A Quick Intro to Python Programming

EE347 - ESAP LABS 2018

General Information

- Unlike C/C++ or Java, Python statements do not end in a semicolon
- In Python, indentation is the way you indicate the scope of a conditional, function, etc.
- Look, no braces!
- Python is interpretive, meaning you don't have to write programs.
- You can just enter statements into the Python environment and they'll execute
- For the most part, we'll be writing programs

The Python Shell

- Because Python is interpretive, you can do simple things with the shell
- In the graphical shell on Linux, double-click on LXTerminal
- At the prompt, type `Python`
- You should have a `>>>` prompt
- Type in:

```
print(“hello, world”)
```

- You have written your first Python program
- Keep the shell up; we’ll be using it

The Python Shell

- This is good for simple calculations but not for real programming
- For programming, we'll use Idle
- There are two versions: Idle for Python 2.7 and Idle3 for Python 3.2
- Idle will give you access to a shell but also to an IDE for writing and saving programs

Python Modules

- In practice, only the simplest programs are run in the shell
- You can create a module by going to the File->New Window menu option
- This brings up a text editor that lets you create a Python program and run it
- Write your first “Hello World!” program thus:

```
print(“Hello, World!”)
```

Python Modules

- Press F5
- It will ask you to save the file before you run it
- Save it to your home directory as HelloWorld.py
- You **must** provide the .py extension
- If you want to run it outside of the development environment simply type:

```
python HelloWorld.py
```

- **Note that Linux is case sensitive**

Variables

- As in every language, a variable is the name of a memory location
- Python is weakly typed
- That is, you don’t declare variables to be a specific type
- A variable has the type that corresponds to the value you assign to it
- Variable names begin with a letter or an underscore and can contain letters, numbers, and underscores
- Python has reserved words that you can’t use as variable names

Variables

- At the >>> prompt, do the following:

```
x=5
type(x)
x=“this is text”
type(x)
x=5.0
type(x)
```

Printing

- You've already seen the print statement
- You can also print numbers with formatting
- These are identical to Java or C format specifiers

Comments

- All code must contain comments that describe what it does
- In Python, lines beginning with a # sign are comment lines
- On American English keyboards, this is over the 3 key; I don't know where it is on British English keyboards

You can also have comments on the same line as a statement

```
# This entire line is a comment
x=5       # Set up loop counter
```

Operators

- Arithmetic operators we will use:

`+ - * /`	addition, subtraction/negation, multiplication, division
`%`	modulus, a.k.a. remainder
`**`	exponentiation

- **precedence**: Order in which operations are computed.
  - `* / % **` have a higher precedence than `+ -`

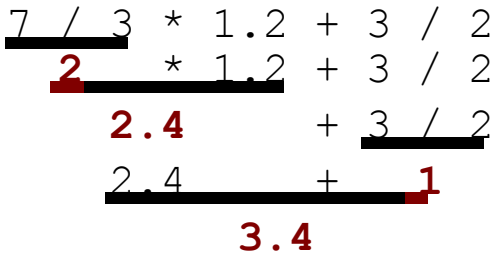
    `1 + 3 * 4` is `13`

  - Parentheses can be used to force a certain order of evaluation.

    `(1 + 3) * 4` is `16`

Expressions

- When integers and reals are mixed, the result is a real number.
  - Example: `1 / 2.0` is `0.5`
  - The conversion occurs on a per-operator basis.



Math Functions

- Use this at the top of your program: `from math import *`

Command name	Description
`abs(`***value***`)`	absolute value
`ceil(`***value***`)`	rounds up
`cos(`***value***`)`	cosine, in radians
`floor(`***value***`)`	rounds down
`log(`***value***`)`	logarithm, base *e*
`log10(`***value***`)`	logarithm, base 10
`max(`***value1***`, `***value2***`)`	larger of two values
`min(`***value1***`, `***value2***`)`	smaller of two values
`round(`***value***`)`	nearest whole number
`sin(`***value***`)`	sine, in radians
`sqrt(`***value***`)`	square root

Constant	Description
`e`	2.7182818...
`pi`	3.1415926...

Relational Operators

- Many logical expressions use *relational operators*:

Operator	Meaning	Example	Result
==	equals	1 + 1 == 2	True
!=	does not equal	3.2 != 2.5	True
<	less than	10 < 5	False
>	greater than	10 > 5	True
<=	less than or equal to	126 <= 100	False
>=	greater than or equal to	5.0 >= 5.0	True

Logical Operators

- These operators return true or false

Operator	Example	Result
and	`9 != 6 and 2 < 3`	True
or	`2 == 3 or -1 < 5`	True
not	`not 7 > 0`	False

The `if` Statement

- Syntax:

```
if <condition>:
     <statements>


x = 5
if x > 4:
     print(“x is greater than 4”)
print(“This is not in the scope of the if”)
```

The `if` Statement

- The colon is required for the `if`
- Note that all statement indented one level in from the `if` are within it scope:

```
x = 5
if x > 4:
    print(“x is greater than 4”)
    print(“This is also in the scope of the if”)
```

The `if/else` Statement

```
if <condition>:
     <statements>
else:
     <statements>
```

- Note the colon following the else
- This works exactly the way you would expect

The `for` Loop

- This is similar to what you're used to from C or Java, but not the same
- Syntax:

```
for variableName in groupOfValues:
    <statements>
```

- variableName gives a name to each value, so you can refer to it in the statements.
- groupOfValues can be a range of integers, specified with the range function.

- Example:

```
for x in range(1, 6):
    print x, "squared is", x * x
```

Range

- The range function specifies a range of integers:

`range(start, stop)` - the integers between start (inclusive) and stop (exclusive)

- It can also accept a third value specifying the change between values.

`range(start, stop, step)` - the integers between start (inclusive) and stop (exclusive) by step

The `while` Loop

- Executes a group of statements as long as a condition is True.
- Good for indefinite loops (repeat an unknown number of times)
- Syntax:

```
while <condition>:
    <statements>
```

- Example:

```
number = 1
while number < 200:
    print number,
    number = number * 2
```

Strings

- String: A sequence of text characters in a program.
- Strings start and end with quotation mark " or apostrophe ' characters.
- Examples:
  "hello"
  "This is a string"
  "This, too, is a string. It can be very long!"
- A string may not span across multiple lines or contain a " character.
  "This is not
  a legal String."

  "This is not a "legal" String either."

Strings

- A string can represent characters by preceding them with a backslash.
  - \t tab character
  - \n new line character
  - \" quotation mark character
  - \\ backslash character

- Example: "Hello\tthere\nHow are you?"

Indexing Strings

- As with other languages, you can use square brackets to index a string as if it were an array:

```
name = “Arpita Nigam”
print(name, “starts with “, name[0])
```

String Functions

- `len(`***`string`***`)` - number of characters in a string
- `str.lower(`***`string`***`)` - lowercase version of a string
- `str.upper(`***`string`***`)` - uppercase version of a string
- `str.isalpha(`***`string`***`)` - True if the string has only alpha chars
- Many others: split, replace, find, format, etc.

- Note the “dot” notation: These are static methods.

Byte Arrays and Strings

- Strings are Unicode text and not mutable
- Byte arrays are mutable and contain raw bytes
- For example, reading Internet data from a URL gets bytes
- Convert to string:

```
cmd = response.read()
strCmd = str(cmd)
```

Other Built-in Types

- tuples, lists, sets, and dictionaries
- They all allow you to group more than one item of data together under one name
- You can also search them

Tuples

- Unchanging Sequences of Data
- Enclosed in parentheses:

```
tuple1 = (“This”, “is”, “a”, “tuple”)
print(tuple1)
```

- This prints the tuple exactly as shown

```
Print(tuple1[1])
```

- Prints “is” (without the quotes)

Lists

- Changeable sequences of data
- Lists are created by using square brackets:

```
breakfast = [ “coffee”, “tea”, “toast”, “egg” ]
```

- You can add to a list:

```
breakfast.append(“waffles”)
breakfast.extend([“cereal”, “juice”])
```

Dictionaries

- Groupings of Data Indexed by Name
- Dictionaries are created using braces

```
sales = {}
sales[“January”] = 10000
sales[“February”] = 17000
sales[“March”] = 16500
```

- The keys method of a dictionary gets you all of the keys as a list

Sets

- Sets are similar to dictionaries in Python, except that they consist of only keys with no associated values.
- Essentially, they are a collection of data with no duplicates.
- They are very useful when it comes to removing duplicate data from data collections.

Writing Functions

- Define a function:

```
def <function name>(<parameter list>)
```

- The function body is indented one level:

```
def computeSquare(x)
     return x * x
# Anything at this level is not part of the function
```

Python File I/O

- You can read and write text files in Python much as you can in other languages, and with a similar syntax.
- To open a file for reading:

```
try:
    configFile = open(configName, "r")
except IOError as err:
    print(“could not open file: “ + str(err))
```

Python File I/O

- To read from a file:

```
while 1:
   line = configFile.readline()
   if len(line) == 0:
      break
```

Python File I/O

- You can also read all lines from a file into a set, then iterate over the set:

```
lines = file.readlines()
for line in lines:
    print(line)
file.close()
```

Python File I/O

- Writing to a text file

```
file=open('test.txt',"w")
file.write("This is how you create a new text file")
file.close()
```