Basic Python

FOSSEE

Department of Aerospace Engineering
IIT Bombay

Outline

1. The Language
2. The Interpreter
3. Basic Datatypes and Operators
4. Strings
5. Conditionals
6. Loops
7. Lists
8. I/O
9. Files
10. Functions
11. Tuples
12. Dictionaries
13. Sets

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Python!

- Programming Language
- Powerful, High-level, Interpreted, Multi-Platform
- Elegant and highly readable syntax
- Efficient high-level data structures

- Easy to learn
- Allows to concentrate on the problem instead of the language
- Increased Productivity

- Guido van Rossum – BDFL
- Conceived in December 1989
- Named after “Monty Python’s Flying Circus”, a 70s comedy

Why Python?

- Extremely readable; Forces programmers to write readable code
- Interactive; Offers a very fast edit-test-debug cycle
- Doesn't get in your way; High-level data structures let you focus on the problem
- Handles memory management
- Batteries included; Huge standard library for wide range of tasks
- Object-oriented
- C, C++ and FORTRAN interfacing allows use of legacy code
- Your time is more valuable than machine time!

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Python interpreter

- Let's get our hands dirty!
- Start Python from your shell

```
$ python
```

```
Python 2.7.1 (r271:86832, Feb 21 2011, 01:28:26)
[GCC 4.5.2 20110127 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license"
>>>
```

- First line shows Python version (2.7.1)
- >>> the interpreter's prompt
- The interpreter is ready and waiting for your command!

Hello World!

- Type`print 'Hello World'` and hitting enter

```
>>> print 'Hello, World!'
Hello, World!
```

- The interpreter prints out the words *Hello World*
- Hit `Ctrl-D` to exit the interpreter
- We shall look at IPython, an enhanced interpreter

Versions

Before moving on . . .

- Currently has two stable branches or versions, 2.x and 3.x
- 3.x is not backward compatible
- 3.x is deemed to be the future of Python
- But, we shall stick to 2.x for this course
- The ecosystem around Python 2.x hasn't yet moved to 3.x

Invoking IPython

- An enhanced Python interpreter
- Tab-completion, Easier access to help, Better history

```
$ ipython
```

If `ipython is not installed`, you need to install it!

- The prompt is `In [1]:` instead of `>>>`
- `In` stands for input, 1 indicates the command number
- Try `Hello World`

```
In []: print 'Hello, World!'
Out[]: Hello, World!
```

the numbers have been omitted to avoid confusion

- Hit `Ctrl-D` to exit `ipython`; Say `y` when prompted.

Getting comfortable

- Let's try some simple math to get comfortable

```
In []: 1 + 2
In []: 5 - 3
In []: 7 - 4
In []: 6 * 5
```

- We get back the expected output
- Output is displayed with an `Out[]`

History & Arrow Keys

- Change the `print 1+2`
- Use <UP-Arrow> to go back to `1+2` command
- Use <LEFT-Arrow> to get to start of line; type `print`
- Hit <RETURN>

```
In []: print 1 + 2
```

- Now, change the previous command to `print 10*2`

Tab-Completion

- We want to use `round` function
- Type `ro`, and hit <TAB>

```
In []: ro<TAB>
```

- Type `r`, and hit <TAB>
- All possibilities are listed out, when ambiguous

? for Help

- To get help for `abs` function

```
In []: abs?
In []: abs(19)
In []: abs(-10.5)
```

- Look at documentation for `round`
- Optional arguments are denoted with square brackets `[]`

```
In []: round(2.484)
In []: round(2.484, 1)
In []: round(2.484, 2)
```

? for Help

- To get help for `abs` function

```
In []: abs?
In []: abs(19)
In []: abs(-10.5)
```

- Look at documentation for `round`
- Optional arguments are denoted with square brackets `[]`

```
In []: round(2.484)
In []: round(2.484, 1)
In []: round(2.484, 2)
```

Interrupting

```
In []: round(2.484
   ...:
```

- The ... prompt is the continuation prompt
- It comes up, since we haven't completed previous command
- Either complete by typing the missing `)`
- OR hit `Ctrl-C` to interrupt the command

```
In []: round(2.484
   ...: ^C
```

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Basic Datatypes

- Numbers
    - int
    - float
    - complex
- Boolean
- Sequence
    - Strings
    - Lists
    - Tuples

int

```
In []: a = 13
In []: a
```

- a is a variable of the int type
- Use the type command to verify

```
In []: type(a)
```

- Integers can be arbitrarily long

```
In []: b = 99999999999999999999999999999
In []: b
```

float

```
In []: p = 3.141592
In []: p
```

- Decimal numbers are represented using the `float` type
- Notice the loss of precision
- Floats have a fixed precision

complex

```
In []: c = 3+4j
```

- A complex number with real part 3, imaginary part 4

```
In []: c.real
In []: c.imag
In []: abs(c)
```

- It's a combination of two floats
- `abs` gives the absolute value

Operations on numbers

```
In []: 23 + 74
In []: 23 - 56
In []: 45 * 76

In []: 8 / 3
In []: 8.0 / 3
In []: float(8) / 3
```

- The first division is an integer division
- To avoid integer division, at least one number should be float
- `float` function is changing int to float

```
In []: 87 % 6
In []: 7 ** 8
```

- `%` is used for modulo operation
- `**` is used for exponentiation

Variables & assignment

- All the operations could be done on variables

```
In []: a = 23
In []: b = 74
In []: a * b

In []: c = 8
In []: d = 8.0
In []: f = c / 3
```

- Last two commands show assignment

```
In []: c = c / 3
```

An operation like the one above, may equivalently be written as

```
In []: c /= 3
```

Booleans & Operations

- All the operations could be done on variables

```
In []: t = True
In []: t
In []: f = not t
In []: f
In []: f or t
In []: f and t
```

- Multiple operation in a single command
- We use parenthesis for explicitly stating what we mean
- No discussion of operator precedence

```
In []: (f and t) or t
In []: f and (t or t)
```

Sequences

- Hold a bunch of elements in a sequence
- Elements are accessed based on position in the sequence
- The sequence data-types
  - str
  - list
  - tuple

Strings, Lists & Tuples

- Anything withing quotes is a string

```
In []: greet_str = "hello"
```

- Items enclosed in `[ ]` and separated by `,`s constitute a list

```
In []: num_list = [1, 2, 3, 4, 5, 6, 7, 8]
```

- Items of a tuple are enclosed by `( )` instead of `[ ]`

```
In []: num_tuple = (1, 2, 3, 4, 5, 6, 7, 8)
```

Operations on Sequences

- Accessing elements

```
In []: num_list[2]
In []: num_tuple[2]
In []: greet_str[2]
```

- Add two sequences of same type

```
In []: num_list + [3, 4, 5, 6]
In []: greet_str + " world!"
```

- Get the length of a sequence

```
In []: len(num_list)
In []: len(greet_str)
```

Operations on Sequences . . .

- Check for container-ship of elements

```
In []: 3 in num_list
In []: 'h' in greet_str
In []: 'w' in greet_str
In []: 2 in num_tuple
```

- Finding maximum and minimum

```
In []: max(num_list)
In []: min(greet_str)
```

- Slice a sequence

```
In []: num_list[1:5]
```

- Stride over a sequence

```
In []: num_list[1:8:2]
```

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

What are Strings?

- Anything quoted is a string
- Single quotes, double quotes or triple single/double quotes
- Any length — single character, null string, . . .

```
In[]: 'This is a string'
In[]: "This is a string too"
In[]: '''This is a string as well'''
In[]: """This is also a string"""
In[]: ''   # empty string
```

Why so many?

- Reduce the need for escaping

```
In[]: "Python's strings are powerful!"
In[]: 'He said, "I love Python!"'
```

- Triple quoted strings can be multi-line
- Used for doc-strings

Assignment & Operations

```
In[]: a = 'Hello'
In[]: b = 'World'
In[]: c = a + ', ' + b + '!'
```

- Strings can be multiplied with numbers

```
In[]: a = 'Hello'
In[]: a * 5
```

Accessing Elements

```
In[]: print a[0], a[4], a[-1], a[-4]
```

- Can we change the elements?

```
In[]: a[0] = 'H'
```

- Strings are immutable!

Problem - Day of the Week?

- Strings have methods to manipulate them

Problem

Given a list, `week`, containing names of the days of the week and a string `s`, check if the string is a day of the week. We should be able to check for any of the forms like, *sat, Sat, SAT*

- Get the first 3 characters of the string
- Convert it all to lower case
- Check for existence in the list, `week`

Slicing

```
In[]: q = "Hello World"
In[]: q[0:3]
In[]: q[:3]
In[]: q[3:]
In[]: q[:]
In[]: q[-1:1]
In[]: q[1:-1]
```

- One or both of the limits, is optional

Striding

```
In[]: q[0:5:1]
In[]: q[0:5:2]
In[]: q[0:5:3]
In[]: q[0::2]
In[]: q[2::2]
In[]: q[::2]
In[]: q[5:0:-1]
In[]: q[::-1]
```

String Methods

```
In[]: s.lower()
In[]: s.upper()
 s.<TAB>
```

- Strings are immutable!
- A new string is being returned

Solution - Day of the Week?

```
In[]: s.lower()[:3] in week
```

OR

```
In[]: s[:3].lower() in week
```

`join` a list of strings

- Given a list of strings
- We wish to join them into a single string
- Possibly, each string separated by a common token

```
In[]: email_list = ["info@fossee.in",
                    "enquiries@fossee.in",
                    "help@fossee.in"]

In[]: '; '.join(email_list)
In[]: ', '.join(email_list)
```

Outline

1. The Language
2. The Interpreter
3. Basic Datatypes and Operators
4. Strings
5. **Conditionals**
6. Loops
7. Lists
8. I/O
9. Files
10. Functions
11. Tuples
12. Dictionaries
13. Sets

if-else block

```
In[]: a = 5
In[]: if a % 2 == 0:
....:     print "Even"
....: else:
....:     print "Odd"
```

- A code block – `:` and indentation
- Exactly one block gets executed in the `if-else`

if-elif-else

```
In[]: if a > 0:
....:     print "positive"
....: elif a < 0:
....:     print "negative"
....: else:
....:     print "zero"
```

- Only one block gets executed, depending on `a`

else is optional

```
In[]: if user == 'admin':
....:     admin_Operations()
....: elif user == 'moderator':
....:     moderator_operations()
....: elif user == 'client':
....:     customer_operations()
```

- Note that there is no `else` block

Ternary operator

- `score_str` is either `'AA'` or a string of one of the numbers in the range 0 to 100.
- We wish to convert the string to a number using `int`
- Convert it to 0, when it is `'AA'`
- `if-else` construct or the ternary operator

```
In[]: if score_str != 'AA':
 ....:     score = int(score_str)
 ....: else:
 ....:     score = 0
```

```
In[]: ss = score_str
In[]: score = int(ss) if ss != 'AA' else 0
```

`pass`

- `pass` is a syntactic filler
- When a certain block has no statements, a `pass` is thrown in
- Mostly, when you want to get back to that part, later.

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

while

- Print squares of all odd numbers less than 10 using `while`

```
In[]: i = 1

In[]: while i<10:
....:     print i*i
....:     i += 2
```

- The loops runs as long as the condition is `True`

for

- Print squares of all odd numbers less than 10 using `for`

```
In[]: for n in [1, 2, 3]:
....:     print n
```

- `for` iterates over each element of a sequence

```
In[]: for n in [1, 3, 5, 7, 9]:
....:     print n*n

In[]: for n in range(1, 10, 2):
....:     print n*n
```

- range([start,] stop[, step])
- Returns a list; Stop value is not included.

break

- breaks out of the innermost loop.
- Squares of odd numbers below 10 using `while` & `break`

```
In[]: i = 1

In[]: while True:
....:     print i*i
....:     i += 2
....:     if i>10:
....:         break
```

continue

- Skips execution of rest of the loop on current iteration
- Jumps to the end of this iteration
- Squares of all odd numbers below 10, not multiples of 3

```
In[]: for n in range(1, 10, 2):
....:     if n%3 == 0:
....:         continue
....:     print n*n
```

Problem - Day of the Week?

- Strings have methods to manipulate them

Problem

Given a list, `week`, containing names of the days of the week and a string `s`, check if the string is a day of the week. We should be able to check for any of the forms like, *sat, saturday, Sat, Saturday, SAT, SATURDAY*

- Get the first 3 characters of the string
- Convert it all to lower case
- Check for existence in the list, `week`

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Creating Lists

```
In[]: empty = []

In[]: p = ['spam', 'eggs', 100, 1.234]
In[]: q = [[4, 2, 3, 4], 'and', 1, 2, 3, 4]
```

- Lists can be empty, with no elements in them
- Lists can be heterogeneous – every element of different kind

Accessing Elements

```
In[]: print p[0], p[1], p[3]

In[]: print p[-1], p[-2], p[-4]
In[]: print p[10]
```

- Indexing starts from 0
- Indexes can be negative
- Indexes should be in the valid range

Accessing Elements & length

```
In[]: print p[0], p[1], p[3]

In[]: print p[-1], p[-2], p[-4]
In[]: print len(p)
In[]: print p[10]
```

- Indexing starts from 0
- Indexes can be negative
- Indexes should be within the `range(0, len(p))`

Adding & Removing Elements

- The append method adds elements to the end of the list

```
In[]: p.append('onemore')
In[]: p
In[]: p.append([1, 6])
In[]: p
```

- Elements can be removed based on their index OR
- based on the value of the element

```
In[]: del p[1]
In[]: p.remove(100)
```

- When removing by value, first element is removed

Concatenating lists

```
In[]: a = [1, 2, 3, 4]
In[]: b = [4, 5, 6, 7]
In[]: a + b
In[]: print a+b, a, b
```

- A new list is returned; None of the original lists change

```
In[]: c = a + b
In[]: c
```

Slicing & Striding

```
In[]: primes = [2, 3, 5, 7, 11, 13, 17, 19, 23,
In[]: primes[4:8]
In[]: primes[:4]

In[]: num = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
In[]: num[1:10:2]
In[]: num[:10]
In[]: num[10:]
In[]: num[::2]
In[]: num[::-1]
```

Sorting

```
In[]: a = [5, 1, 6, 7, 7, 10]
In[]: a.sort()
In[]: a
```

- `sort` method sorts the list in-place
- Use `sorted` if you require a new list

```
In[]: a = [5, 1, 6, 7, 7, 10]
In[]: sorted(a)
In[]: a
```

Reversing

```
In[]: a = [5, 1, 6, 7, 7, 10]
In[]: a.reverse()
In[]: a
```

- `reverse` method reverses the list in-place
- Use `[::-1]` if you require a new list

```
In[]: a = [5, 1, 6, 7, 7, 10]
In[]: a[::-1]
In[]: a
```

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Printing

```
In[]: a = "This is a string"
In[]: a
In[]: print a
```

- Both `a`, and `print a` are showing the value
- What is the difference?
- Typing `a` shows the value; `print a` prints it
- Typing `a` shows the value only in interpreter
- In a script, it has no effect.

```
In[]: b = "A line \n New line"
In[]: b
In[]: print b
```

String formatting

```
In[]: x = 1.5
In[]: y = 2
In[]: z = "zed"
In[]: print "x is %2.1f y is %d z is %s" %(x, y, z)
```

`print x` & `print x,`

- Open an editor
- Type the following code
- Save as `print_example.py`

```
In[]: print "Hello"
In[]: print "World"

In[]: print "Hello",
In[]: print "World"
```

- Run the script using `% run print_example.py`
- `print x` adds a newline whereas `print x,` adds a space

raw_input

```
In[]: ip = raw_input()
```

- The cursor is blinking; waiting for input
- Type `an input` and hit <ENTER>

```
In[]: print ip
```

raw_input ...

```
In[]: c = raw_input()
5.6
In[]: c
In[]: type(c)
```

- `raw_input` always takes a string

```
In[]: name = raw_input("Enter your name: ")
Enter your name:  George
```

- `raw_input` can display a prompt string for the user

Outline

1. The Language
2. The Interpreter
3. Basic Datatypes and Operators
4. Strings
5. Conditionals
6. Loops
7. Lists
8. I/O
9. Files
10. Functions
11. Tuples
12. Dictionaries
13. Sets

Opening files

```
pwd # present working directory
cd /home/fossee # go to location of the file
```

The file is in our present working directory

```
In[]: f = open('pendulum.txt')
In[]: f
```

- `f` is a file object
- Shows the mode in which the file is open (read mode)

Reading the whole file

```
In[]: pend = f.read()
In[]: print pend
```

- We have read the whole file into the variable `pend`

```
In[]: type(pend)
In[]: pend_list = pend.splitlines()
In[]: pend_list
```

- `pend` is a string variable
- We can split it at the newline characters into a list of strings
- Close the file, when done; Also, if you want to read again

```
In[]: f.close()
In[]: f
```

Reading line-by-line

```
In[]: for line in open('pendulum.txt'):
....:     print line
```

- The file object is an “iterable”
- We iterate over it and print each line
- Instead of printing, collect lines in a list

```
In[]: line_list = [ ]
In[]: for line in open('pendulum.txt'):
....:     line_list.append(line)
```

File parsing – Problem

```
A;010002;ANAND R;058;037;42;35;40;212;P;;
```

- File with records like the one above is given
- Each record has fields separated by `;`
- region code; roll number; name;
- marks — 1st L; 2nd L; math; science; social; total
- pass/fail indicated by P/F; W if withheld and else empty

- We wish to calculate mean of math marks in region B

Tokenization

```
In[]: line = "parse this           string"
In[]: line.split()
```

- Original string is split on white-space (if no argument)
- Returns a list of strings
- It can be given an argument to split on that argrument

```
In[]: record = "A;015163;JOSEPH RAJ S;083;042;
In[]: record.split(';')
```

Tokenization . . .

- Since we split on commas, fields may have extra spaces at ends
- We can strip out the spaces at the ends

```
In[]: word = "      B     "
In[]: word.strip()
```

- `strip` is returning a new string

str to float

- After tokenizing, the marks we have are strings
- We need numbers to perform math operations

```
In[]: mark_str = "1.25"
In[]: mark = int(mark_str)
In[]: type(mark_str)
In[]: type(mark)
```

- strip is returning a new string

File parsing – Solution

```
In[]: math_B = [] # empty list to store marks
In[]: for line in open("sslc1.txt"):
....:     fields = line.split(";")

....:     reg_code = fields[0]
....:     reg_code_clean = reg_code.strip()

....:     math_mark_str = fields[5]
....:     math_mark = float(math_mark_str)

....:     if reg_code == "B":
....:         math_B.append(math_mark)

In[]: math_B_mean = sum(math_B) / len(math_B)
In[]: math_B_mean
```

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Abstracting

- Reduce duplication of code
- Fewer lines of code and hence lesser scope for bugs
- Re-usability of code, that's already been written
- Use functions written by others, without exactly knowing how they do, what they are doing
- Enter Functions!

Defining functions

- Consider the function `f(x) = x^2`
- Let's write a Python function, equivalent to this

```
In[]: def f(x):
....:     return x*x
....:

In[]: f(1)
In[]: f(2)
```

- `def` is a keyword
- `f` is the name of the function
- `x` the parameter of the function
- `return` is a keyword; specifies what should be returned

Defining functions ...

```
In[]: def greet():
....:     print "Hello World!"
....:

In[]: greet()
```

- `greet` is a function that takes no arguments
- Also, it is not returning anything explicitly
- But implicitly, Python returns `None`

```
In[]: def avg(a, b):
....:     return (a + b)/2
....:

In[]: avg(12, 10)
```

Doc-strings

- It's highly recommended that all functions have documentation
- We write a doc-string along with the function definition

```
In[]: def avg(a, b):
      """ avg takes two numbers as input
      and returns their average"""

....:     return (a + b)/2
....:

In[]: avg?
In[]: greet?
```

Returning multiple values

- Return area and perimeter of circle, given radius
- Function needs to return two values

```
In[]: def circle(r):
      """returns area and perimeter of a
      circle given, the radius r"""

....:     pi = 3.14
....:     area = pi * r * r
....:     perimeter = 2 * pi * r
....:     return area, perimeter
....:

In[]: circle(4)
In[]: a, p = circle(6)
In[]: print a
```

What? – 1

```
In[]: def what( n ):
....:     if n < 0: n = -n
....:     while n > 0:
....:         if n % 2 == 1:
....:             return False
....:         n /= 10
....:     return True
....:
```

What? – 2

```
In[]: def what( n ):
....:     i = 1
....:     while i * i < n:
....:         i += 1
....:     return i * i == n, i
....:
```

Default arguments

```
In[]: round(2.484)
In[]: round(2.484, 2)

In[]: s.split() # split on spaces
In[]: s.split(';') # split on ';'

In[]: range(10) # returns numbers from 0 to 9
In[]: range(1, 10) # returns numbers from 1 to
In[]: range(1, 10, 2) # returns odd numbers fro
```

Default arguments . . .

```
In[]: def welcome(greet, name="World"):
....:     print greet, name
....:
```

```
In[]: welcome("Hi", "Guido")
In[]: welcome("Hello")
```

- Arguments with default values, should be placed at the end
- The following definition is WRONG

```
In[]: def welcome(name="World", greet):
....:     print greet, name
....:
```

Keyword Arguments

```
In[]: def welcome(greet, name="World"):
....:     print greet, name
....:

In[]: welcome("Hello", "James")

In[]: welcome("Hi", name="Guido")

In[]: welcome(name="Guido", greet="Hey")

In[]: welcome(name="Guido", "Hey")
```

Built-in functions

- Variety of built-in functions are available
- `abs, any, all, len, max, min`
- `pow, range, sum, type`
- Refer here: `http://docs.python.org/library/functions.html`

Arguments are local

```
In[]: def change(q):
....:     q = 10
....:     print q
....:

In[]: change(1)
In[]: print q
```

Variables inside function are local

```
In[]: n = 5
In[]: def change():
....:     n = 10
....:     print n
....:
In[]: change()
In[]: print n
```

global

- Use the `global` statement to assign to global variables

```
In[]: def change():
....:     global n
....:     n = 10
....:     print n
....:
In[]: change()
In[]: print n
```

Mutable variables

- Behavior is different when assigning to a list element/slice
- Python looks up for the name, from innermost scope outwards, until the name is found

```
In[]: name = ['Mr.', 'Steve', 'Gosling']
In[]: def change_name():
....:     name[0] = 'Dr.'
....:
In[]: change_name()
In[]: print name
```

Passing Arguments . . .

```
In[]: n = 5
In[]: def change(n):
....:     n = 10
....:     print "n = %s inside change " %n
....:
In[]: change(n)
In[]: print n

In[]: name = ['Mr.', 'Steve', 'Gosling']
In[]: def change_name(n):
....:     n[0] = 'Dr.'
....:     print "n = %s inside change_name" %n
....:
In[]: change_name(name)
In[]: print name
```

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Tuples – Initialization

```
In[]: t = (1, 2.5, "hello", -4, "world", 1.24,
In[]: t
```

- It is not always necessary to use parenthesis

```
In[]: a = 1, 2, 3
In[]: b = 1,
```

Indexing

```
In[]: t[3]
In[]: t[1:5:2]
In[]: t[2] = "Hello"
```

- Tuples are immutable!

Swapping values

```
In[]: a = 5
In[]: b = 7

In[]: temp = a
In[]: a = b
In[]: b = temp
```

- Here's the Pythonic way of doing it

```
In[]: a, b = b, a
```

- The variables can be of different data-types

```
In[]: a = 2.5
In[]: b = "hello"
In[]: a, b = b, a
```

Tuple packing & unpacking

```
In[]: 5,
```

```
In[]: 5, "hello", 2.5
```

- Tuple packing and unpacking, when swapping

```
In[]: a, b = b, a
```

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Creating Dictionaries

```
In[]: mt_dict = {}

In[]: extensions = {'jpg'  : 'JPEG Image',
                'py'   : 'Python script',
                'html' : 'Html document',
                'pdf'  : 'Portable Document Form

In[]: extensions
```

- Key-Value pairs
- No ordering of keys!

Accessing Elements

```
In[]: print extensions['jpg']
```

- Values can be accessed using keys

```
In[]: print extensions['zip']
```

- Values of non-existent keys cannot, obviously, be accessed

Adding & Removing Elements

- Adding a new key-value pair

```
In[]: extensions['cpp'] = 'C++ code'
In[]: extensions
```

- Deleting a key-value pair

```
In[]: del extension['pdf']
In[]: extensions
```

- Assigning to existing key, modifies the value

```
In[]: extensions['cpp'] = 'C++ source code'
In[]: extensions
```

Containership

```
In[]: 'py' in extensions
In[]: 'odt' in extensions
```

- Allow checking for container-ship of keys; NOT values
- Use the `in` keyword to check for container-ship

Lists of Keys and Values

```
In[]: extensions.keys()
In[]: extensions.values()
```

- Note that the order of the keys and values match
- That can be relied upon and used

```
In[]: for each in extensions.keys():
....:     print each, "-->", extensions[each]
....:
```

Outline

1 The Language
2 The Interpreter
3 Basic Datatypes and Operators
4 Strings
5 Conditionals
6 Loops
7 Lists
8 I/O
9 Files
10 Functions
11 Tuples
12 Dictionaries
13 Sets

Creating Sets

```
In[]: a_list = [1, 2, 1, 4, 5, 6, 2]
In[]: a = set(a_list)
In[]: a
```

- Conceptually identical to the sets in mathematics
- Duplicate elements not allowed
- No ordering of elements exists

Operations on Sets

```
In[]: f10 = set([1, 2, 3, 5, 8])
In[]: p10 = set([2, 3, 5, 7])
```

- Mathematical operations performed on sets, can be performed
- Union

```
In[]: f10 | p10
```

- Intersection

```
In[]: f10 & p10
```

- Difference

```
In[]: f10 - p10
```

- Symmetric Difference

```
In[]: f10 ^ p10
```

Sub-sets

- Proper Subset

```
In[]: b = set([1, 2])
In[]: b < f10
```

- Subsets

```
In[]: f10 <= f10
```

Elements of sets

- Containership

```
In[]: 1 in f10
In[]: 4 in f10
```

- Iterating over elements

```
In[]: for i in f10:
....:     print i,
....:
```

- Subsets

```
In[]: f10 <= f10
```

Sets – Example

Given a list of marks, `[20, 23, 22, 23, 20, 21, 23]` list all the duplicates

```
In[]: marks = [20, 23, 22, 23, 20, 21, 23]
In[]: marks_set = set(marks)
In[]: for mark in marks_set:
....:     marks.remove(mark)

  # left with only duplicates
In[]: duplicates = set(marks)
```