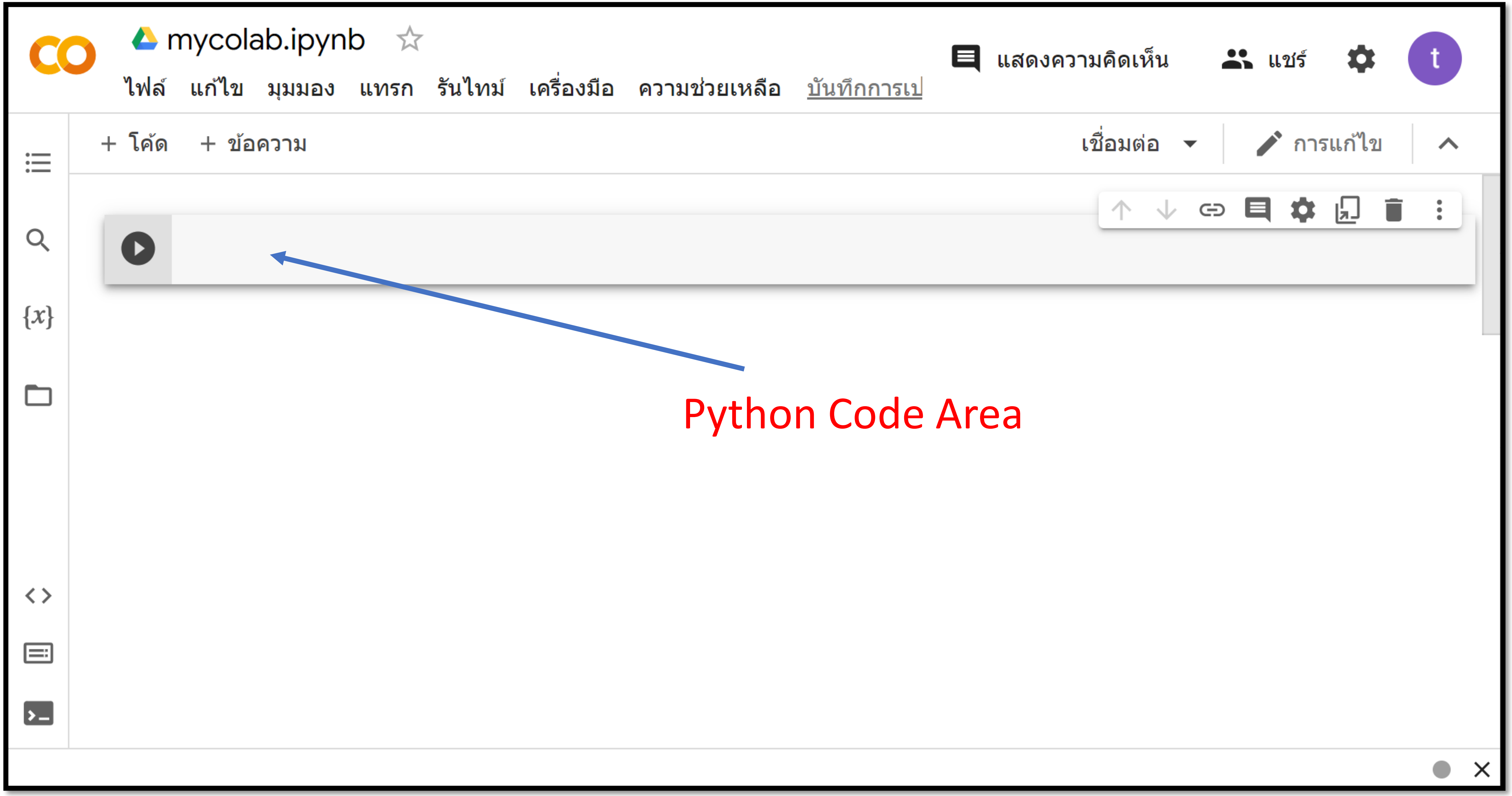
Data Structure and Algorithm

Google Colab

Python

Google colab

- Colaboratory, or “Colab” for short, is a product from Google Research.
- Colab allows anybody to write and execute arbitrary python code through the browser
- Google colab = free cloud service and support free GPU
- Connect with google drive
- machine learning, data analysis and education.
- Keras , Tensorflow , PyTorch , OpenCV
- Support for Tensor Processing Unit (TPU)
- FREE !!!!
- Detail : https://research.google.com/colaboratory/faq.html



Python Code Area

Hardware accelerator

CPU

- Small models
- Small datasets
- Useful for design space exploration

GPU

- Medium-to-large models, datasets
- Image, video processing
- Application on CUDA or OpenCL

TPU

- Matrix computations
- Dense vector processing
- No custom TensorFlow operations

Activity : ลองใช้ Colab ใน google drive

Python Introduction

- An interpreted, compiled, and interactive, object-oriented, dynamic, imperative, and open source programming language.
- Created in early 90's by Guido von Rossum at Stichting Mathematisch Centrum in the Netherlands.
- The name comes from the Monty Python (a group of British comedian) ,not from the snake.
- There is a big community of Python programmers, with conferences and magazines: http://pycon.org/
- Web site: www.python.org.

Activity : ลองใช้งานภาษา python

Python basic : https://colab.research.google.com/github/data-psl/lectures2020/blob/master/notebooks/01_python_basics.ipynb

Assignment Statement

- A simple assignment statement

  Variable = Expression;

- Computes the value (object) of the expression on the right hand side expression (RHS)
- Associates the name (variable) on the left hand side (LHS) with the RHS value
- = is known as the assignment operator.

Multiple Assignments

- Python allows multiple assignments

```
x, y = 10, 20
```

Binds x to 10 and y to 20

- Evaluation of multiple assignment statement:
  - All the expressions on the RHS of the = are first evaluated **before any binding happens.**
  - Values of the expressions are bound to the corresponding variable on the LHS.

```
x, y = 10, 20
x, y = y+1, x+1
```

x is bound to 21 and y to 11 at the end of the program

Input / output

Source code :

```
x = input('input 1 integer :')

print("input = ",x)
```

Output :

```
input 1 integer :1000
input = 1000
```

Operators

Arithmetic	+	-	*	//	/	%	**	
Comparison	==	!=	>	<	>=	<=		
Assignment	=	+=	-=	*=	//=	/=	%=	**=
Logical	and	or	not					
Bitwise	&	|	^	~	>>	<<		
Membership	in	not in						
Identity	is	is not						

Elements of Python

- A Python program is a sequence of **definitions** and **commands (statements)**
- Commands manipulate **objects**
- Each object is associated with a **Type**
- **Type:**
  - A set of values
  - A set of operations on these values
- **Expressions**: An operation (combination of objects and **operators**)

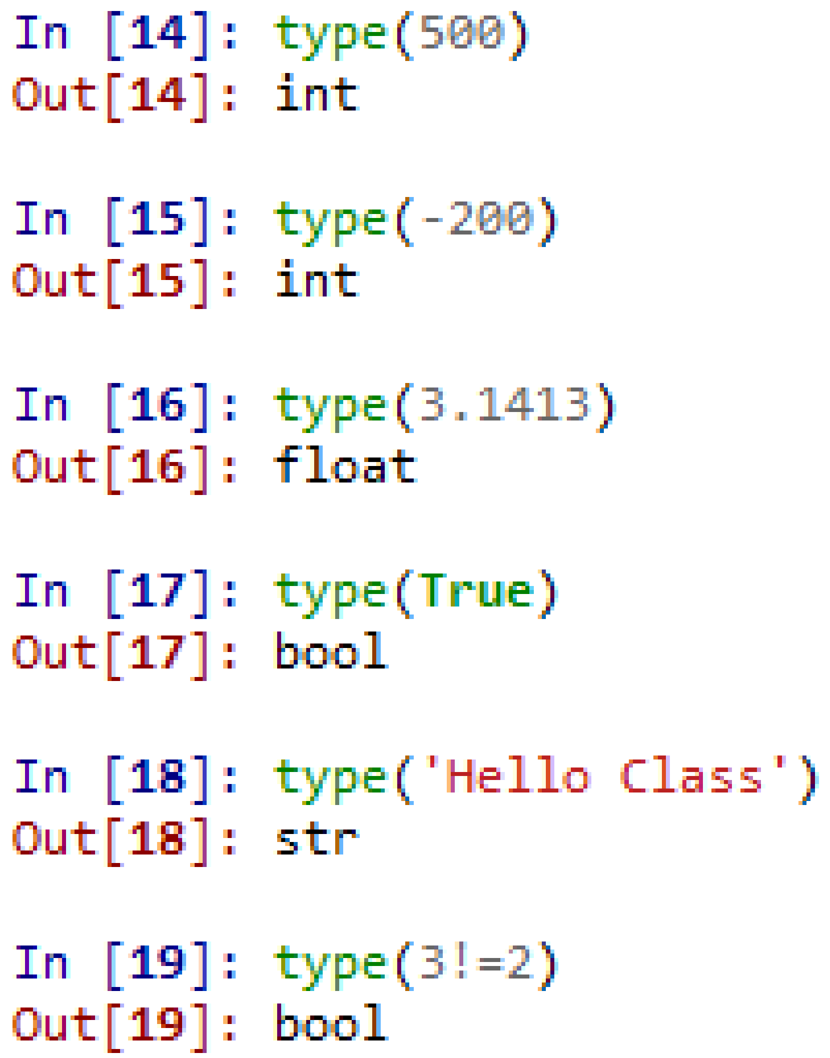
Types in Python

- int
  - Bounded integers, e.g. 732 or -5
- float
  - Real numbers, e.g. 3.14 or 2.0
- long
  - Long integers with unlimited precision
- str
  - Strings, e.g. ‘hello’ or ‘C’

Types in Python

- Scalar
  - Indivisible objects that do not have internal structure
  - **int** (signed integers), **float** (floating point), **bool** (Boolean), ***NoneType***
    - NoneType is a special type with a single value
    - The value is called **None**
- Non-Scalar
  - Objects having internal structure
  - **str** (strings)

Example of Types



```
In [14]: type(500)
Out[14]: int

In [15]: type(-200)
Out[15]: int

In [16]: type(3.1413)
Out[16]: float

In [17]: type(True)
Out[17]: bool

In [18]: type('Hello Class')
Out[18]: str

In [19]: type(3!=2)
Out[19]: bool
```

Built-in Data Structures

List :

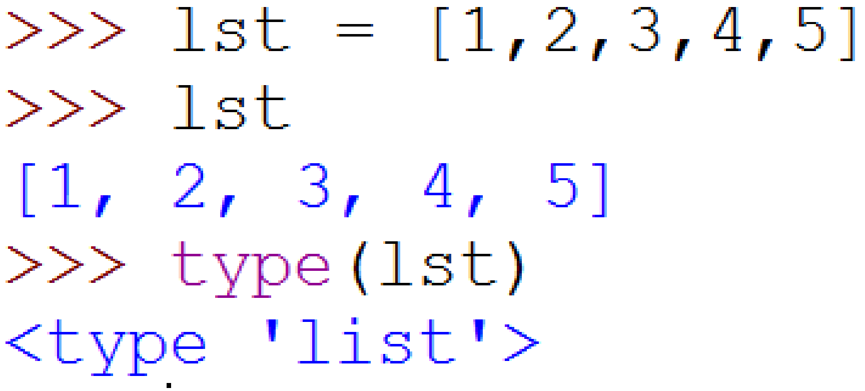
```
a = [1,2,3]

a.append(4)

print (a)
```

Output :

```
[1, 2, 3, 4]
```

Lists

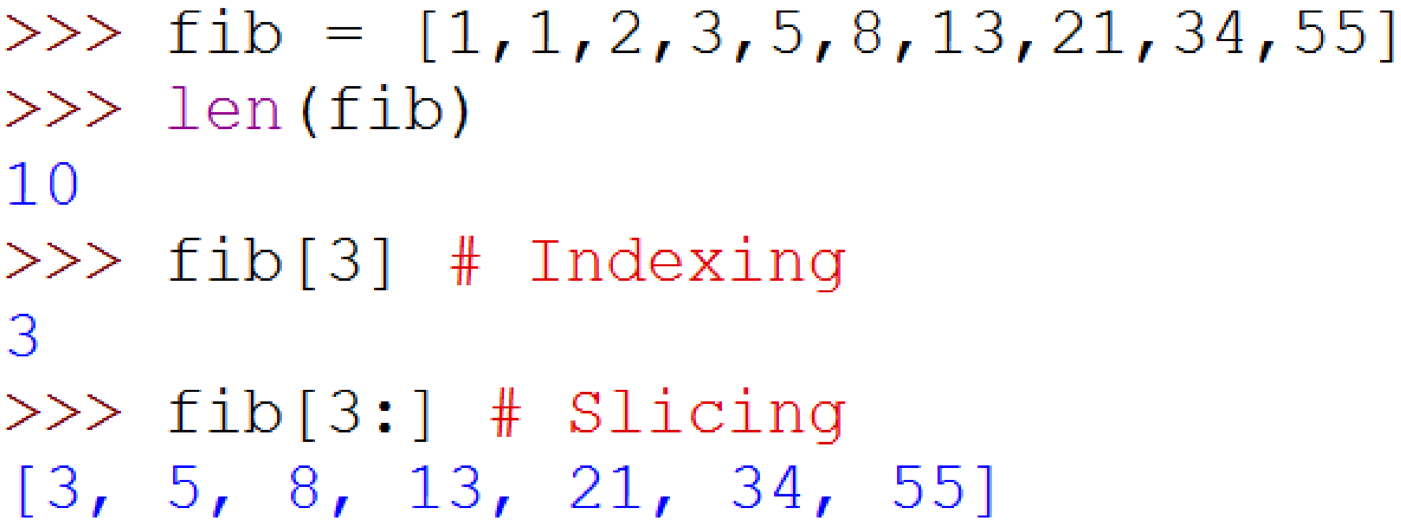
- Ordered sequence of values
- Written as a sequence of comma-separated values between square brackets
- Values can be of different types
    - usually the items all have the same type



```
>>> lst = [1,2,3,4,5]
>>> lst
[1, 2, 3, 4, 5]
>>> type(lst)
<type 'list'>
```

Lists

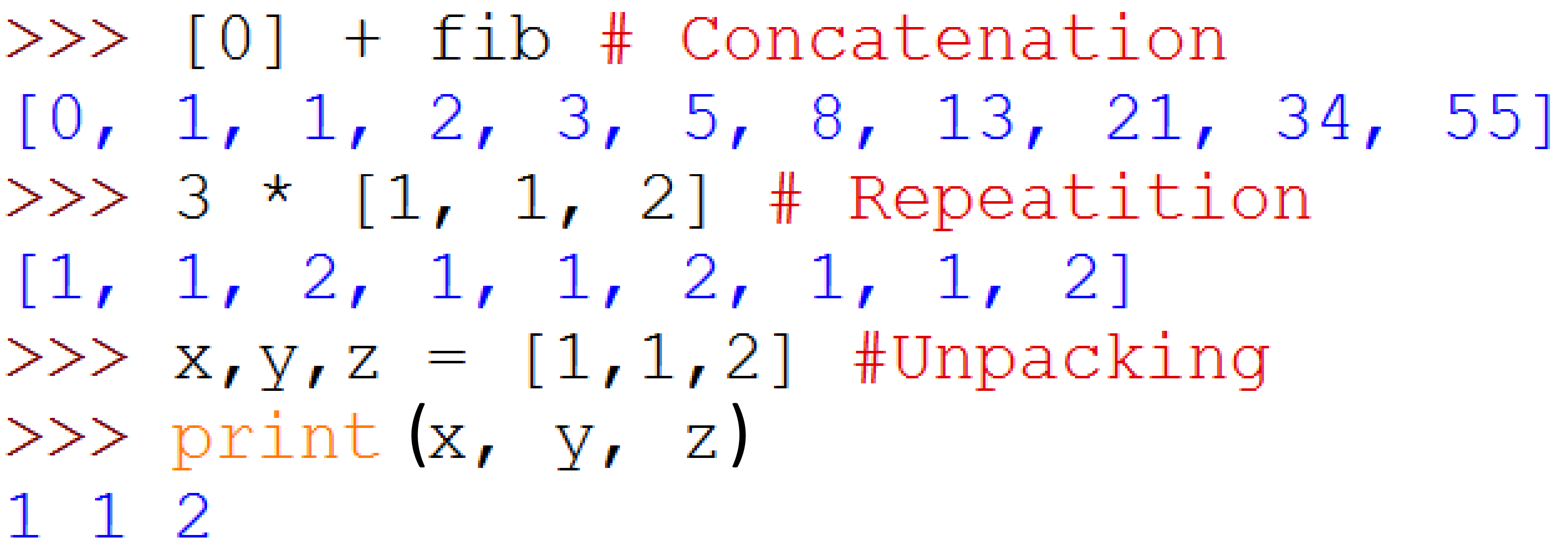
- List is also a sequence type
  - Sequence operations are applicable



```
>>> fib = [1,1,2,3,5,8,13,21,34,55]
>>> len(fib)
10
>>> fib[3] # Indexing
3
>>> fib[3:] # Slicing
[3, 5, 8, 13, 21, 34, 55]
```

Lists

- List is also a sequence type
  - Sequence operations are applicable



```
>>> [0] + fib # Concatenation
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> 3 * [1, 1, 2] # Repeatition
[1, 1, 2, 1, 1, 2, 1, 1, 2]
>>> x,y,z = [1,1,2] #Unpacking
>>> print(x, y, z)
1 1 2
```

More Operations on Lists

- L.append(x)
- L.extend(seq)
- L.insert(i, x)
- L.remove(x)
- L.pop(i)
- L.pop()
- L.index(x)
- L.count(x)
- L.sort()
- L.reverse()

x is any value, seq is a sequence value (list, string, tuple, ...),

i is an integer value

Summary of Sequences

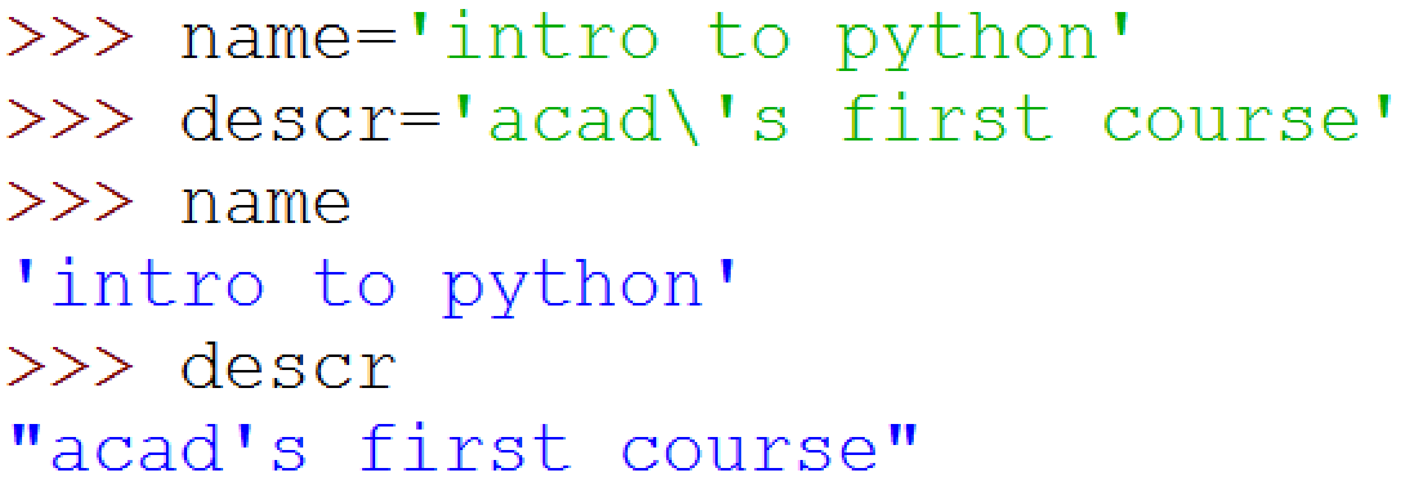
Sequence types include String, Tuple and List.
Lists are mutable, Tuple and Strings immutable.

Operation	Meaning
seq[i]	i-th element of the sequence
len(seq)	Length of the sequence
seq1 + seq2	Concatenate the two sequences
num*seq seq*num	Repeat seq num times
seq[start:end]	slice starting from **start**, and ending at **end-1**
e in seq	True if e is present is seq, False otherwise
e not in seq	True if e is not present is seq, False otherwise
for e in seq	Iterate over all elements in seq (e is bound to one element per iteration)

Strings

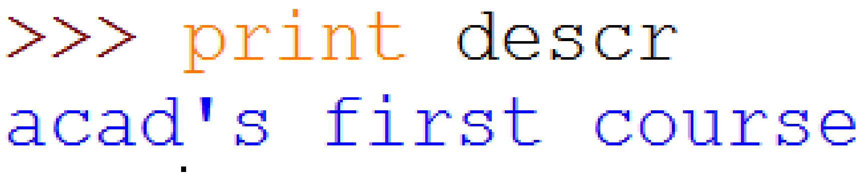
- Strings in Python have type str
- Sequence of characters
  - Python does not have a type corresponding to character.
- Strings are enclosed in single quotes(') or double quotes(“)
  - Both are equivalent
- Backslash (\) is used to escape quotes and special characters

Strings



```
>>> name='intro to python'
>>> descr='acad\'s first course'
>>> name
'intro to python'
>>> descr
"acad's first course"
```

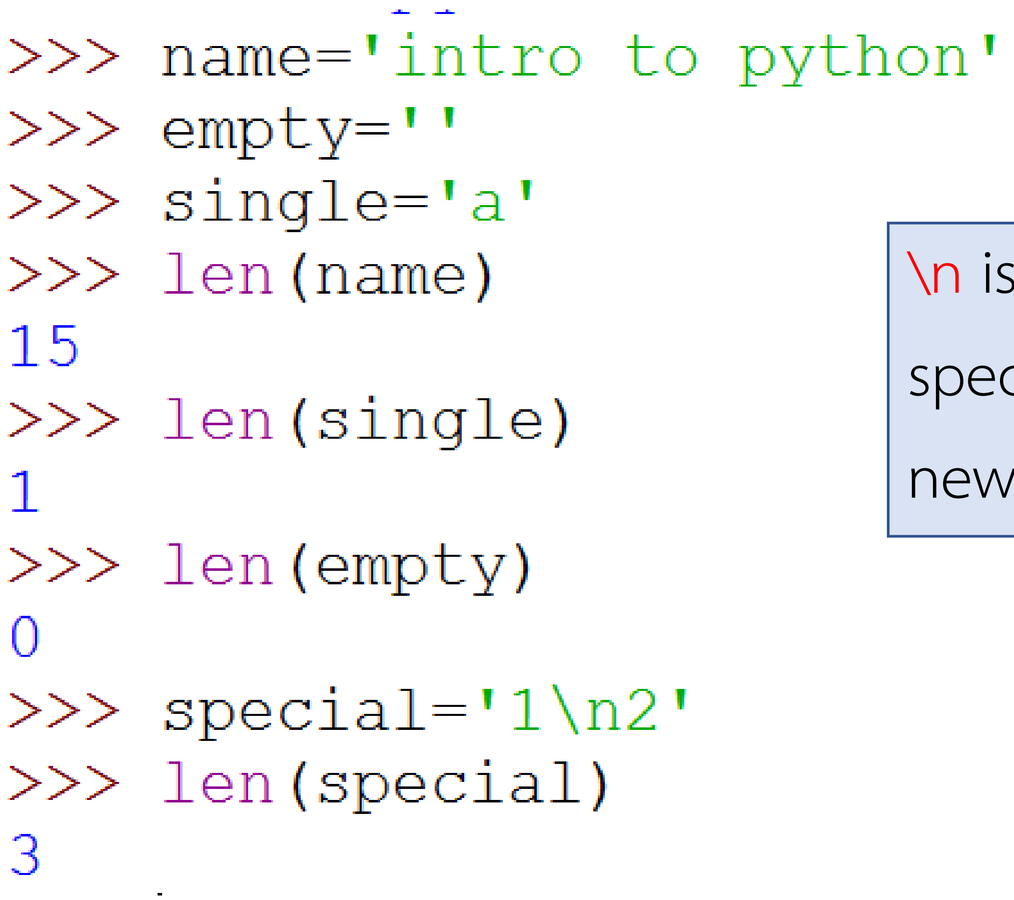
- More readable when print is used



```
>>> print descr
acad's first course
```

Length of a String

- len function gives the length of a string



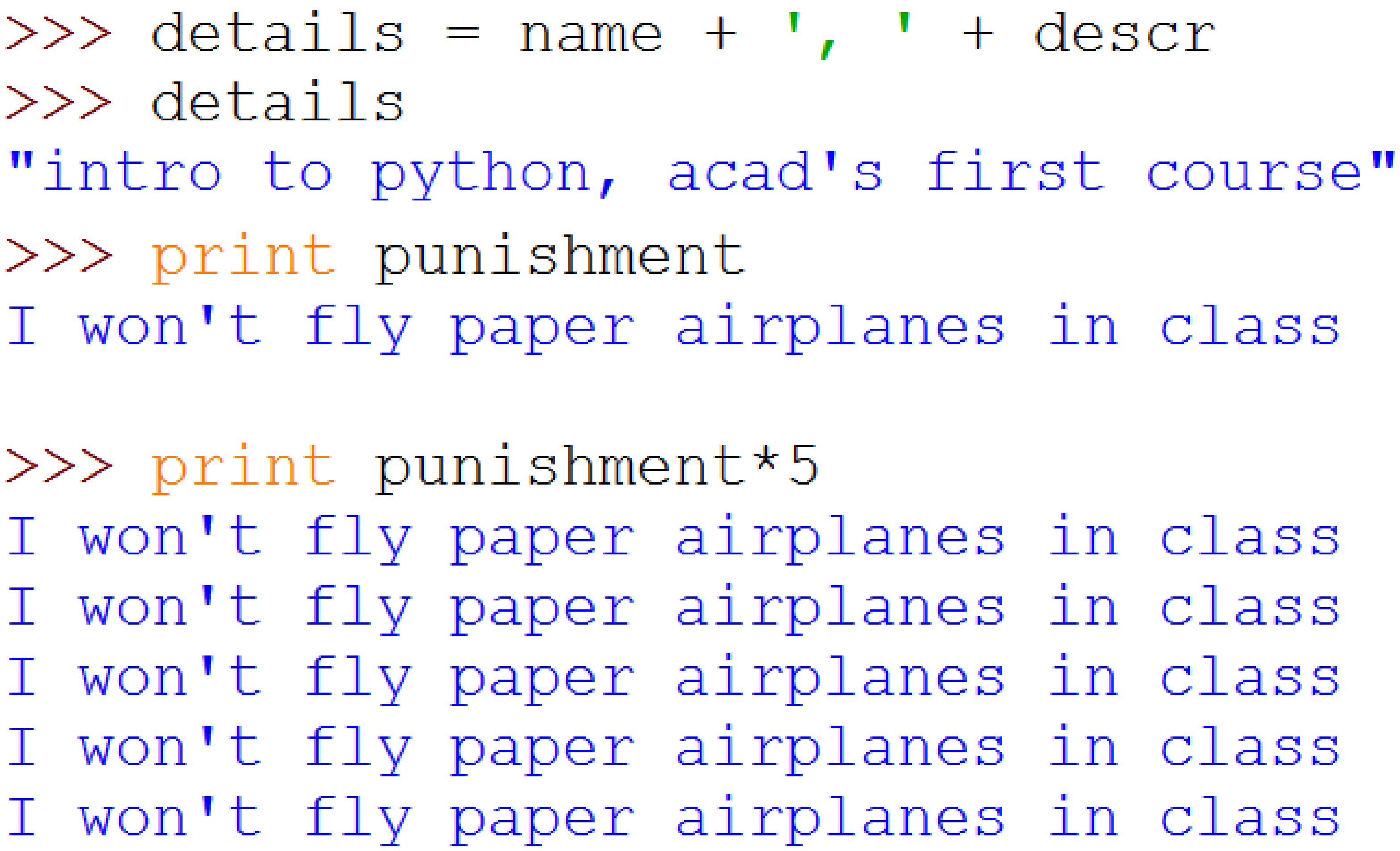
```
>>> name='intro to python'
>>> empty=''
>>> single='a'
>>> len(name)
15
>>> len(single)
1
>>> len(empty)
0
>>> special='1\n2'
>>> len(special)
3
```

\n is a **single** character: the special character representing newline

Concatenate and Repeat

- In Python, + and * operations have special meaning when operating on strings
  - + is used for concatenation of (two) strings
  - * is used to repeat a string, an int number of time
  - Function/Operator Overloading

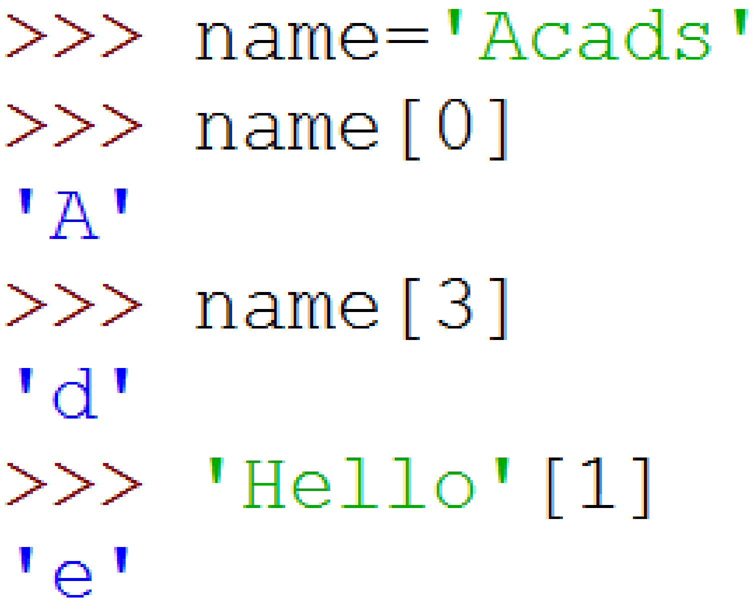
Concatenate and Repeat



```
>>> details = name + ', ' + descr
>>> details
"intro to python, acad's first course"
>>> print punishment
I won't fly paper airplanes in class

>>> print punishment*5
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
```

Indexing

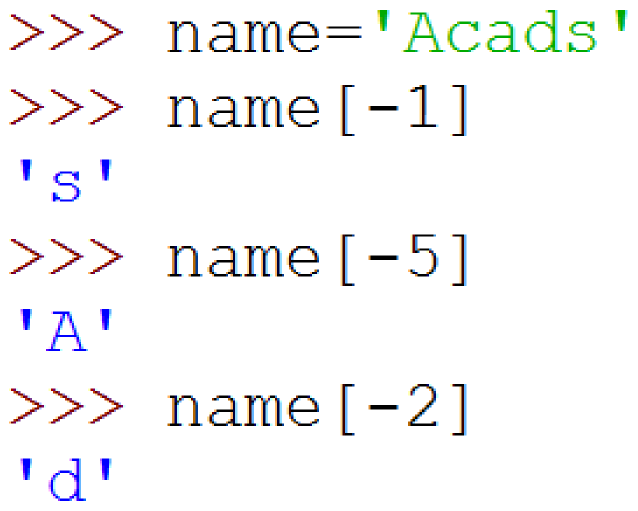
- Strings can be indexed
- First character has index 0



```
>>> name='Acads'
>>> name[0]
'A'
>>> name[3]
'd'
>>> 'Hello'[1]
'e'
```

Indexing

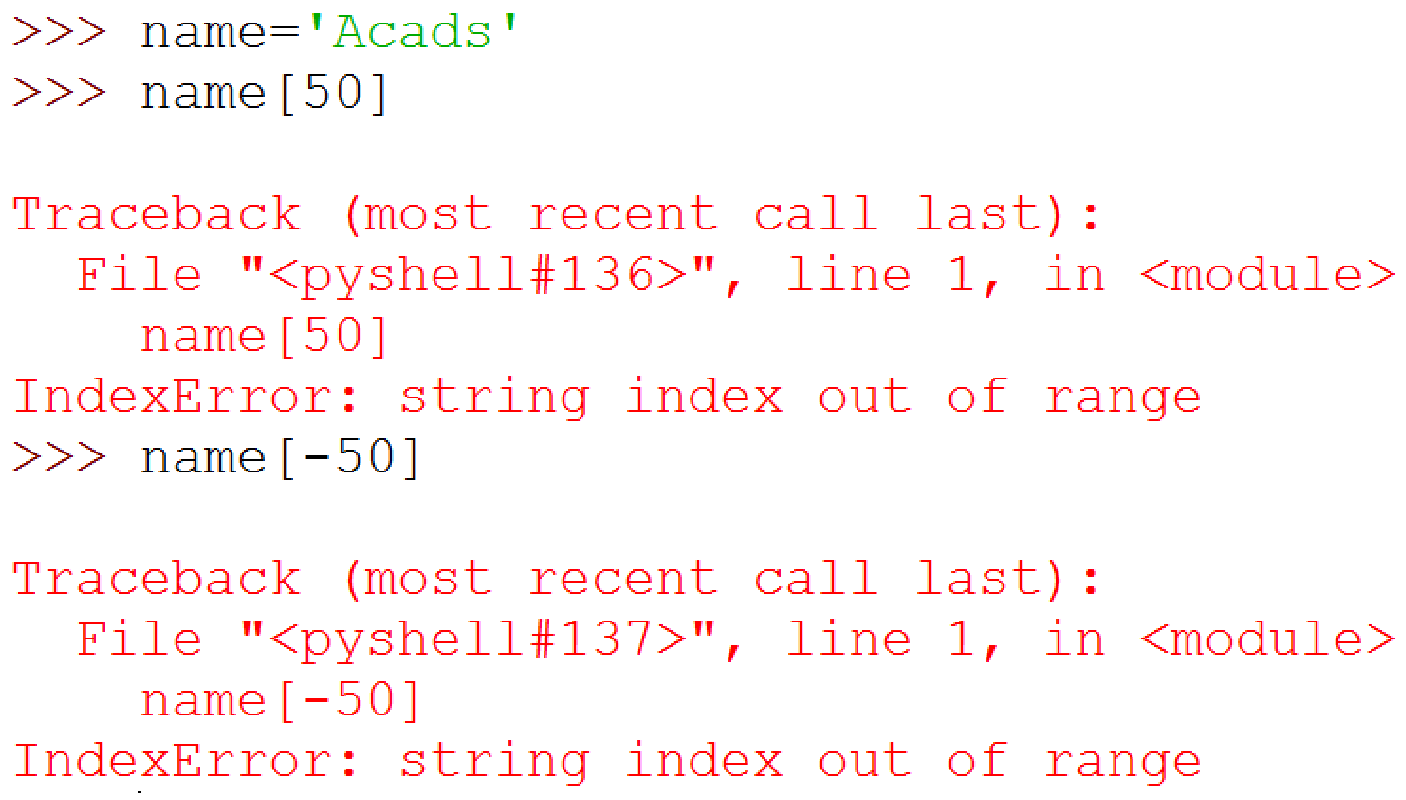
- Negative indices start counting from the right
- Negatives indices start from -1
- -1 means last, -2 second last, ...



```
>>> name='Acads'
>>> name[-1]
's'
>>> name[-5]
'A'
>>> name[-2]
'd'
```

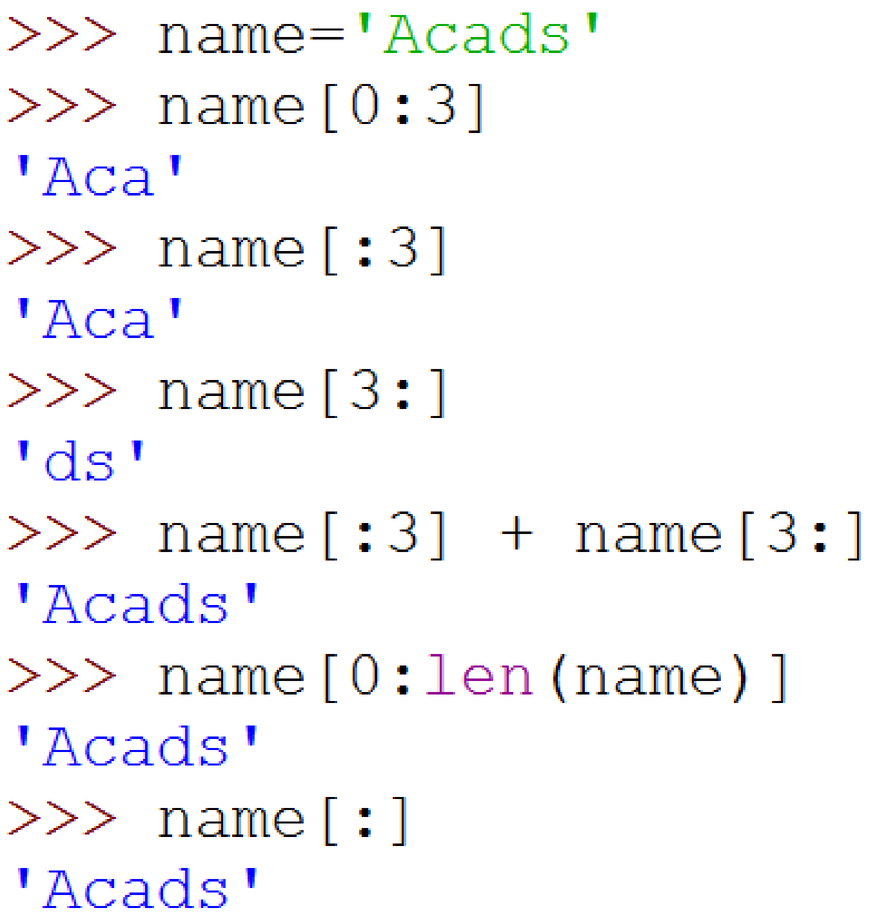
Indexing

- Using an index that is too large or too small results in “index out of range” error



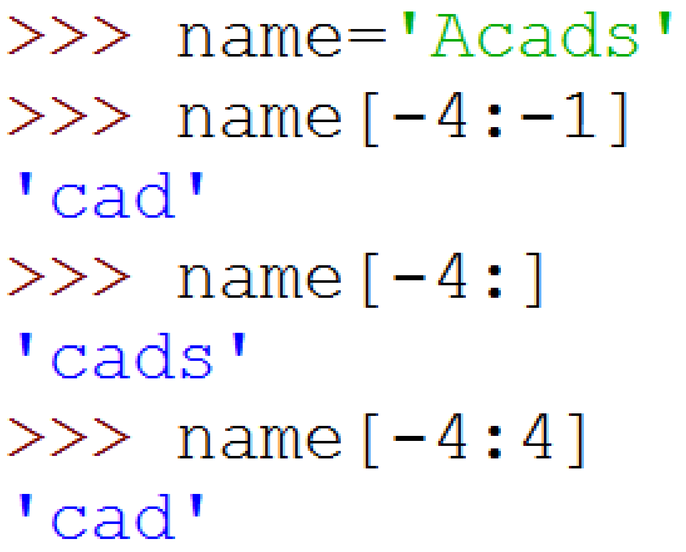
```
>>> name='Acads'
>>> name[50]

Traceback (most recent call last):
  File "<pyshell#136>", line 1, in <module>
    name[50]
IndexError: string index out of range
>>> name[-50]

Traceback (most recent call last):
  File "<pyshell#137>", line 1, in <module>
    name[-50]
IndexError: string index out of range
```

Slicing



```
>>> name='Acads'
>>> name[0:3]
'Aca'
>>> name[:3]
'Aca'
>>> name[3:]
'ds'
>>> name[:3] + name[3:]
'Acads'
>>> name[0:len(name)]
'Acads'
>>> name[:]
'Acads'
```

More Slicing



```
>>> name='Acads'
>>> name[-4:-1]
'cad'
>>> name[-4:]
'cads'
>>> name[-4:4]
'cad'
```

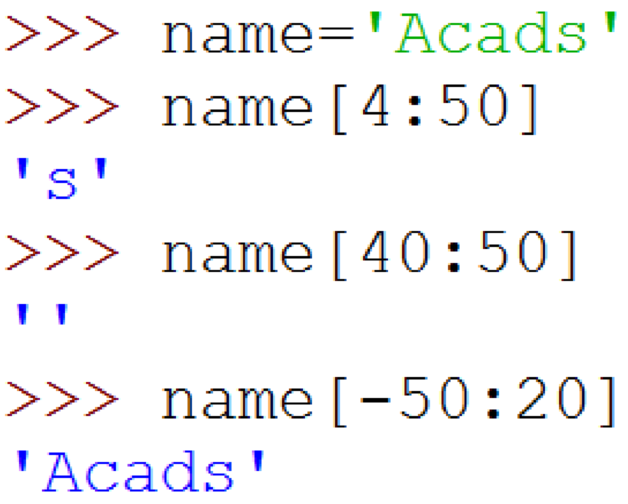
Understanding Indices for slicing

A	c	a	d	s	
0	1	2	3	4	5
-5	-4	-3	-2	-1	

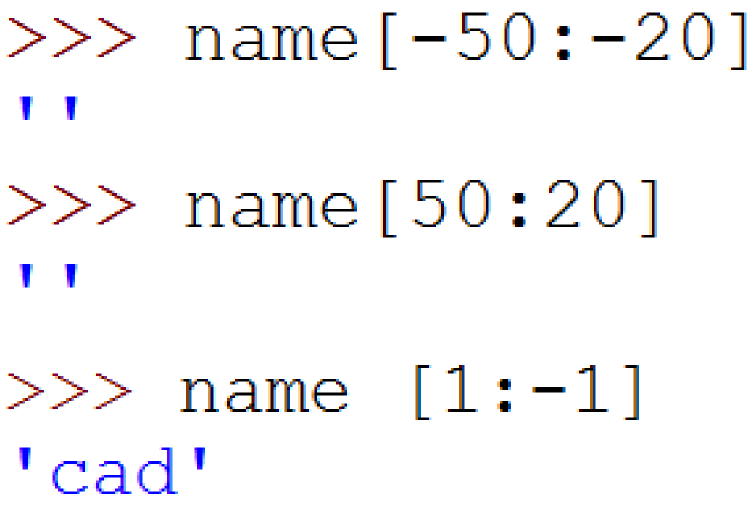
Out of Range Slicing

A	c	a	d	s
0	1	2	3	4
-5	-4	-3	-2	-1

- Out of range indices are ignored for slicing
- when start and end have the same sign, if start >=end, empty slice is returned



```
>>> name='Acads'
>>> name[4:50]
's'
>>> name[40:50]
''
>>> name[-50:20]
'Acads'
```



```
>>> name[-50:-20]
''
>>> name[50:20]
''
>>> name [1:-1]
'cad'
```

Built-in Data Structures

- Tupples: <u>constant</u> arrays

  b = (1,2,3)

- Can't change each b value

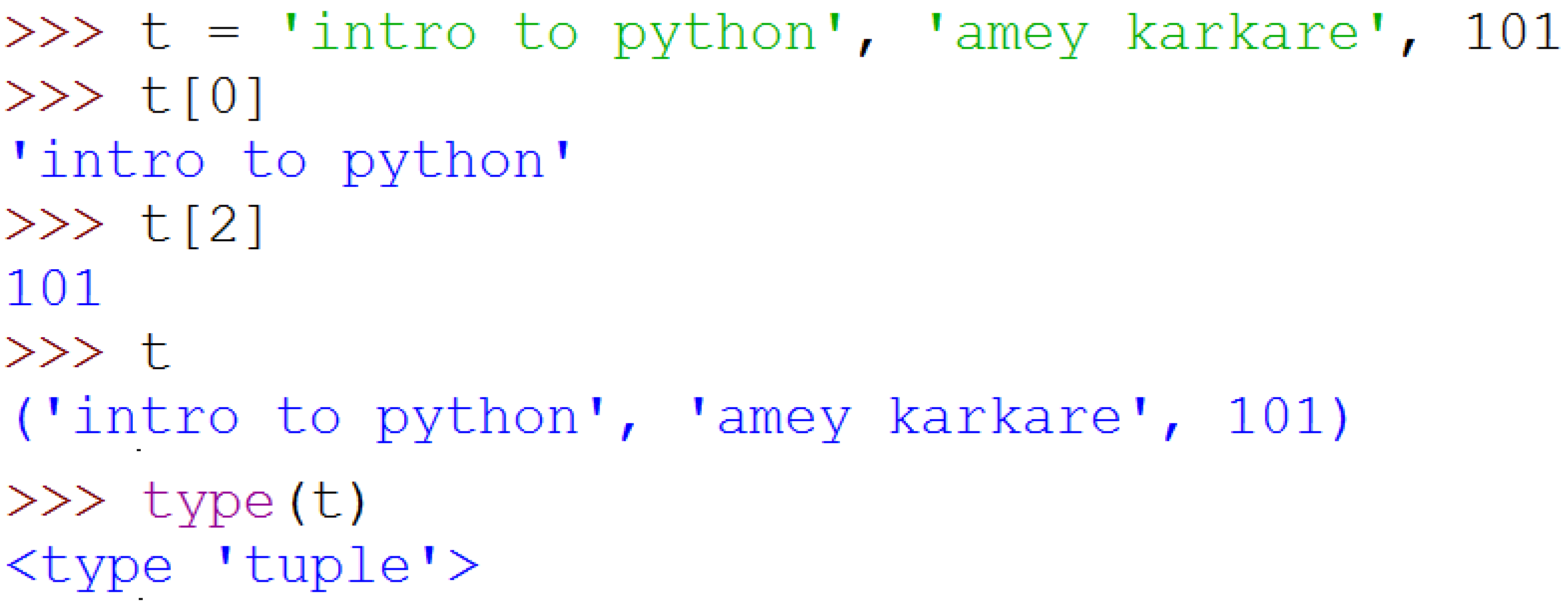
  b[2] = 10 !!!

- Can use each data like array

  print (b[2]) # output = 3

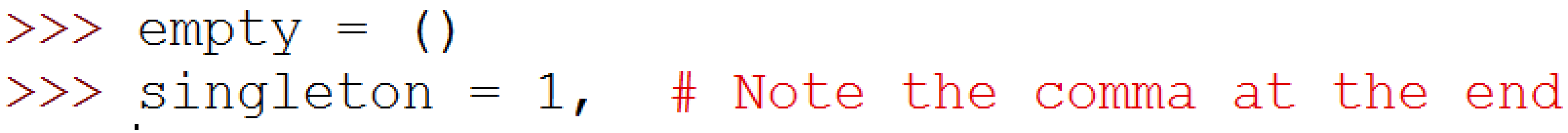
Tuples

- A tuple consists of a number of values separated by commas



```
>>> t = 'intro to python', 'amey karkare', 101
>>> t[0]
'intro to python'
>>> t[2]
101
>>> t
('intro to python', 'amey karkare', 101)
>>> type(t)
<type 'tuple'>
```

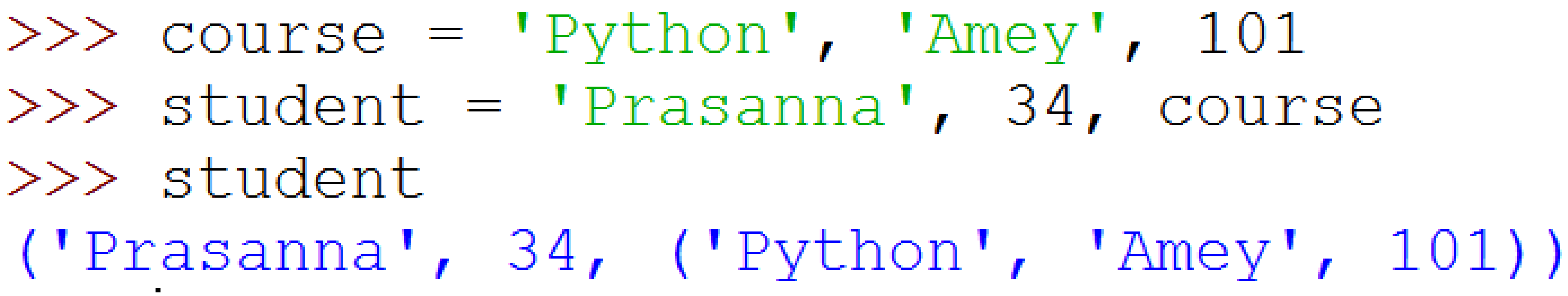
- Empty and Singleton Tuples



```
>>> empty = ()
>>> singleton = 1,   # Note the comma at the end
```

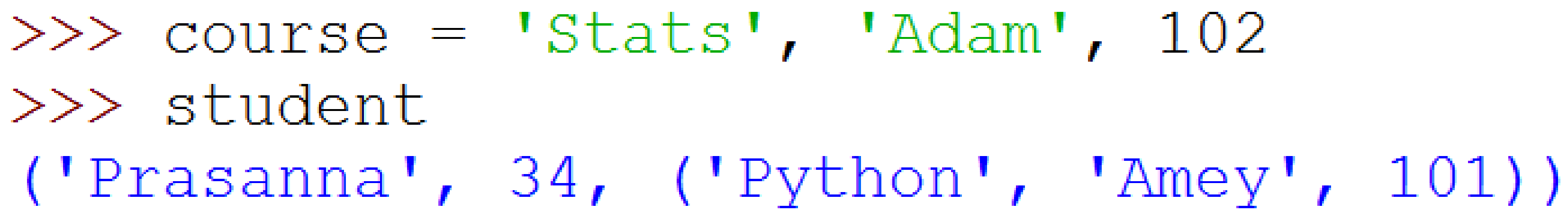
Nested Tuples

- Tuples can be nested



```
>>> course = 'Python', 'Amey', 101
>>> student = 'Prasanna', 34, course
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
```

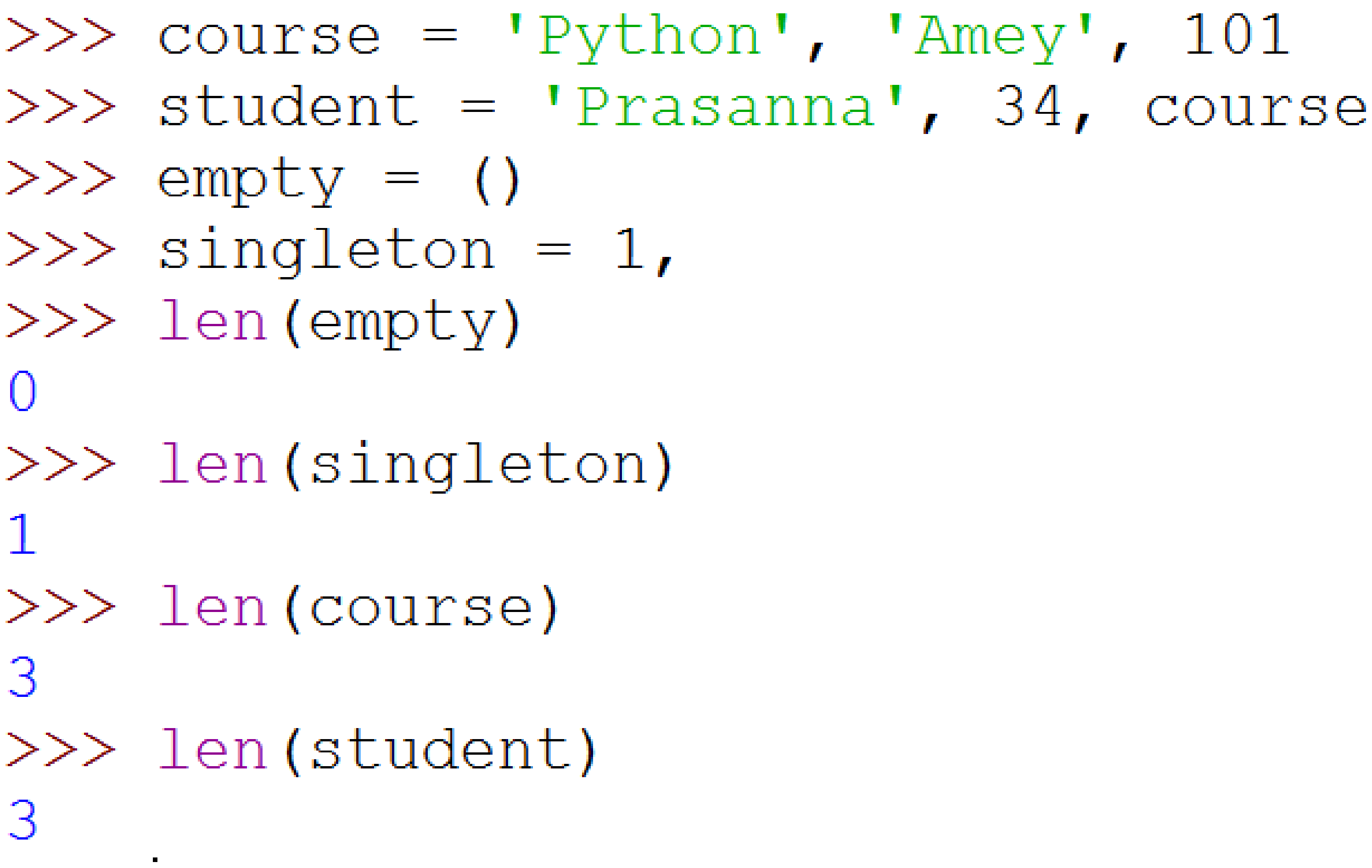
- Note that course tuple is copied into student.
  - Changing course does not affect student



```
>>> course = 'Stats', 'Adam', 102
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
```

Length of a Tuple

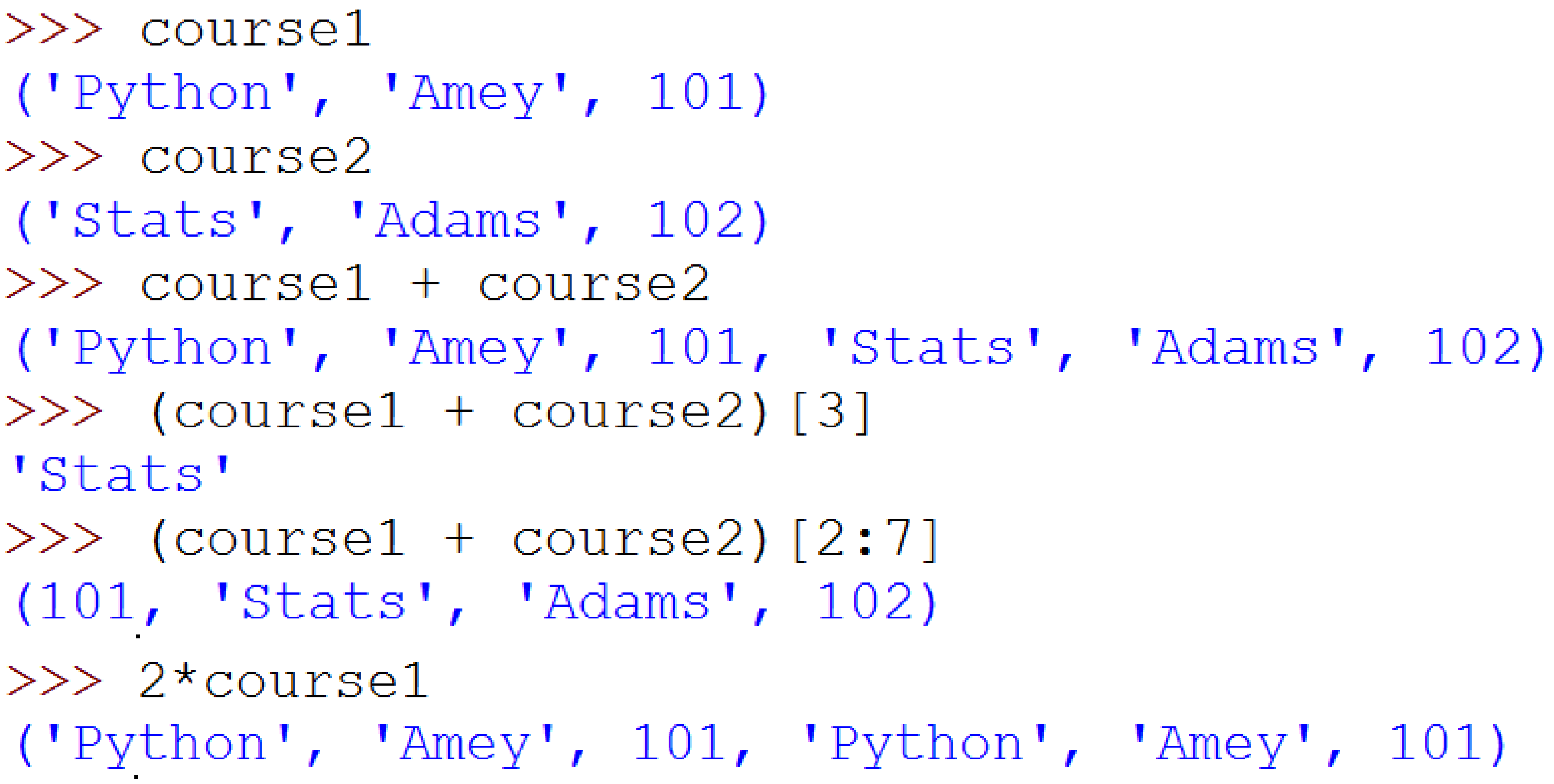
- len function gives the length of a tuple



```
>>> course = 'Python', 'Amey', 101
>>> student = 'Prasanna', 34, course
>>> empty = ()
>>> singleton = 1,
>>> len(empty)
0
>>> len(singleton)
1
>>> len(course)
3
>>> len(student)
3
```

More Operations on Tuples

- Tuples can be concatenated, repeated, indexed and sliced

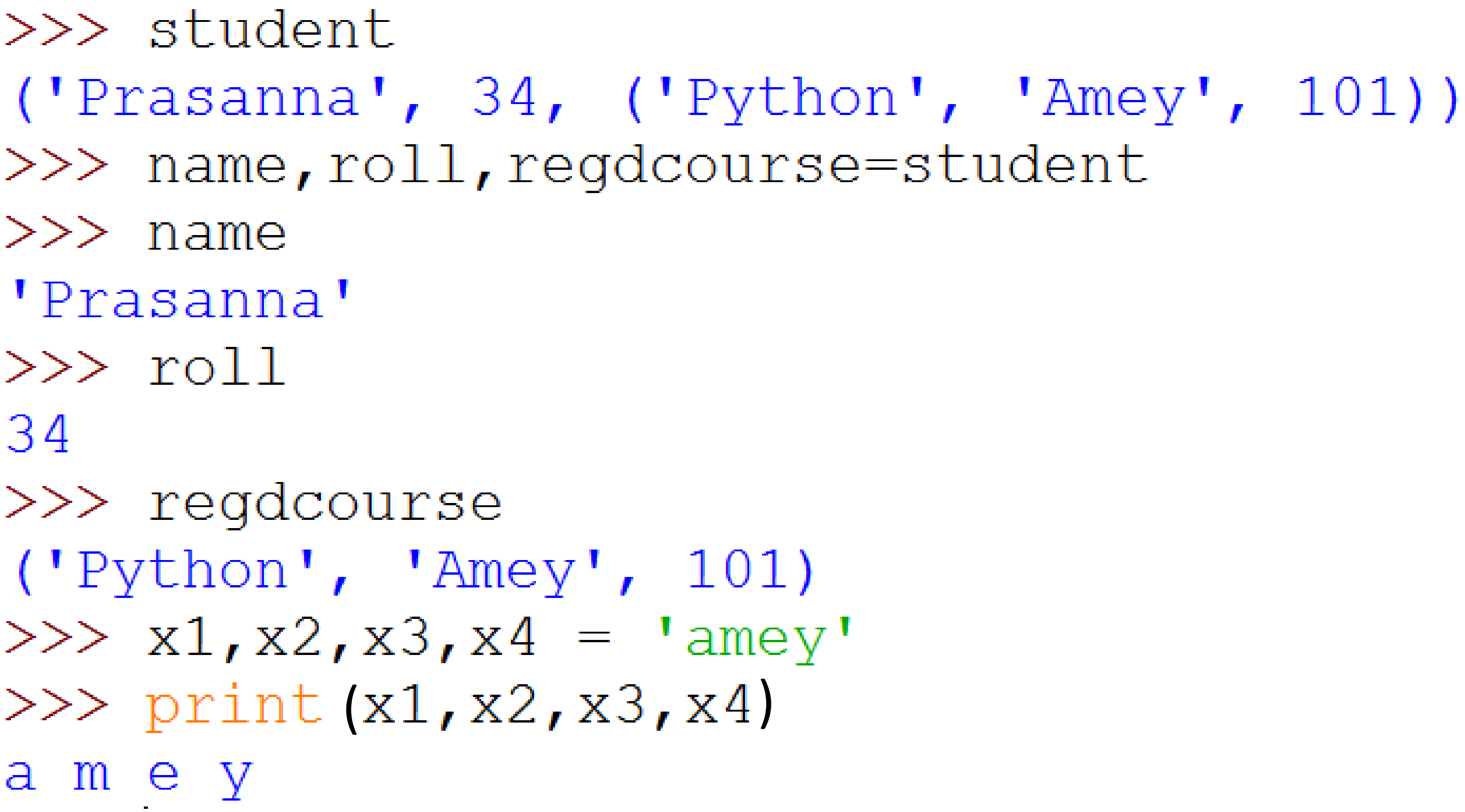


```
>>> course1
('Python', 'Amey', 101)
>>> course2
('Stats', 'Adams', 102)
>>> course1 + course2
('Python', 'Amey', 101, 'Stats', 'Adams', 102)
>>> (course1 + course2)[3]
'Stats'
>>> (course1 + course2)[2:7]
(101, 'Stats', 'Adams', 102)
>>> 2*course1
('Python', 'Amey', 101, 'Python', 'Amey', 101)
```

Unpacking Sequences

- Strings and Tuples are examples of sequences
  - Indexing, slicing, concatenation, repetition operations applicable on sequences
- Sequence Unpacking operation can be applied to sequences to get the components
  - *Multiple assignment* statement
  - LHS and RHS must have equal length

Unpacking Sequences



```
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
>>> name,roll,regdcourse=student
>>> name
'Prasanna'
>>> roll
34
>>> regdcourse
('Python', 'Amey', 101)
>>> x1,x2,x3,x4 = 'amey'
>>> print(x1,x2,x3,x4)
a m e y
```

Built-in Data Structures

Dictionaries : association lists with fast access

Source Code :

```
x = {}
x['abc']=2
x['def']=5
for k in x.keys():
  print(k,x[k])
```

Output :

```
abc 2
def 5
```

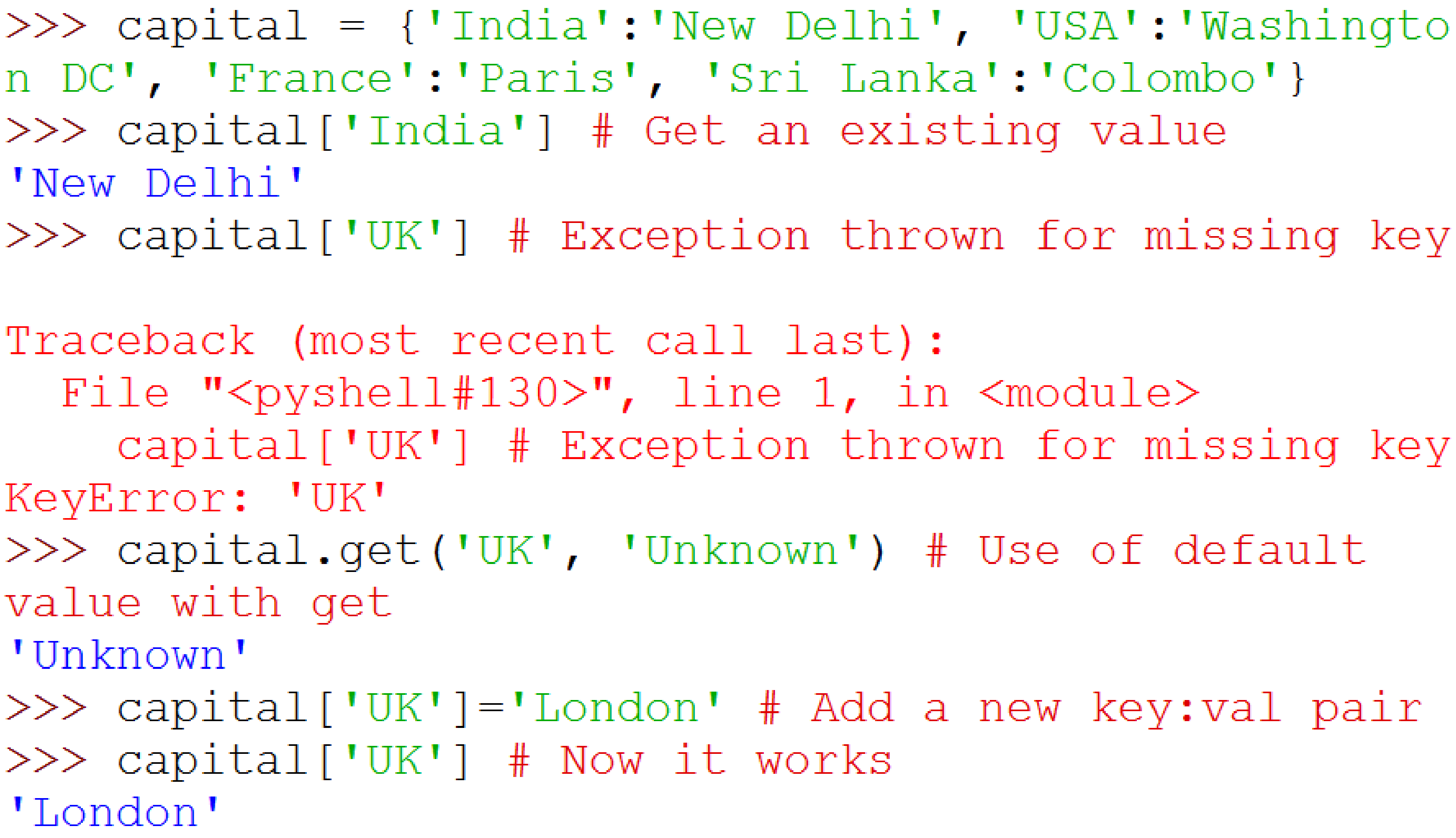
Dictionaries

- Unordered set of *key:value* pairs,
- Keys have to be unique and immutable
- Key:value pairs enclosed inside curly braces {...}
- Empty dictionary is created by writing {}
- Dictionaries are mutable
  - add new key:value pairs,
  - change the pairing
  - delete a key (and associated value)

Operations on Dictionaries

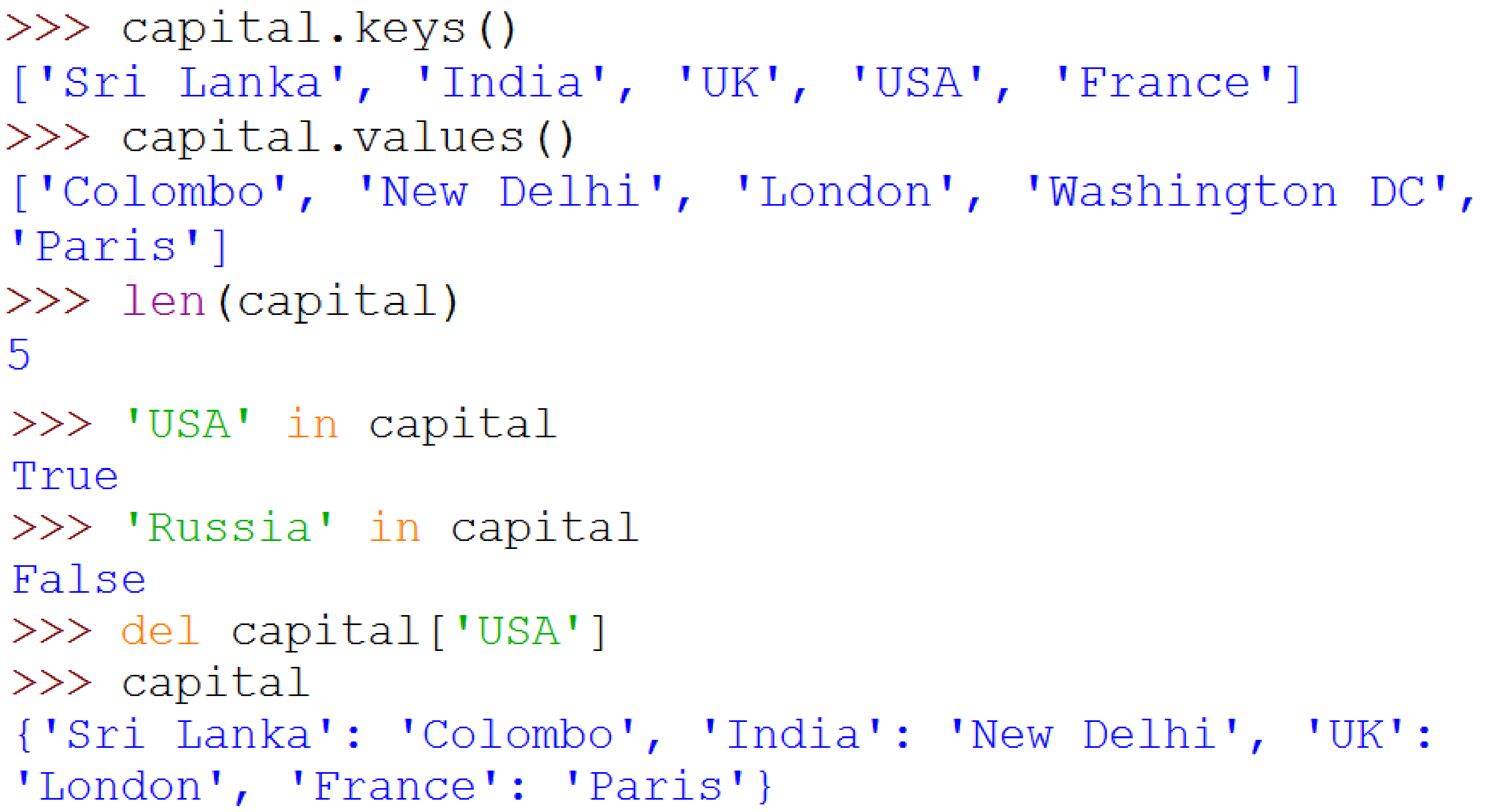
Operation	Meaning
len(d)	Number of key:value pairs in d
d.keys()	List containing the keys in d
d.values()	List containing the values in d
k in d	True if key k is in d
d[k]	Value associated with key k in d
d.get(k, v)	If k is present in d, then d[k] else v
d[k] = v	Map the value v to key k in d (replace d[k] if present)
del d[k]	Remove key k (and associated value) from d
for k in d	Iterate over the keys in d

Operations on Dictionaries



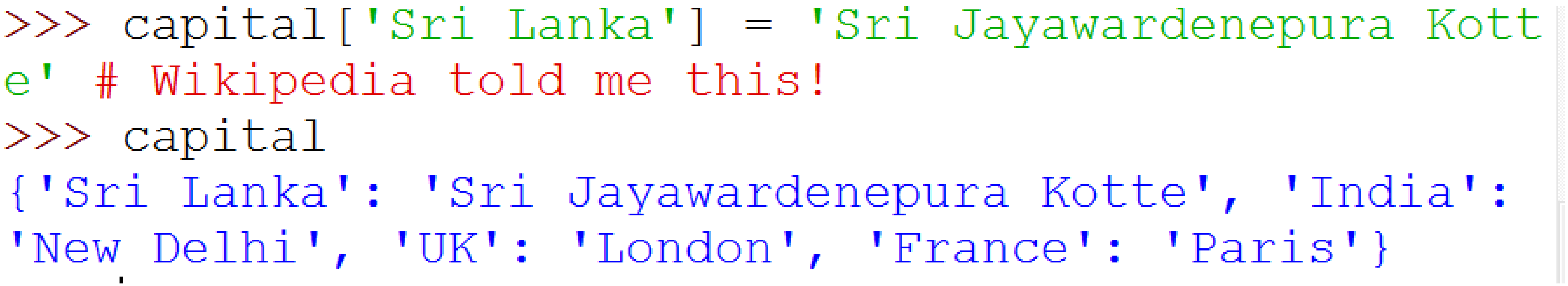
```
>>> capital = {'India':'New Delhi', 'USA':'Washingto
n DC', 'France':'Paris', 'Sri Lanka':'Colombo'}
>>> capital['India'] # Get an existing value
'New Delhi'
>>> capital['UK'] # Exception thrown for missing key

Traceback (most recent call last):
  File "<pyshell#130>", line 1, in <module>
    capital['UK'] # Exception thrown for missing key
KeyError: 'UK'
>>> capital.get('UK', 'Unknown') # Use of default
value with get
'Unknown'
>>> capital['UK']='London' # Add a new key:val pair
>>> capital['UK'] # Now it works
'London'
```

Operations on Dictionaries

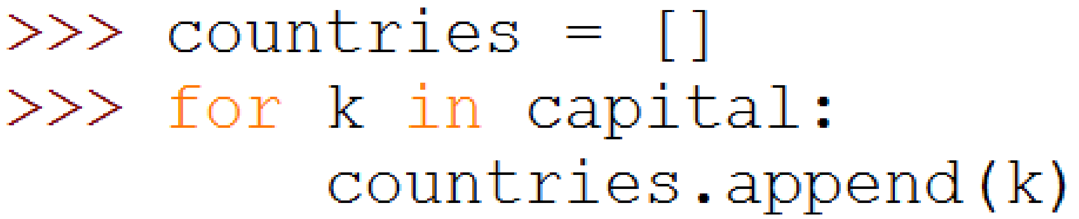


```
>>> capital.keys()
['Sri Lanka', 'India', 'UK', 'USA', 'France']
>>> capital.values()
['Colombo', 'New Delhi', 'London', 'Washington DC',
'Paris']
>>> len(capital)
5
>>> 'USA' in capital
True
>>> 'Russia' in capital
False
>>> del capital['USA']
>>> capital
{'Sri Lanka': 'Colombo', 'India': 'New Delhi', 'UK':
'London', 'France': 'Paris'}
```

Operations on Dictionaries

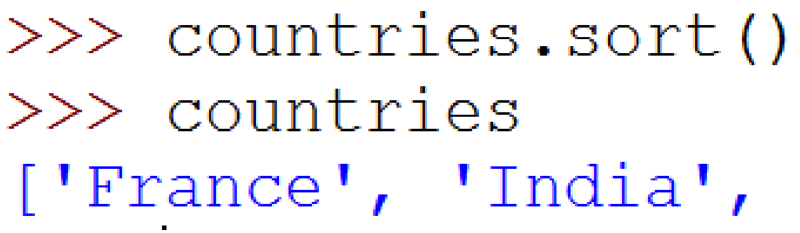


```
>>> capital['Sri Lanka'] = 'Sri Jayawardenepura Kott
e' # Wikipedia told me this!
>>> capital
{'Sri Lanka': 'Sri Jayawardenepura Kotte', 'India':
'New Delhi', 'UK': 'London', 'France': 'Paris'}
```



```
>>> countries = []
>>> for k in capital:
        countries.append(k)
```

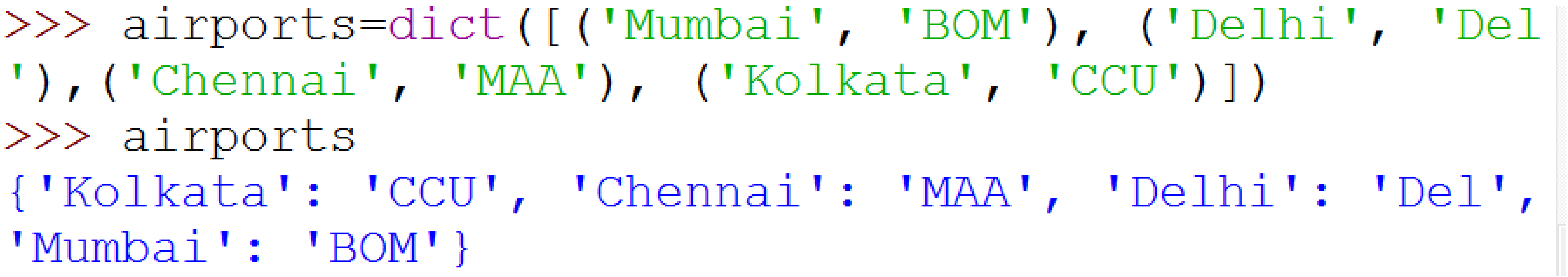
# Remember: for ... in iterates over keys only



```
>>> countries.sort()  # Sort values in a list
>>> countries
['France', 'India', 'Sri Lanka', 'UK']
```

Dictionary Construction

- The dict constructor: builds dictionaries directly from *sequences of key-value pairs*

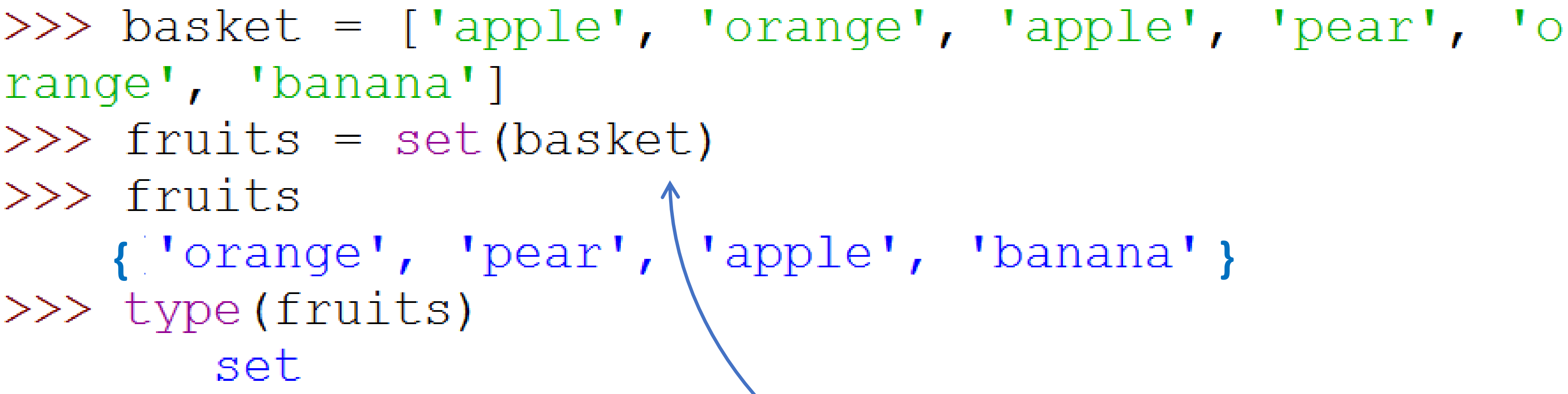


```
>>> airports=dict([('Mumbai', 'BOM'), ('Delhi', 'Del
'),('Chennai', 'MAA'), ('Kolkata', 'CCU')])
>>> airports
{'Kolkata': 'CCU', 'Chennai': 'MAA', 'Delhi': 'Del',
'Mumbai': 'BOM'}
```

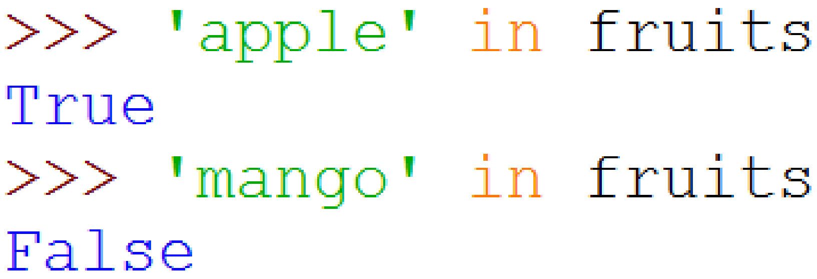
Sets

- An unordered collection with no duplicate elements
- Supports
  - membership testing
  - eliminating duplicate entries
  - Set operations: union, intersection, difference, and symmetric difference.

Sets



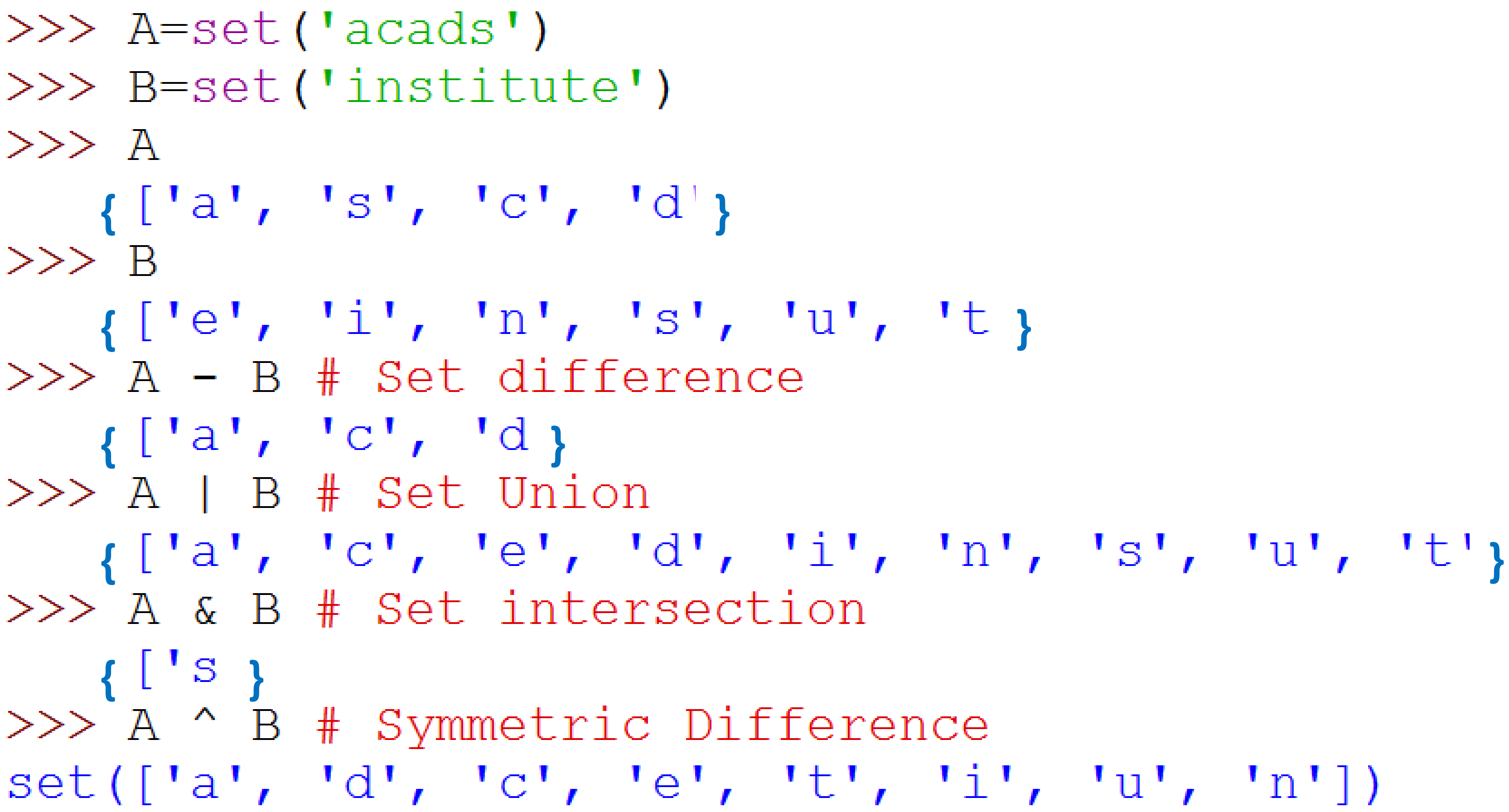
```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'o
range', 'banana']
>>> fruits = set(basket)
>>> fruits
    {'orange', 'pear', 'apple', 'banana'}
>>> type(fruits)
        set
```



```
>>> 'apple' in fruits
True
>>> 'mango' in fruits
False
```

Create a set from a sequence

Set Operations



```
>>> A=set('acads')
>>> B=set('institute')
>>> A
    {['a', 's', 'c', 'd'}
>>> B
    {['e', 'i', 'n', 's', 'u', 't }
>>> A - B # Set difference
    {['a', 'c', 'd }
>>> A | B # Set Union
    {['a', 'c', 'e', 'd', 'i', 'n', 's', 'u', 't'}
>>> A & B # Set intersection
    {['s }
>>> A ^ B # Symmetric Difference
set(['a', 'd', 'c', 'e', 't', 'i', 'u', 'n'])
```

Condition

Source code :

```
y = 10
if y <0 :
  print("negative")
elif y == 0:
  print("zero")
else:
  print("positive")
```

Output :

```
positive
```

Loop

Source code :

```
for i in range(0,5):
         print(i)
```

Output:

```
0
1
2
3
4
```

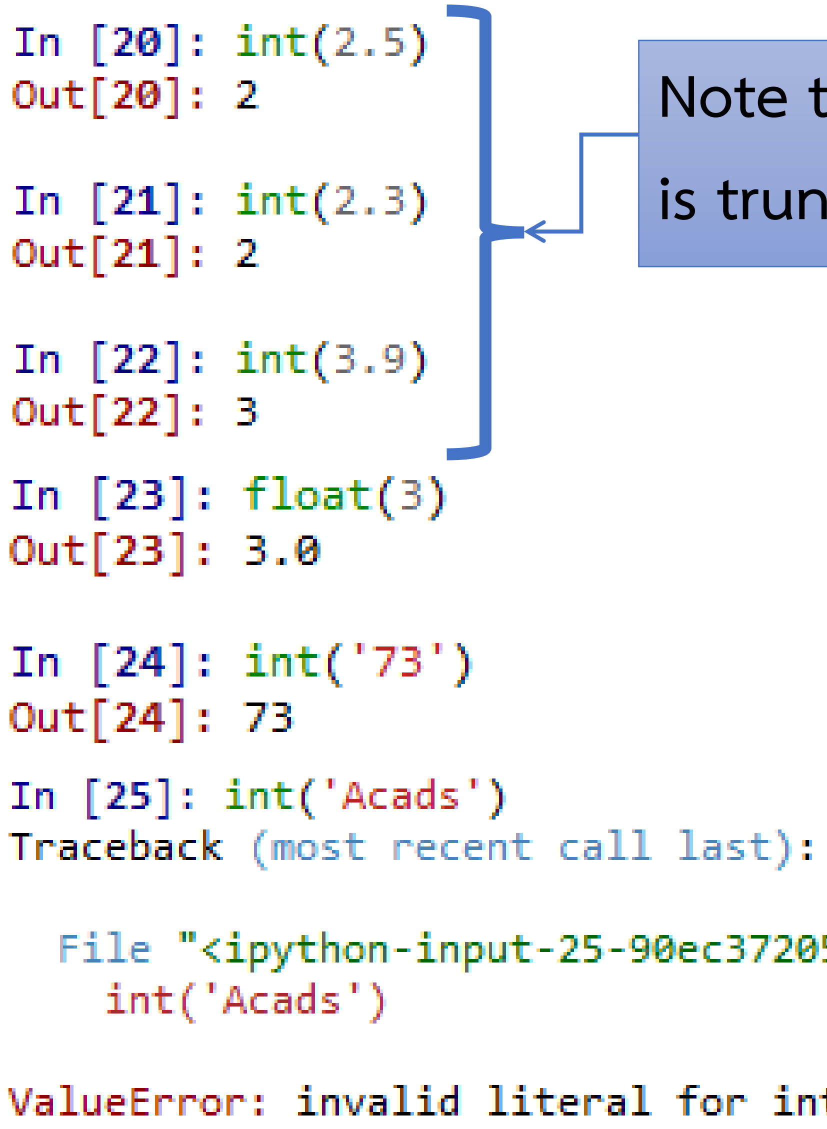
Source code :

```
p = 0
while p<5:
  print(p)
  p+=1
```

Output:

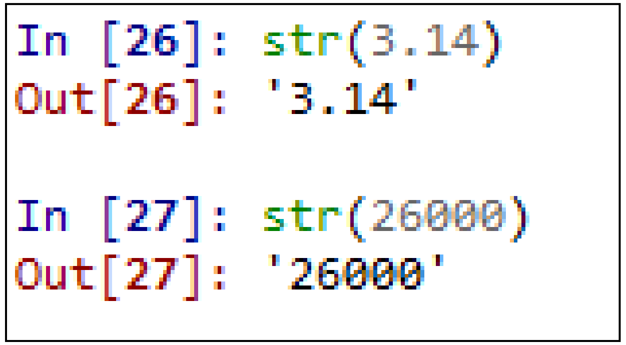
```
0
1
2
3
4
```

Type Conversion Examples



```
In [20]: int(2.5)
Out[20]: 2

In [21]: int(2.3)
Out[21]: 2

In [22]: int(3.9)
Out[22]: 3
In [23]: float(3)
Out[23]: 3.0

In [24]: int('73')
Out[24]: 73
In [25]: int('Acads')
Traceback (most recent call last):

  File "<ipython-input-25-90ec37205222>", line 1, in <module>
    int('Acads')

ValueError: invalid literal for int() with base 10: 'Acads'
```

Note that float to int conversion is truncation, not rounding off



```
In [26]: str(3.14)
Out[26]: '3.14'

In [27]: str(26000)
Out[27]: '26000'
```

Booleans

- Truth values: True and False.
- False is equivalent with 0, and empty list [], an empty dictionary {}.
- Anything else is equivalent to True.
- Example:

```
x = 0
if not x:
    print “0 is False”
```

Functions and Parameters

- Function definition:

```
def function_name (par1, par2, ...):
    body of the function
```

```
def max(a,b):
    if a > b:
        return a
    else:
        return b
```

- It supports default values for parameters.
- All parameters are value parameters.
- Any variable storing a complex data structure contains a reference to it. Any changes to the content of such a data structure in the function will affect the variable passed in the function call.
- Assignments involving a complex data structure don't make a copy of it.

Default Values for Parameters

Default values:

- def function (var1 = value, var2 = value, …):
- def GCD1(a=10, b=20): …
- GCD1() -> 10
- GCD1(125) -> 5
- GCD1(12, 39) -> 3

File I/O

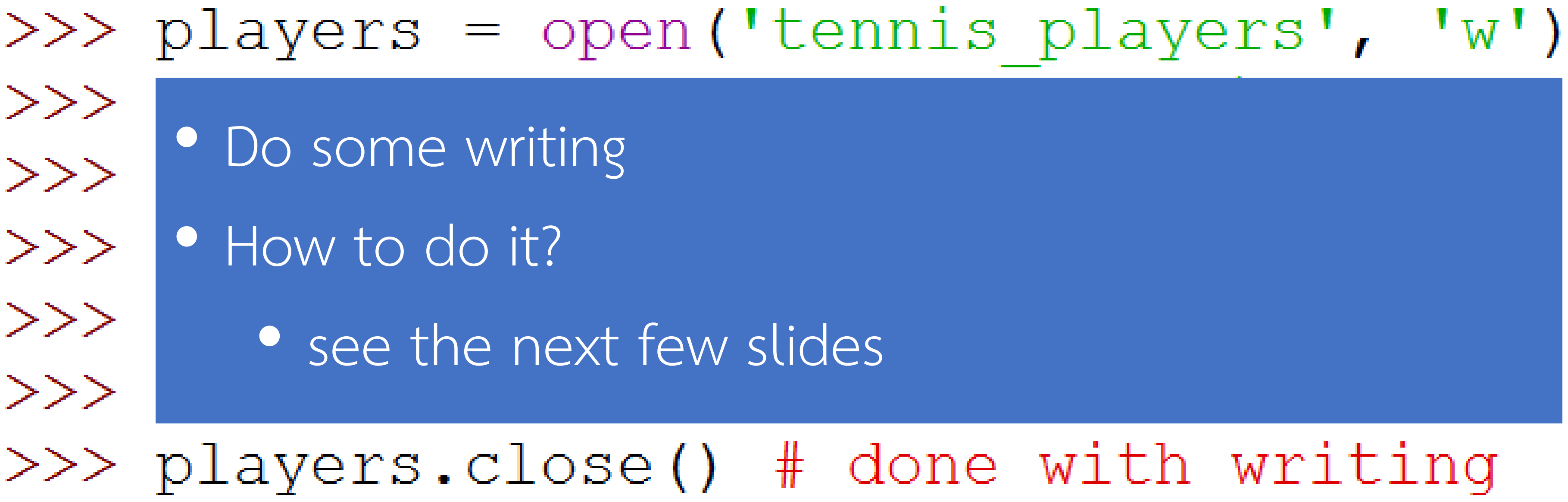
- Files are persistent storage
- Allow data to be stored beyond program lifetime
- The basic operations on files are
  - open, close, read, write
- Python treat files as sequence of lines
  - sequence operations work for the data read from files

File I/O: open and close

open(filename, mode)

- While opening a file, you need to supply
  - The name of the file, including the path
  - The mode in which you want to open a file
  - Common modes are r (read), w (write), a (append)
- Mode is optional, defaults to r
- open(..) returns a file object
- close() on the file object closes the file
  - finishes any buffered operations

File I/O: Example



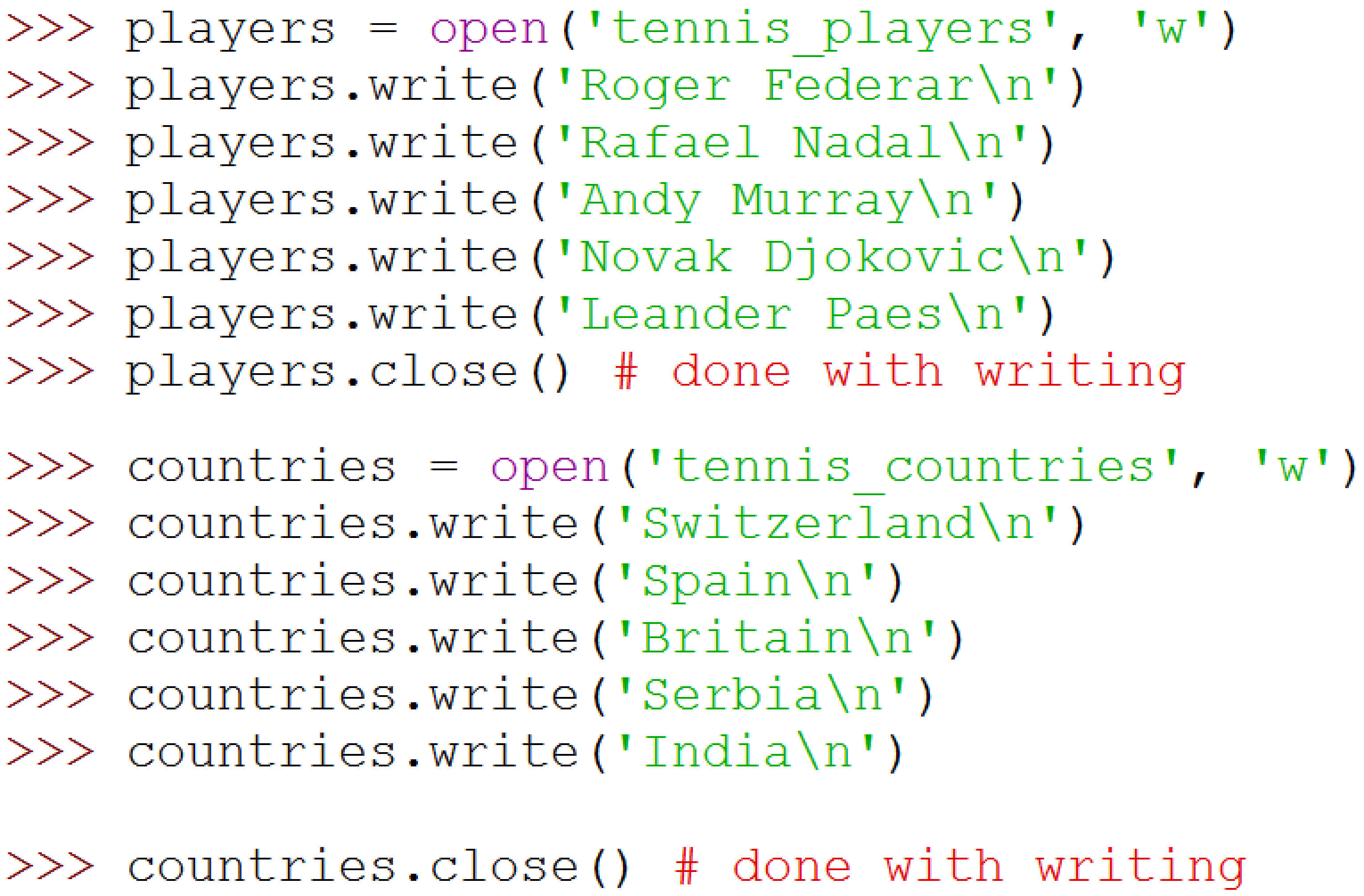
```
>>> players = open('tennis_players', 'w')
>>>
>>>
>>>
>>>
>>>
>>> players.close() # done with writing
```

- Do some writing
- How to do it?
  - see the next few slides

File I/O: read, write and append

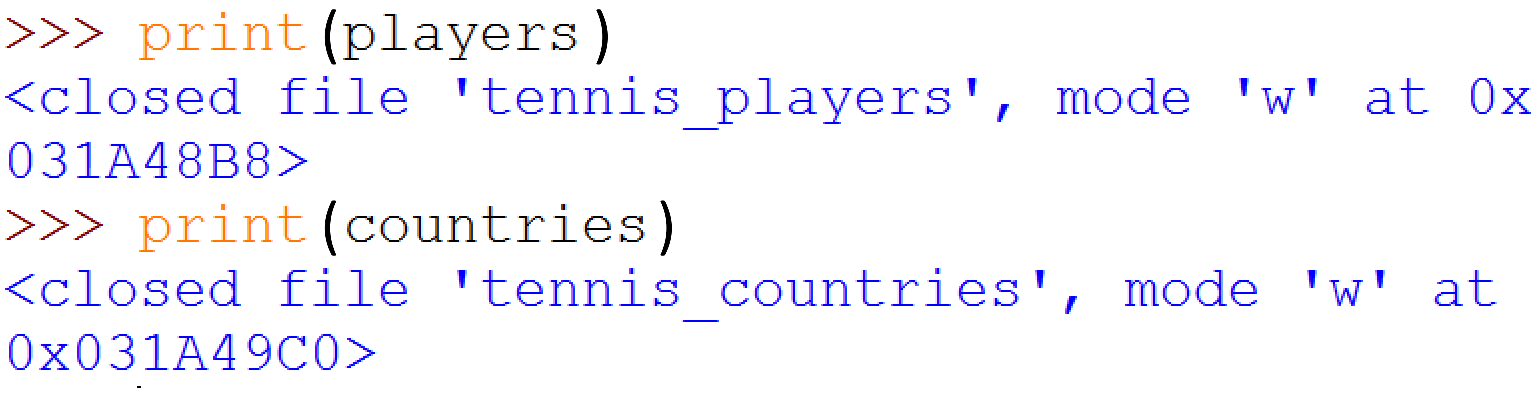
- Reading from an open file returns the contents of the file
  - as **sequence** of lines in the program
- Writing to a file
  - IMPORTANT: If opened with mode 'w', **clears** the existing contents of the file
  - Use append mode ('a') to preserve the contents
  - Writing happens at the end

File I/O: Examples



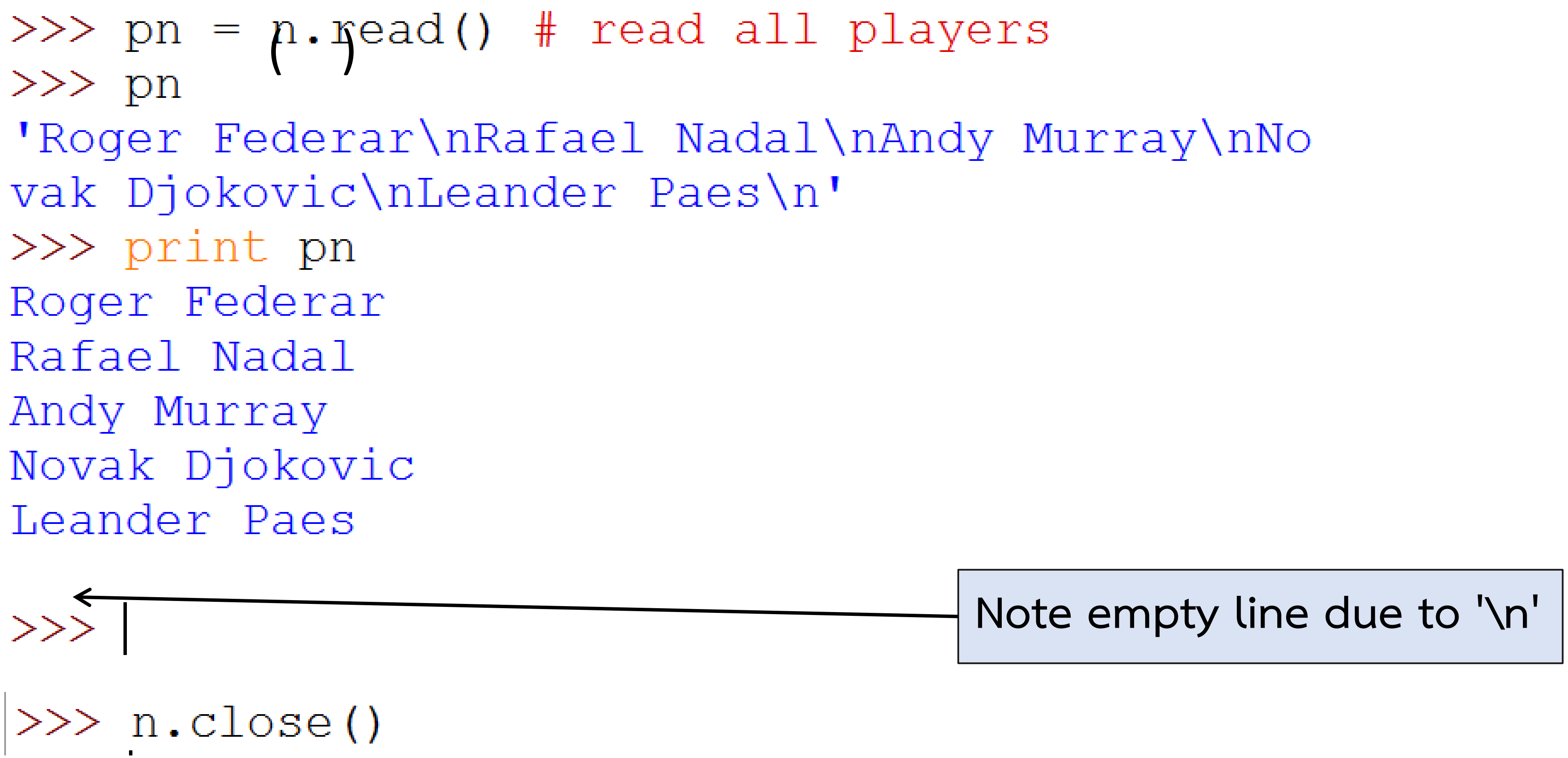
```
>>> players = open('tennis_players', 'w')
>>> players.write('Roger Federar\n')
>>> players.write('Rafael Nadal\n')
>>> players.write('Andy Murray\n')
>>> players.write('Novak Djokovic\n')
>>> players.write('Leander Paes\n')
>>> players.close() # done with writing

>>> countries = open('tennis_countries', 'w')
>>> countries.write('Switzerland\n')
>>> countries.write('Spain\n')
>>> countries.write('Britain\n')
>>> countries.write('Serbia\n')
>>> countries.write('India\n')

>>> countries.close() # done with writing
```

File I/O: Examples



```
>>> print(players)
<closed file 'tennis_players', mode 'w' at 0x
031A48B8>
>>> print(countries)
<closed file 'tennis_countries', mode 'w' at
0x031A49C0>
```

File I/O: Examples

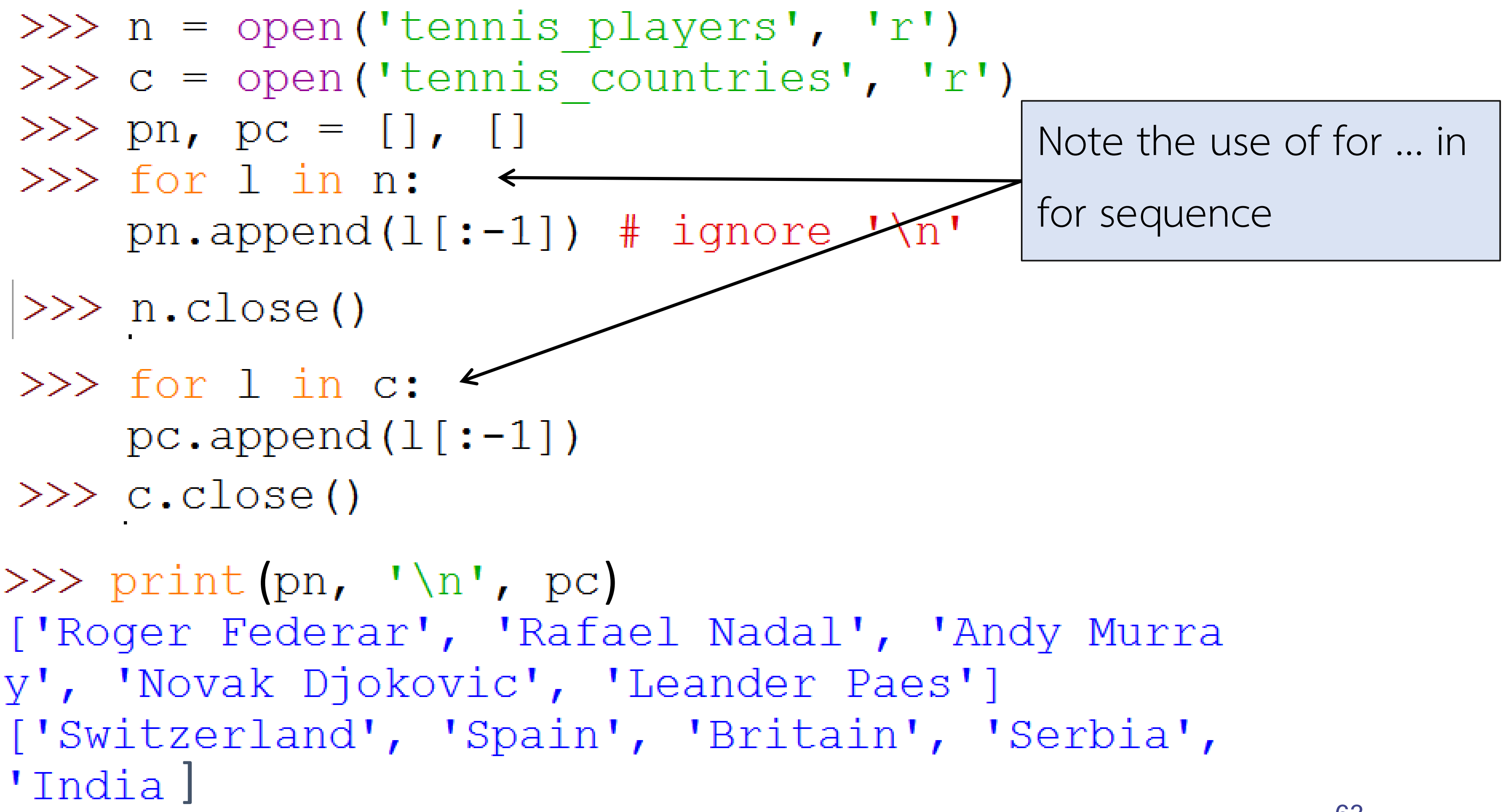


```
>>> pn = n.read() # read all players
>>> pn
'Roger Federar\nRafael Nadal\nAndy Murray\nNo
vak Djokovic\nLeander Paes\n'
>>> print pn
Roger Federar
Rafael Nadal
Andy Murray
Novak Djokovic
Leander Paes

>>> 
```

Note empty line due to '\n'

```
>>> n.close()
```

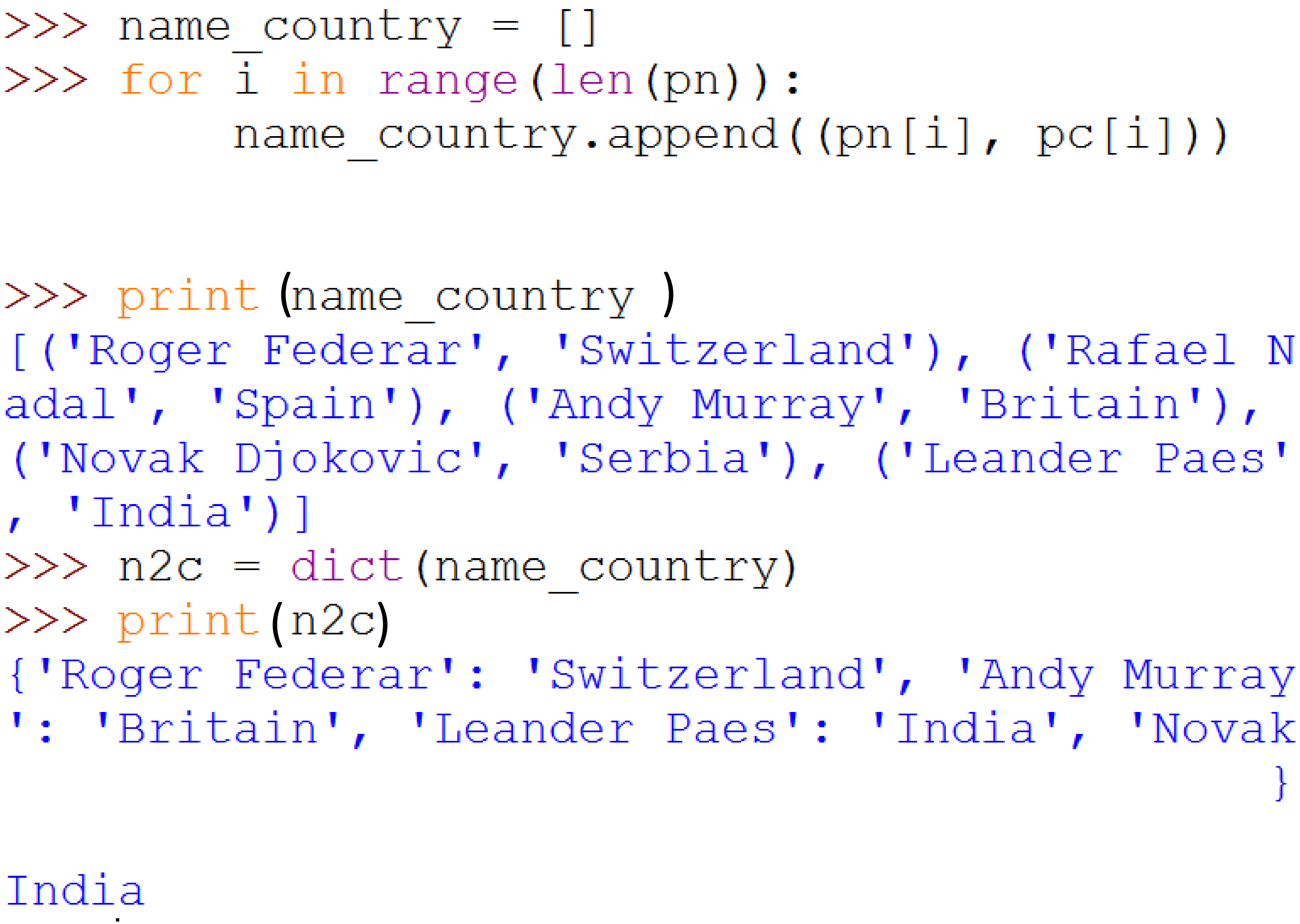
File I/O: Examples



```
>>> n = open('tennis_players', 'r')
>>> c = open('tennis_countries', 'r')
>>> pn, pc = [], []
>>> for l in n:
    pn.append(l[:-1]) # ignore '\n'

>>> n.close()

>>> for l in c:
    pc.append(l[:-1])
>>> c.close()
>>> print(pn, '\n', pc)
['Roger Federar', 'Rafael Nadal', 'Andy Murra
y', 'Novak Djokovic', 'Leander Paes']
['Switzerland', 'Spain', 'Britain', 'Serbia',
'India ]
```

Note the use of for ... in for sequence

File I/O: Examples



```
>>> name_country = []
>>> for i in range(len(pn)):
        name_country.append((pn[i], pc[i]))


>>> print (name_country )
[('Roger Federar', 'Switzerland'), ('Rafael N
adal', 'Spain'), ('Andy Murray', 'Britain'),
('Novak Djokovic', 'Serbia'), ('Leander Paes'
, 'India')]
>>> n2c = dict(name_country)
>>> print(n2c)
{'Roger Federar': 'Switzerland', 'Andy Murray
': 'Britain', 'Leander Paes': 'India', 'Novak
                                             }

India
```

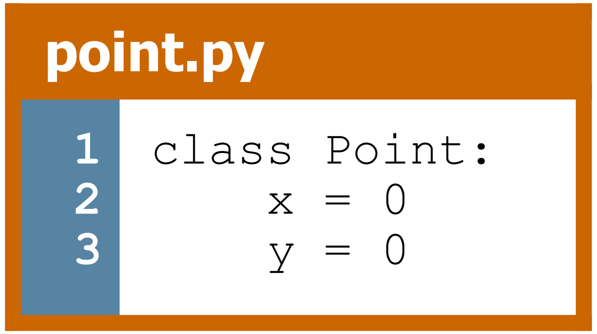
OOP, Defining a Class

- Python was built as a procedural language
  - OOP exists and works fine, but feels a bit more "tacked on“

- Declaring a class:

```
class name:
    statements
```

Fields



point.py

```
class Point:
    x = 0
    y = 0
```

name = value

- Example:
  class Point:
      x = 0
      y = 0
  # main
  p1 = Point()
  p1.x = 2
  p1.y = -5

- can be declared directly inside class (as shown here)
  or in constructors (more common)

- Python does not really have encapsulation or private fields
  - relies on caller to "be nice" and not mess with objects' contents

Using a Class

import **class**

- client programs must import the classes they use

point_main.py

```
from Point import *

# main
p1 = Point()
p1.x = 7
p1.y = -3
...

# Python objects are dynamic (can add fields any time!)
p1.name = "Tyler Durden"
```

Object Methods

def **name**(self**, parameter, ..., parameter**):

statements

- self *must* be the first parameter to any object method
    - represents the "implicit parameter" (this in Java)
- *must* access the object's fields through the self reference

```
class Point:
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
    ...
```

"Implicit" Parameter (self)

- Python: self, explicit

```
def translate(self, dx, dy):
    self.x += dx
    self.y += dy
```

    - Exercise: Write distance, set_location, and distance_from_origin methods.

Exercise Answer

point.py

```
from math import *

class Point:
    x = 0
    y = 0

    def set_location(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)
```

Calling Methods

- A client can call the methods of an object in two ways:
  - (the value of self can be an implicit or explicit parameter)

1) **object**.**method**(**parameters**) or

2) **Class**.**method**(**object**, **parameters**)

- Example:

```
p = Point(3, -4)
p.translate(1, 5)
Point.translate(p, 1, 5)
```

Constructors

```
def __init__(self, parameter, ..., parameter):
    statements
```

- a constructor is a special method with the name __init__
- Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    ...
```

  - How would we make it possible to construct a Point() with no parameters to get (0, 0)?

toString and __str__

```
def __str__(self):
    return string
```

- equivalent to Java's toString (converts object to a string)
- invoked automatically when str or print is called

Exercise: Write a __str__ method for Point objects that returns strings like "(3, -14)"

```
def __str__(self):
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Complete Point Class

point.py

```python
from math import *

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)

    def translate(self, dx, dy):
        self.x += dx
        self.y += dy

    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Operator Overloading

- **operator overloading**: You can define functions so that Python's built-in operators can be used with your class.
  - See also: http://docs.python.org/ref/customization.html

Operator	Class Method
-	__neg__(self, other)
+	__pos__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)

Unary Operators

-	__neg__(self)
+	__pos__(self)

Operator	Class Method
==	__eq__(self, other)
!=	__ne__(self, other)
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)

Generating Exceptions

raise **ExceptionType**("**message**")

- useful when the client uses your object improperly
- types: ArithmeticError, AssertionError, IndexError, NameError, SyntaxError, TypeError, ValueError
- Example:

  ```
  class BankAccount:
      ...
      def deposit(self, amount):
          if amount < 0:
              raise ValueError("negative amount")
          ...
  ```

Inheritance

```
class name(superclass):
    statements
```

- Example:

```
class Point3D(Point):   # Point3D extends Point
    z = 0
    ...
```

- Python also supports *multiple inheritance*

```
      class name(superclass, ..., superclass):
statements
```

(if > 1 superclass has the same field/method, conflicts are resolved in left-to-right order)

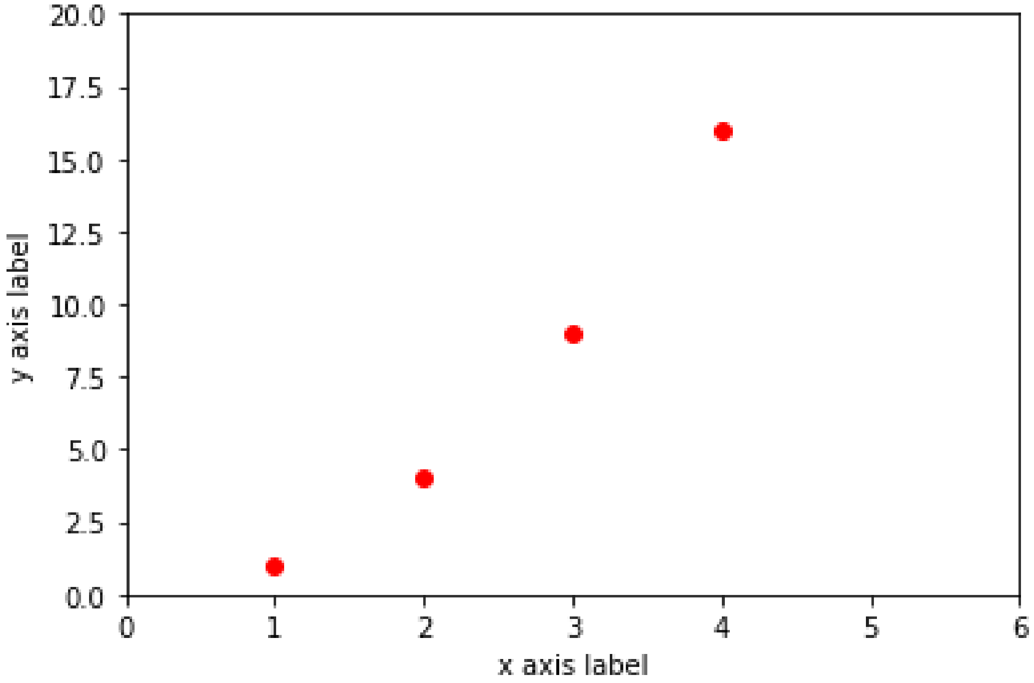
Calling Superclass Methods

- methods: **class.method**(**object**, **parameters**)
- constructors: **class**.__init__(**parameters**)

```
class Point3D(Point):
    z = 0
    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z
    def translate(self, dx, dy, dz):
        Point.translate(self, dx, dy)
        self.z += dz
```

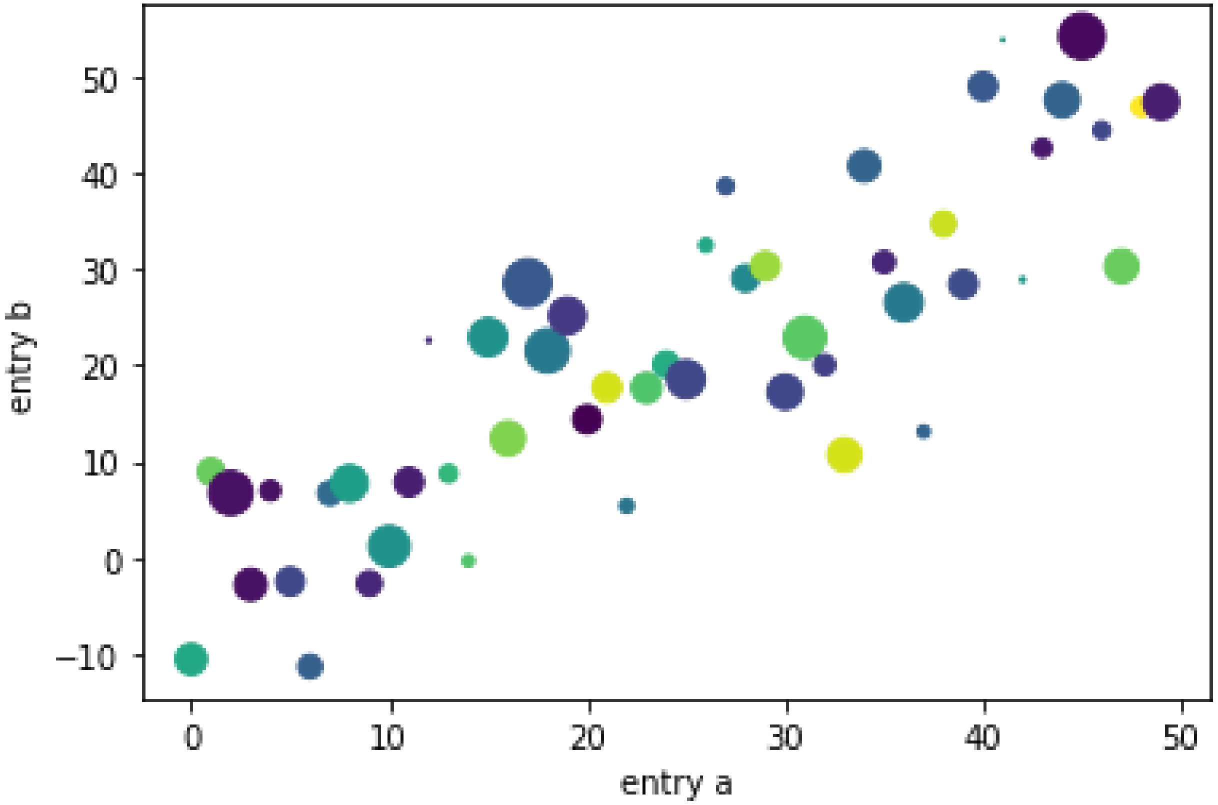
Python module

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis([0, 6, 0, 20])
plt.ylabel('y axis label')
plt.xlabel('x axis label')
plt.show()
```



Python module

```
import numpy as np
import matplotlib.pyplot as plt

data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```


entry b
entry a