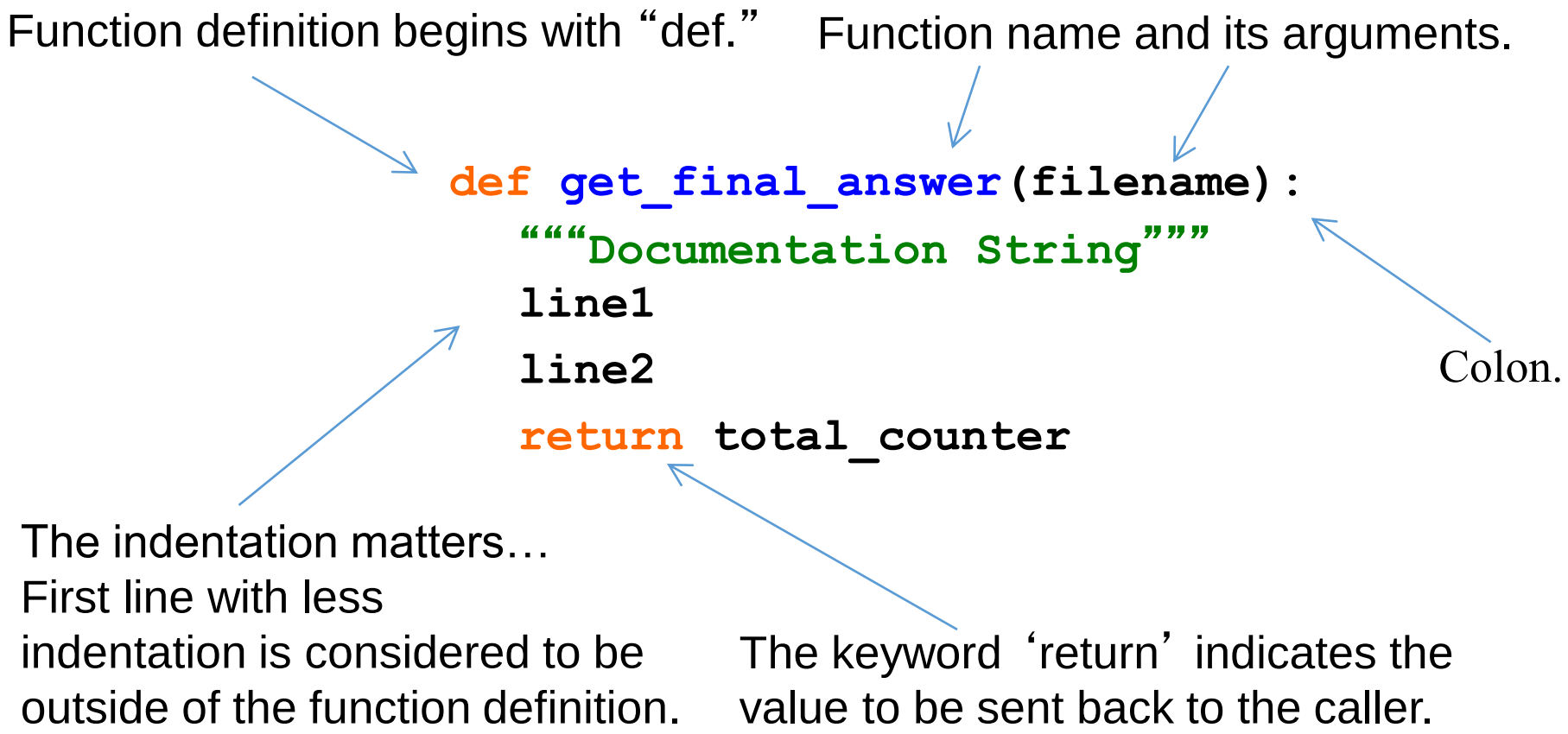
  def functionName(arguments):

- Let's define a function called printName that takes in a string:

  def printName(userName):

- Like with if, elif, else, and while blocks, indentation is important
  - The function is the code indented immediately following the function definition

# Defining Functions




No header file or declaration of types of function or arguments

# Simple Function

- Let's create a function that does one thing, prints "CMPT 165"

```
def printWord():
    print("CMPT165!!!")
```

- To call this function we simply use its name:

```
printWord()
```

- This function takes in no arguments and returns no values
  - All it does when we call it is print(the string "CMPT165")

# Changing our printWord()

- printWord is a pretty boring function
- Again it has no arguments and returns nothing
- So, let's change it so that we can give it any string to print

```
def printWord(wordToPrint):
    print(wordToPrint)
```

- Now when we call this function we have to provide something for it to print

```
printWord("CMPT165 Rules!")
```

# Passing Values into Functions

- The function `printWord` takes in the argument `wordToPrint`
  - `wordToprint` is a variable
- When we call the function `printWord`, whatever value we put inside the () is the value that the function uses

```
printWord("butterfly")
printWord("peanut")
printWord(2+4)   #prints 6
```

- The value in parenthesis in the function call is assigned to the argument variable

# Functions with Multiple Arguments

- We can have functions with many arguments
  - As many as we want
- Let's do a math example where we want to do some fancy math

```
def fancyMath(a,b)
    c = a*b+b/ (a+b) +b**a
    print(c)
```

# fancyMath(a,b), cont.

- fancyMath takes in two arguments
  - Need to be numbers (either ints or floats)
- When we call the function fancyMath we have to put in two values for a and b

```
fancyMath(2,3)
fancyMath(5.0,1)
fancyMath(num1, 3)
fancyMath(num1, num2)
```

- We can use integers, floats, or variables
  - And any combination thereof
- Remember that the value we pass into the function gets stored in the variables a and b

# In-Class

- Using fancyMath

# Return Values

- Functions can take in any number of arguments
  - From 0 ->whatever you want
- But functions can only return one value
- To return a value we use the *return* keyword
- We an return numbers, strings, and any values stored in variables

# Adding Return to fancyMath

- Right now fancyMath prints the value of c
  - Has the result of our fancy math calculation
- Instead, let's have it return the value so we can do something else with it in the main part of the program

```
def fancyMath(a,b)
    c = a*b+b/ (a+b) +b**a
    return c
```

# Returning Values, cont.

```
def fancyMath(a,b)
      c = a*b+b/ (a+b) +b**a
      return c
```

- Now fancyMath will have a value when we call it
- We can assign the value of that function to a variable

```
fancyNumber = fancyMath(3,2.4)
```

# Return Values, cont.

- The str() function takes in a number and returns a string
- But so far when using it, we've not assigned the value of that function to a variable before printing
- We can do the same thing with fancyMath

  print(fancyMath(2, 3.5)
- The value that fancyMath returns is then printed

# Functions, cont.

- If a function returns a value, we can use it as the argument for another function
- `fancyMath` returns a number
- But if we want to print(the result along with other strings, we can convert it to a string so we can concatenate it
- We are using the result of `fancyMath(3, 2.3)` as the argument for `str()`

```
print("Your result is: "+str(fancyMath(3,2.3)))
```

# Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8

# Keyword Arguments

- Can call a function with some/all of its arguments out of order as long as you specify their names

```
>>> def foo(x,y,z): return(2*x,4*y,8*z)
>>> foo(2,3,4)
(4, 12, 32)
>>> foo(z=4, y=2, x=3)
(6, 8, 32)
>>> foo(-2, z=-4, y=-3)
(-4, -12, -32)
```

- Can be combined with defaults, too

```
>>> def foo(x=1,y=2,z=3):
  return(2*x,4*y,8*z)
>>> foo()
(2, 8, 24)
>>> foo(z=100)
(2, 8, 800)
```

# Functional Programming

# Functions are first-class objects

Functions can be used as any other datatype, eg:

- Arguments to function
- Return values of functions
- Assigned to variables
- Parts of tuples, lists, etc

```
>>> def square(x): return x*x

>>> def applier(q, x): return q(x)

>>> applier(square, 7)
49
```

# Lambda Notation

Python's lambda creates anonymous functions

```
>>> lambda x: x + 1
<function <lambda> at 0x1004e6ed8>
>>> f = lambda x: x + 1
>>> f
<function <lambda> at 0x1004e6f50>
>>> f(100)
101
```

# Lambda Notation

- Python's lambda creates anonymous functions

```
>>> applier(lambda z: z * 2, 7)
14
```

- Note: only **one** expression in the lambda body; its value is always returned
- Python supports functional programming idioms: map, filter, closures, continuations, etc.

# Lambda Notation

Be careful with the syntax

```
>>> f = lambda x,y : 2 * x + y
>>> f
<function <lambda> at 0x87d30>
>>> f(3, 4)
10
>>> v = lambda x: x*x(100)
>>> v
<function <lambda> at 0x87df0>
>>> v = (lambda x: x*x)(100)
>>> v
10000
```

# map

```
>>> def add1(x): return x+1
>>> map(add1, [1,2,3,4])
[2, 3, 4, 5]
>>> map(lambda x: x+1, [1,2,3,4])
[2, 3, 4, 5]
>>> map(+, [1,2,3,4], [100,200,300,400])
map(+,[1,2,3,4],[100,200,300,400])
          ^
SyntaxError: invalid syntax
```

# map

- + is an operator, not a function
- We can define a corresponding add function

```
>>> def add(x, y): return x+y
>>> map(add,[1,2,3,4],[100,200,300,400])
[101, 202, 303, 404]
```

- Or import the operator module

```
>>> from operator import *
>>> map(add, [1,2,3,4], [100,200,300,400])
[101, 202, 303, 404]
>>> map(sub, [1,2,3,4], [100,200,300,400])
[-99, -198, -297, -396]
```

# filter, reduce

- Python has buiting for reduce and filter

```
>>> reduce(add, [1,2,3,4])
10
>>> filter(odd, [1,2,3,4])
[1, 3]
```

- The map, filter and reduce functions are also at risk ☹

# 4. Map, Filter and Reduce

These are three functions which facilitate a functional approach to programming. We will discuss them one by one and understand their use cases.

## 4.1. Map

`Map` applies a function to all the items in an input_list. Here is the blueprint:

### Blueprint

```
map(function_to_apply, list_of_inputs)
```

## 4.2. Filter

As the name suggests, `filter` creates a list of elements for which a function returns true. Here is a short and concise example:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)

# Output: [-5, -4, -3, -2, -1]
```

The filter resembles a for loop but it is a builtin function and faster.

**Note:** If map & filter do not appear beautiful to you then you can read about `list/dict/tuple` comprehensions.

## 4.3. Reduce

`Reduce` is a really useful function for performing some computation on a list and returning the result. It applies a rolling computation to sequential pairs of values in a list. For example, if you wanted to compute the product of a list of integers.

So the normal way you might go about doing this task in python is using a basic for loop:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

Now let's try it with reduce:

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Output: 24
```

# SUMMARY

# Python Syntax

- Python uses indentation and/or whitespace to delimit statement blocks rather than keywords or braces
- **if __name__** == "__main__":
  **print(**“Salve Mundo"**)**
  *# ‘\n’ is auto-included unless you state end=“ ” in print function*

*CONDITIONALS*

- **if** (i == 1): do_something1()
  **elif** (i == 2): do_something2()
  **elif** (i == 3): do_something3()
  **else**: do_something4()

# Conditionals Cont.

- **if** (value **is not None**) **and** (value == 1):
  **print(**"value equals 1",end=" "**)**
  **print(**"more can come in this block"**)**
- **if** (list1 <= list2) **and** (**not** age < 80):
  **print(**"1 = 1, 2 = 2, but 3 <= 7 so its True"**)**
- **if** (job == "millionaire") **or** (state != "dead"**)**:
  **print(**"a suitable husband found"**)**
  **else**:
  **print(**"not suitable“**)**
- **if** ok: **print(**"ok“**)**

# Loops/Iterations

- sentence = ['Marry','had','a','little','lamb']
  for word in sentence:
  print(word, len(word))
- for i in range(10):
  print(i)
  for i in range(1000):# does not allocate all initially
  print(i)
- while True:
  pass
- for i in range(10):
  if i == 3: continue
  if i == 5: break
  print(i, end=" "})

# Functions

- def print_hello():# returns nothing
  print(“hello”)
- def has_args(arg1,arg2=['e', 0]):
  num = arg1 + 4
  mylist = arg2 + ['a',7]
  return [num, mylist]

  has_args(5.16,[1,'b'])# returns [9.16,[[1, ‘b’],[ ‘a’,7]]
- def duplicate_n_maker(n): #lambda on the fly func.
  return lambda arg1:arg1*n

  dup3 = duplicate_n_maker(3)
  dup_str = dup3('go')  # dup_str == 'gogogo'

# Exception handling

- ```
  try:
          f = open("file.txt")
  except IOError:
          print("Could not open“)
  else:
          f.close()
  ```
- ```
  a = [1,2,3]
  try:
          a[7] = 0
  except (IndexError,TypeError):
          print("IndexError caught”)
  except Exception, e:
          print("Exception: ", e )
  except:       # catch everything
          print("Unexpected:“)
          print(sys.exc_info()[0])
          raise # re-throw caught exception
  try:
          a[7] = 0
  finally:
          print("Will run regardless“)
  ```
- Easily make your own exceptions:

  ```
  class myException(except)
          def __init__(self,msg):
                  self.msg = msg
          def __str__(self):
                  return repr(self.msg)
  ```