Python Introduction

Vaibhav Garg, IEEE-MAIT

What is Python?

- Python is a portable, interpreted, object-oriented scripting language created by Guido van Rossum.
- Its development started at the National Research Institute for Mathematics and Computer Science in the Netherlands , and continues under the ownership of the Python Software Foundation.
- Even though its name is commonly associated with the snake, it is actually named after the British comedy troupe, *Monty Python's Flying Circus.*

Why learn a scripting language?

- Easier to learn than more traditional compiled languages like C/C++
- Allows concepts to be prototyped faster
- Ideal for administrative tasks which can benefit from automation
- Typically cross platform

Why Python?

So, why use Python over other scripting languages?

- Well tested and widely used
- Great documentation
- Strong community support
- Widely Cross-platform
- Open sourced
- Can be freely used and distributed, even for commercial purposes.

How do I install Python?

- Simple… download it from www.python.org and install it.
- Most Linux distributions already come with it, but it may need updating.
- Current version, as of this presentation, is 2.4.2 Click here to download this version for Windows.

The print function

- The print function simply outputs a string value to our script's console.

```
print "Hello World!"
```

- It's very useful for communicating with users or outputting important information.

```
print "Game Over!"
```

- We get this function for free from Python.
- Later, we'll learn to write our own functions.

Adding Comments

- Comments allow us to add useful information to our scripts, which the Python interpreter will ignore completely.
- Each line for a comment must begin with the number sign '#'.

```
# This is a programming tradition…
print "Hello World!"
```

- Do yourself a favor and use them!

Quoting Strings

- Character strings are denoted using quotes.
- You can use double quotes…

```
print “Hello World!” # Fine.
```

- Or, you can use single quotes…

```
print ‘Hello World!’ # Also fine.
```

- It doesn’t matter as long as you don’t mix them

```
print “Hello World!’ # Syntax Error!
```

Triple Quotes

- You can even use triple quotes, which make quoting multi-line strings easier.

```
print """
This is line one.
This is line two.
This is line three.
Etc…
"""
```

Strings and Control Characters

- There are several control characters which allow us to modify or change the way character strings get printed by the print function.
- They're composed of a backslash followed by a character. Here are a few of the more important ones:

\n : New line
\t : Tabs
\\ : Backslash
\' : Single Quote
\" : Double Quote

Variables

- Like all programming languages, Python variables are similar to variables in Algebra.
- They act as place holders or symbolic representations for a value, which may or may not change over time.
- Here are six of the most important variable types in Python:

int	Plain Integers	(25)
long	Long Integers	(4294967296)
float	Floating-point Numbers	(3.14159265358979)
bool	Booleans	(True, False)
str	Strings	(“Hello World!”)
list	Sequenced List	([25, 50, 75, 100])

Type-less Variables

- Even though it supports variable types, Python is actually type-less, meaning you do not have to specify the variable's type to use it.
- You can use a variable as a character string one moment and then overwrite its value with a integer the next.
- This makes it easier to learn and use the language but being type-less opens the door to some hard to find bugs if you're not careful.
- If you're uncertain of a variable's type, use the type function to verify its type.

Rules for Naming Variables

- You can use letters, digits, and underscores when naming your variables.
- But, you cannot start with a digit.

```
var    = 0 # Fine.
var1   = 0 # Fine.
var_1= 0 # Fine.
_var   = 0 # Fine.
1var   = 0 # Syntax Error!
```

Rules for Naming Variables

- Also, you can't name your variables after any of Python's reserved keywords.

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

Numerical Precision

- Integers
  - Generally 32 signed bits of precision
  - [2,147,483,647 .. –2,147,483,648]
  - or basically (-2^{32} , 2^{32})
  - Example: 25
- Long Integers
  - Unlimited precision or size
  - Format: <number>L
  - Example: 4294967296L
- Floating-point
  - Platform dependant “double” precision
  - Example: 3.141592653589793

Type Conversion

- The special constructor functions int, long, float, complex, and bool can be used to produce numbers of a specific type.
- For example, if you have a variable that is being used as a float, but you want to use it like an integer do this:

```
myFloat = 25.12
myInt = 25
print myInt + int( myFloat )
```

- With out the explicit conversion, Python will automatically upgrade your addition to floating-point addition, which you may not want especially if your intention was to drop the decimal places.

Type Conversion

- There is also a special constructor function called str that converts numerical types into strings.

```
myInt = 25
myString = "The value of 'myInt' is "
print myString + str( myInt )      # Concatenation as well
```

- You will use this function a lot when debugging!
- Note how the addition operator was used to join the two strings together as one.

Arithmetic Operators

- Arithmetic Operators allow us to perform mathematical operations on two variables or values.
- Each operator returns the result of the specified operation.

+	Addition
-	Subtraction
*	Multiplication
/	Float Division
**	Exponent
abs	Absolute Value

Comparison Operators

- Comparison Operators return a True or False value for the two variables or values being compared.

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Is equal to
!=	Is not equal to

Boolean Operators

- Python also supports three Boolean Operators, and, or, and not, which allow us to make use of Boolean Logic in our scripts.
- Below are the Truth Tables for and, or, and not.

and

InA	InB	Out
0	0	0
0	1	1
1	0	1
1	1	1

or

InA	InB	Out
0	0	0
0	1	0
1	0	0
1	1	1

not

In	Out
0	1
1	0

Boolean Operators

Suppose that… var1 = 10.

- The and operator will return True if and only if both comparisons return True.

```
print var1 == 10 and var1 < 5      (Prints False)
```

- The or operator will return True if either of the comparisons return True.

```
print var1 == 20 or var1 > 5       (Prints True)
```

- And the not operator simply negates or inverts the comparison's result.

```
print not var1 == 10               (Prints False)
```

Special String Operators

- It may seem odd but Python even supports a few operators for strings.
- Two strings can be joined (concatenation) using the + operator.

```
print "Game " + "Over!"
```

Outputs to the console.

- A string can be repeated (repetition) by using the * operator.

```
print "Bang! " * 3
```

Outputs "! " to the console.

Flow Control

- Flow Control allows a program or script to alter its flow of execution based on some condition or test.
- The most important keywords for performing Flow Control in Python are if, else, elif, for, and while.

If Statement

- The most basic form of Flow Control is the if statement.

```
# If the player's health is less than or equal to 0 - kill him!
if health <= 0:
    print "You're dead!"
```

- Note how the action to be taken by the if statement is indented or tabbed over. This is not a style issue – it's required.
- Also, note how the if statement ends with a semi-colon.

If-else Statement

- The if-else statement allows us to pick one of two possible actions instead of a all-or-nothing choice.

```
health = 75

if health <= 0:
    print “You're dead!”
else:
    print “You're alive!”
```

- Again, note how the if and else keywords and their actions are indented. It’s very important to get this right!

If-elif-else Statement

- The if-elif-else statement allows us to pick one of several possible actions by chaining two or more if statements together.

```
health = 24

if health <= 0:
    print "You're dead!"
elif health < 25:
    print "You're alive - but badly wounded!"
else:
    print "You're alive!“
```

while Statement

- The while statement allows us to continuously repeat an action until some condition is satisfied.

```
numRocketsToFire = 3
rocketCount = 0

while rocketCount < numRocketsToFire:
    # Increase the rocket counter by one
     rocketCount = rocketCount + 1
    print "Firing rocket #" + str( rocketCount )
```

for Statement

- The for statement allows us to repeat an action based on the iteration of a Sequenced List.

```
weapons = [ “Pistol”, “Rifle”, “Grenade”, “Rocket Launcher” ]

print “-- Weapon Inventory --”

for x in weapons:
    print x
for x in range(100):
    print x

for x in range(0,100,2):
    print x
```

- The for statement will loop once for every item in the list.
- Note how we use the temporary variable ‘x’ to represent the current item being worked with.

break Keyword

- The break keyword can be used to escape from while and for loops early.

```
numbers = [100, 25, 125, 50, 150, 75, 175]

for x in numbers:
    print x
    # As soon as we find 50 - stop the search!
    if x == 50:
        print "Found It!"
        break;
```

- Instead of examining every list entry in “numbers”, The for loop above will be terminated as soon as the value 50 is found.

continue Keyword

- The continue keyword can be used to short-circuit or bypass parts of a while or for loop.

```
numbers = [100, 25, 125, 50, 150, 75, 175]

for x in numbers:
    # Skip all triple digit numbers
    if x >= 100:
        continue;
    print x
```

The for loop above only wants to print double digit numbers. It will simply continue on to the next iteration of the for loop if x is found to be a triple digit number.

Functions

- A function allows several Python statements to be grouped together so they can be called or executed repeatedly from somewhere else in the script.
- We use the def keyword to define a new function.

Defining functions

- Below, we define a new function called "printGameOver", which simply prints out, "Game Over!".

```
def printGameOver():
    print "Game Over!"
```

- Again, note how indenting is used to denote the function's body and how a semi-colon is used to terminate the function's definition.

Function arguments

- Often functions are required to perform some task based on information passed in by the user.
- These bits of Information are passed in using function arguments.
- Function arguments are defined within the parentheses “()”, which are placed at the end of the function’s name.

Function arguments

- Our new version of printGameOver, can now print out customizable, “Game Over!”, messages by using our new argument called “playersName”.

```
def printGameOver( playersName ):
    print “Game Over... ” + playersName + “!”
```

- Now, when we call our function we can specify which player is being killed off.

Function Return Values

- A function can also output or return a value based on its work.
- The function below calculates and returns the average of a list of numbers.

```
def average( numberList ):
    numCount = 0
    runningTotal = 0

    for n in numberList:
        numCount = numCount + 1
        runningTotal = runningTotal + n

    return runningTotal / numCount
```

- Note how the list's average is returned using the return keyword.

Conclusion

- This concludes your introduction to Python.
- You now know enough of the basics to write useful Python scripts and to teach yourself some of the more advanced features of Python.

Operators and their usage

Operator	Name	Explanation	Examples
+	Plus	Adds the two objects	`3 + 5` gives `8`. `'a' + 'b'` gives `'ab'`.
-	Minus	Either gives a negative number or gives the subtraction of one number from the other	`-5.2` gives a negative number. `50 - 24` gives `26`.
*	Multiply	Gives the multiplication of the two numbers or returns the string repeated that many times.	`2 * 3` gives `6`. `'la' * 3` gives `'lalala'`.
**	Power	Returns x to the power of y	`3 ** 4` gives `81` (i.e. `3 * 3 * 3 * 3`)
/	Divide	Divide x by y	`4/3` gives `1` (division of integers gives an integer). `4.0/3` or `4/3.0` gives `1.3333333333333333`

Operator	Name	Explanation	Examples
//	Floor Division	Returns the floor of the quotient	`4 // 3.0` gives `1.0`
%	Modulo	Returns the remainder of the division	`8%3` gives `2`. `-25.5%2.25` gives `1.5`.
<<	Left Shift	Shifts the bits of the number to the left by the number of bits specified. (Each number is represented in memory by bits or binary digits i.e. 0 and 1)	`2 << 2` gives `8`. `2` is represented by `10` in bits. Left shifting by 2 bits gives `1000` which represents the decimal `8`.
>>	Right Shift	Shifts the bits of the number to the right by the number of bits specified.	`11 >> 1` gives `5` - `11` is represented in bits by `1011` which when right shifted by 1 bit gives `101` which is nothing but decimal `5`.
&	Bitwise AND	Bitwise AND of the numbers	`5 & 3` gives `1`.
|	Bit-wise OR	Bitwise OR of the numbers	`5 | 3` gives `7`
^	Bit-wise XOR	`5 ^ 3` gives `6`	
~	Bit-wise invert	The bit-wise inversion of x is -(x+1)	`~5` gives `-6`.
<	Less Than	Returns whether x is less than y. All comparison operators return `1` for true and `0` for false. This is equivalent to the special variables `True` and `False` respectively. Note the capitalization of these variables' names.	`5 < 3` gives `0` (i.e. `False`) and `3 < 5` gives `1` (i.e. `True`). Comparisons can be chained arbitrarily: `3 < 3 < 7` gives `True`.
>	Greater Than	Returns whether x is greater than y	`5 < 3` returns `True`. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns `False`.
<=	Less Than or Equal To	Returns whether x is less than or equal to y	`x = 3; y = 6; x <= y` returns `True`.
>=	Greater Than or Equal To	Returns whether x is greater than or equal to y	`x = 4; y = 3; x >= 3` returns `True`.
==	Equal To	Compares if the objects are equal	`x = 2; y = 2; x == y` returns `True`. `x = 'str'; y = 'stR'; x == y` returns `False`. `x = 'str'; y = 'str'; x == y` re-

Operator	Name	Explanation	Examples
//	Floor Division	Returns the floor of the quotient	`4 // 3.0` gives `1.0`
%	Modulo	Returns the remainder of the division	`8%3` gives `2`. `-25.5%2.25` gives `1.5`.
<<	Left Shift	Shifts the bits of the number to the left by the number of bits specified. (Each number is represented in memory by bits or binary digits i.e. 0 and 1)	`2 << 2` gives `8`. `2` is represented by `10` in bits. Left shifting by 2 bits gives `1000` which represents the decimal `8`.
>>	Right Shift	Shifts the bits of the number to the right by the number of bits specified.	`11 >> 1` gives `5`. `11` is represented in bits by `1011` which when right shifted by 1 bit gives `101` which is nothing but decimal `5`.
&	Bitwise AND	Bitwise AND of the numbers	`5 & 3` gives `1`.
|	Bit-wise OR	Bitwise OR of the numbers	`5 | 3` gives `7`
^	Bit-wise XOR	`5 ^ 3` gives `6`	
~	Bit-wise invert	The bit-wise inversion of x is -(x+1)	`~5` gives `-6`.
<	Less Than	Returns whether x is less than y. All comparison operators return `1` for true and `0` for false. This is equivalent to the special variables `True` and `False` respectively. Note the capitalization of these variables' names.	`5 < 3` gives `0` (i.e. `False`) and `3 < 5` gives `1` (i.e. `True`). Comparisons can be chained arbitrarily: `3 < 5 < 7` gives `True`.
>	Greater Than	Returns whether x is greater than y	`5 < 3` returns `True`. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns `False`.
<=	Less Than or Equal To	Returns whether x is less than or equal to y	`x = 3; y = 6; x <= y` returns `True`.
>=	Greater Than or Equal To	Returns whether x is greater than or equal to y	`x = 4; y = 3; x >= 3` returns `True`.
==	Equal To	Compares if the objects are equal	`x = 2; y = 2; x == y` returns `True`. `x = 'str'; y = 'stR'; x == y` returns `False`. `x = 'str'; y = 'str'; x == y` re-