A Quick Tour of Python

Amey Karkare
Dept. of CSE
IIT Kanpur

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
```

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] # Repetition
[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

Mutable and Immutable Types

- Tuples and List types look very similar
- However, there is one major difference: Lists are **mutable**
  - Contents of a list can be modified
- Tuples and Strings are **immutable**
  - Contents can not be modified

Summary of Sequences

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)

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

Summary of Sequences

- For details and many useful functions, refer to:

https://docs.python.org/3.2/tutorial/datastructures.html

Programming with Python

Sets and Dictionaries

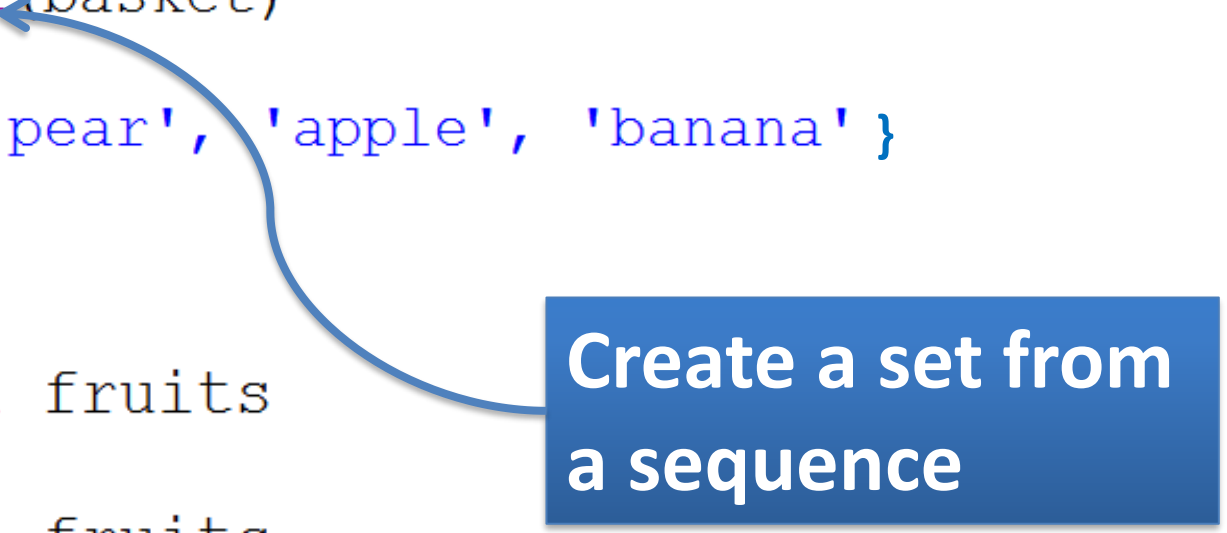
Sets

- 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
```



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'])
```

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 Kotte' # 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'}
```

Programming with Python

File I/O

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>

>>> n = open('tennis_players', 'r')
>>> c = open('tennis_countries', 'r')
>>> n
<open file 'tennis_players', mode 'r' at 0x03
1A4910>
>>> c
<open file 'tennis_countries', mode 'r' at 0x
031A4A70>
```

```
>>> 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

>>>
>>> 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()
```

Note the use of for ... in for sequence

```
>>> print(pn, '\n', pc)
['Roger Federar', 'Rafael Nadal', 'Andy Murra
y', 'Novak Djokovic', 'Leander Paes']
['Switzerland', 'Spain', 'Britain', 'Serbia',
'India']
```

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
Djokovic': 'Serbia', 'Rafael Nadal': 'Spain'}
>>> print(n2c['Leander Paes'])
India
```

Programming using Python

Modules and Packages

Amey Karkare

Dept. of CSE

IIT Kanpur

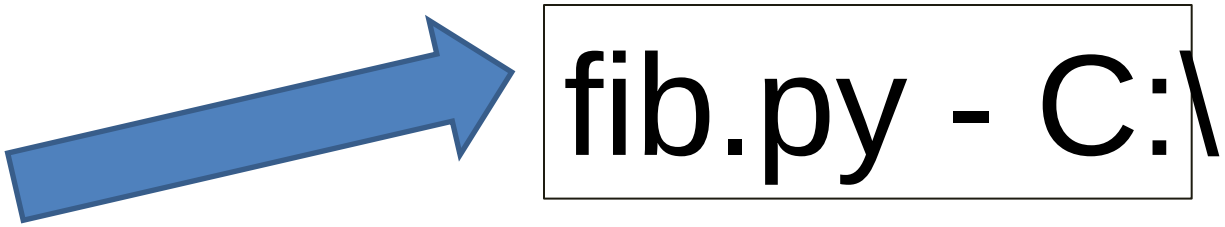
Modules

- As program gets longer, need to organize them for easier access and easier maintenance.
- Reuse same functions across programs without copying its definition into each program.
- Python allows putting definitions in a file
  - use them in a script or in an interactive instance of the interpreter
- Such a file is called a *module*
  - definitions from a module can be *imported* into other modules or into the *main* module

Modules

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Within a module, the module's name is available in the global variable __name__.

Modules Example



fib.py - C:\Users\karkare\Google Drive\IITK\Courses\2016Python\Programs\fib.py (2.7.12)

File Edit Format Run Options Window Help

```
# Module for fibonacci numbers

def fib_rec(n):
    '''recursive fibonacci'''
    if (n <= 1):
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

Modules Example

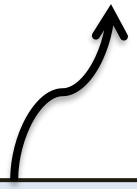
```
def fib_rec(n):
    '''recursive fibonacci'''
    if (n <= 1):
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)

def fib_iter(n):
    '''iterative fibonacci'''
    cur, nxt = 0, 1
    for k in range(n):
        cur, nxt = nxt, cur+nxt
    return cur

def fib_upto(n):
    '''given n, return list of fibonacci
    numbers <= n'''
    cur, nxt = 0, 1
    lst = []
    while (cur < n):
        lst.append(cur)
        cur, nxt = nxt, cur+nxt
    return lst
```

```
>>> import fib
>>> fib.fib_upto(5)
[0, 1, 1, 2, 3]
>>> fib.fib_rec(10)
55
>>> fib.fib_iter(20)
6765

>>> fib.__name__
'fib'
```



Within a module, the module's name is available as the value of the global variable __name__.

Importing Specific Functions

- To import specific functions from a module

```
>>> from fib import fib_upto
>>> fib_upto(6)
[0, 1, 1, 2, 3, 5]
>>> fib_iter(1)

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    fib_iter(1)
NameError: name 'fib_iter' is not defined
```

- This brings only the imported functions in the current symbol table
  - No need of modulename. (absence of `fib.` in the example)

Importing ALL Functions

- To import *all* functions from a module, in the current symbol table

```
>>> from fib import *
>>> fib_upto(6)
[0, 1, 1, 2, 3, 5]
>>> fib_iter(8)
21
```

- This imports all names except those beginning with an underscore (_).

`__main__` in Modules

- When you run a module on the command line with

  ```
  python fib.py <arguments>
  ```

  the code in the module will be executed, just as if you imported it, but with the `__name__` set to "__main__".

- By adding this code at the end of your module

  ```
  if __name__ == "__main__":
      ... # Some code here
  ```

  you can make the file usable as a script as well as an importable module

`__main__` in Modules

```
if __name__ == "__main__":
    import sys
    print (fib_iter(int(sys.argv[1])))
```

- This code parses the command line only if the module is executed as the “main” file:

```
$ python fib.py 10
55
```

- If the module is imported, the code is not run:

```
>>> import fib
>>>
```

Package

- A Python package is a collection of Python modules.
- Another level of *organization.*
- *Packages* are a way of structuring Python's module namespace by using *dotted module names*.
  - The module name A.B designates a submodule named B in a package named A.
  - The use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

A **sound** Package

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

https://docs.python.org/3/tutorial/modules.html

A **sound** Package

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

What are these files with funny names?

https://docs.python.org/3/tutorial/modules.html

`__init.py__`

- The `__init__.py` files are required to make Python treat directories containing the file as packages.
- This prevents directories with a common name, such as `string`, unintentionally hiding valid modules that occur later on the module search path.
- `__init__.py` can just be an empty file
- It can also execute initialization code for the package

Importing Modules from Packages

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

https://docs.python.org/3/tutorial/modules.ht

Importing Modules from Packages

`import sound.effects.echo`

- Loads the submodule `sound.effects.echo`
- It must be referenced with its full name:

```
sound.effects.echo.echofilter(
    input, output,
    delay=0.7, atten=4
)
```

Importing Modules from Packages

`from sound.effects import echo`

- This also loads the submodule `echo`
- Makes it available without package prefix
- It can be used as:

```
echo.echofilter(
    input, output,
    delay=0.7, atten=4
)
```

Importing Modules from Packages

```
from sound.effects.echo import echofilter
```

- This loads the submodule `echo`, but this makes its function `echofilter()` directly available.

```
echofilter(input, output,
           delay=0.7, atten=4)
```

Popular Packages

- pandas, numpy, scipy, matplotlib, …
- Provide a lot of useful functions