Python Basics 101

Python : Introduction

- Created in 1990 by **Guido Van Rossum**
- Great as a first language as it is concise and easy to read
- A general-purpose, versatile and popular programming language.
- Python actually got its name from a BBC comedy series from the seventies **"Monty Python's Flying Circus"**. The designer needed a name that was short, unique, and slightly mysterious. Since he was a fan of the show he thought this name was great. :)

Applications

- Object Oriented
- Interpreted
- Multi-purpose(Scripting, GUI, Software development, Gaming)
- Strongly typed and Dynamically typed

Who all use python?

- Yahoo, Google, Linux..
- Battlefield 2, Star Trek Bridge Commander..
- Walt Disney animations, RoboFrog..
- NASA, ESRI..
- ..and list goes on..

Why is Python better than other languages?

- Easier to learn
- More readable code
- Concise and clean code
- YES, it is slow, but development time is so much faster!

Should I use Python 2 or Python 3 for my development activity?

Short version: Python 2.x is legacy, Python 3.x is the present and future of the language.

- Python 3.0 was released in 2008. The final 2.x version 2.7 release came out in mid-2010, with a statement of extended support for this end-of-life release.
- The 2.x branch will see no new major releases after that.
- 3.x is under active development and has already seen over five years of stable releases,This means that all recent standard library improvements, for example, are only available by default in Python 3.x

The __future__ module

Python 3.x introduced some Python 2-incompatible keywords and features that can be imported via the in-built __future__ module in Python 2. IFor example, if we want Python 3.x's integer division behavior in Python 2, we can import it via

```
from __future__ import division
```

The print function

Python 2's print statement has been replaced by the print() function, meaning that we have to wrap the object that we want to print in parentheses.

Python Basics

Python programs

- Saved with extension .py
- Same py file can be executed as program or module
- Python modules and programs are differentiated by :
  - .py files directly executed are programs
  - .py files included using ‘import’ statement are modules

Remember..

- Python is case-sensitive
- Python embraces indentation
- Indentation is whitespace; not curly brackets

Variables, Expressions, and Statements

Constants

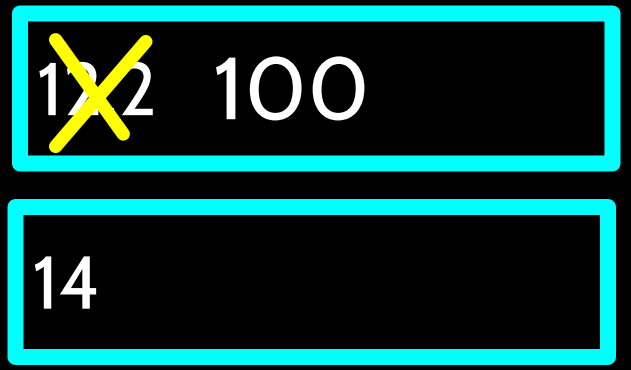
- Fixed values such as numbers, letters, and strings are called "constants" because their value does not change
- Numeric constants are as you expect
- String constants use single quotes (') or double quotes (")

```
>>> print 123
123
>>> print 98.6
98.6
>>> print 'Hello world'
Hello world
```

Variables

- A variable is a named place in the memory where a programmer can store data and later retrieve the data using the variable “name”
- Programmers get to choose the names of the variables
- You can change the contents of a variable in a later statement

```
x = 12.2
y = 14
x = 100
```



Python Variable Name Rules

1. Must start with a letter or underscore _
2. Must consist of letters and numbers and underscores
3. Case Sensitive

- Good: spam eggs spam23 _speed
- Bad: 23spam #sign var.12
- Different: spam Spam SPAM

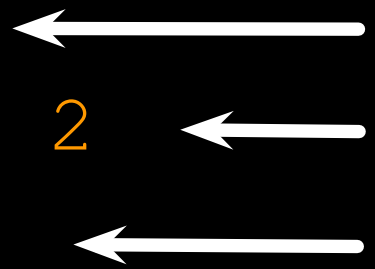
Reserved Words

- You cannot use reserved words as variable names / identifiers

```
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 as with
```

Sentences or Lines

```
x = 2
x = x + 2
print x
```



Assignment statement

Assignment with expression

Print statement

Variable

Operator

Constant

Reserved Word

Assignment Statements

- We assign a value to a variable using the assignment statement (=)
- An assignment statement consists of an expression on the right-hand side and a variable to store the result

```
x = 3.9 * x * ( 1 - x )
```

Numeric Expressions

- Because of the lack of mathematical symbols on computer keyboards - we use “computer-speak” to express the classic math operations
- Asterisk is multiplication
- Exponentiation (raise to a power) looks different from in math.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Numeric Expressions

```
>>> xx = 2
>>> xx = xx + 2
>>> print xx
4
>>> yy = 440 * 12
>>> print yy
5280
>>> zz = yy / 1000
>>> print zz
5
```

```
>>> jj = 23
>>> kk = jj % 5
>>> print kk
3
>>> print 4 ** 3
64
```

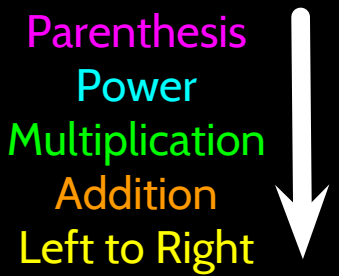
```
    4 R 3
  ┌─────
5 │ 23
    20
   ──
     3
```

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Operator Precedence Rules

Highest precedence rule to lowest precedence rule:

- Parenthesis are always respected
- Exponentiation (raise to a power)
- Multiplication, Division, and Remainder
- Addition and Subtraction
- Left to right



Python Integer Division is Weird!

- Integer division truncates
- Floating point division produces floating point numbers

```
>>> print 10 / 2
5
>>> print 9 / 2
4
>>> print 99 / 100
0
>>> print 10.0 / 2.0
5.0
>>> print 99.0 / 100.0
0.99
```

This changes in Python 3.0

Mixing Integer and Floating

- When you perform an operation where one operand is an integer and the other operand is a floating point, the result is a floating point
- The integer is converted to a floating point before the operation

```
>>> print 99 / 100
0
>>> print 99 / 100.0
0.99
>>> print 99.0 / 100
0.99
>>> print 1 + 2 * 3 / 4.0 - 5
-2.5
>>>
```

What does “Type” Mean?

- In Python variables, literals and constants have a “type”
- Python knows the difference between an integer number and a string
- For example “+” means “addition” if something is a number and “concatenate” if something is a string

```
>>> ddd = 1 + 4
>>> print ddd
5
>>> eee = 'hello ' + 'there'
>>> print eee
hello there
```

concatenate = put together

Type Matters

- Python knows what “type” everything is
- Some operations are prohibited
- You cannot “add 1” to a string
- We can ask Python what type something is by using the type() function

```
>>> eee = 'hello ' + 'there'
>>> eee = eee + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: cannot concatenate
'str' and 'int' objects
>>> type(eee)
<type 'str'>
>>> type('hello')
<type 'str'>
>>> type(1)
<type 'int'>
>>>
```

Several Types of Numbers

- Numbers have two main types
  - Integers are whole numbers: -14, -2, 0, 1, 100, 401233
  - Floating Point Numbers have decimal parts: -2.5 , 0.0, 98.6, 14.0
- There are other number types - they are variations on float and integer

```
>>> xx = 1
>>> type (xx)
<type 'int'>
>>> temp = 98.6
>>> type(temp)
<type 'float'>
>>> type(1)
<type 'int'>
>>> type(1.0)
<type 'float'>
>>>
```

Type Conversions

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float

- You can control this with the built-in functions int() and float()

```
>>> print float(99) / 100
0.99
>>> i = 42
>>> type(i)
<type 'int'>
>>> f = float(i)
>>> print f
42.0
>>> type(f)
<type 'float'>
>>> print 1 + 2 * float(3) / 4 - 5
-2.5
>>>
```

String Conversions

- You can also use int() and float() to convert between strings and integers
- You will get an error if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<type 'str'>
>>> print sval + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: cannot concatenate 'str'
and 'int'
>>> ival = int(sval)
>>> type(ival)
<type 'int'>
>>> print ival + 1
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
ValueError: invalid literal for
int()
```

User Input

- We can instruct Python to pause and read data from the user using the raw_input() function
- The raw_input() function returns a string

```
nam = raw_input('Who are you?')
print 'Welcome', nam
```

Who are you? Chuck
Welcome Chuck

Converting User Input

- If we want to read a number from the user, we must convert it from a string to a number using a type conversion function

```
inp = raw_input('Europe floor?')
usf = int(inp) + 1
print 'US floor', usf
```

- Later we will deal with bad input data

Europe floor? 0
US floor 1

Comments in Python

- Anything after a # is ignored by Python
- Why comment?
  - Describe what is going to happen in a sequence of code
  - Document who wrote the code or other ancillary information
  - Turn off a line of code - perhaps temporarily

```
# Get the name of the file and open it
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
words = text.split()

# Count word frequency
counts = dict()
for word in words:
   counts[word] = counts.get(word,0) + 1

# Find the most common word
bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count >
bigcount:
        bigword = word
        bigcount = count

# All done
print bigword, bigcount
```

String Operations

- Some operators apply to strings
  - + implies “concatenation”
  - * implies “multiple concatenation”
- Python knows when it is dealing with a string or a number and behaves appropriately

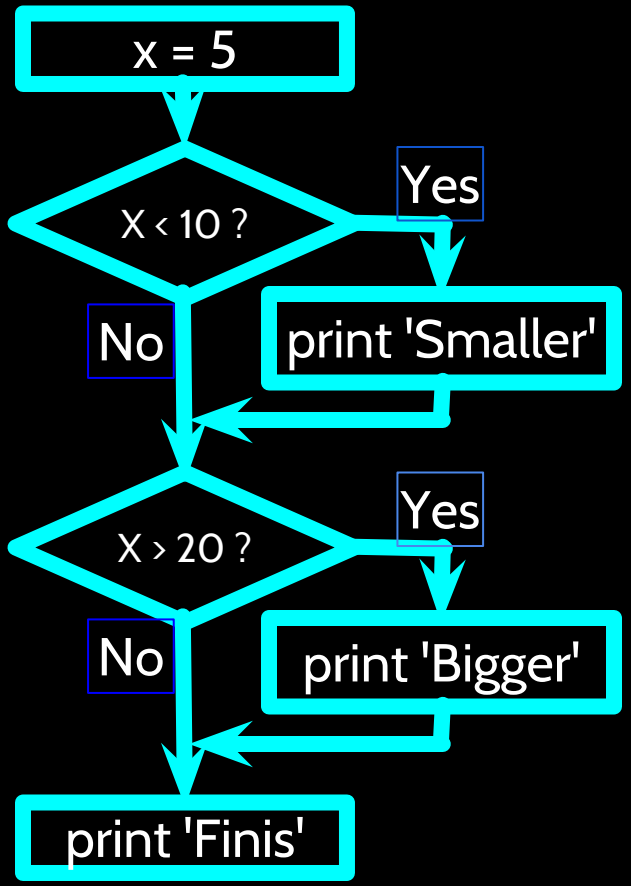
```
>>> print 'abc' + '123'
abc123
>>> print 'Hi' * 5
HiHiHiHiHi
>>>
```

Mnemonic Variable Names

- Since we programmers are given a choice in how we choose our variable names, there is a bit of “best practice”
- We name variables to help us remember what we intend to store in them (“mnemonic” = “memory aid”)
- This can confuse beginning students because well-named variables often “sound” so good that they must be keywords

http://en.wikipedia.org/wiki/Mnemonic

Conditional Execution



Conditional Steps

Program:

```
x = 5
if x < 10:
   print 'Smaller'

if x > 20:
    print 'Bigger'

print 'Finis'
```

Output:

```
Smaller
Finis
```

Indentation

- Increase indent indent after an if statement or for statement (after :)
- Maintain indent to indicate the scope of the block (which lines are affected by the if/for)
- Reduce indent *back to* the level of the if statement or for statement to indicate the end of the block
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored with regard to indentation

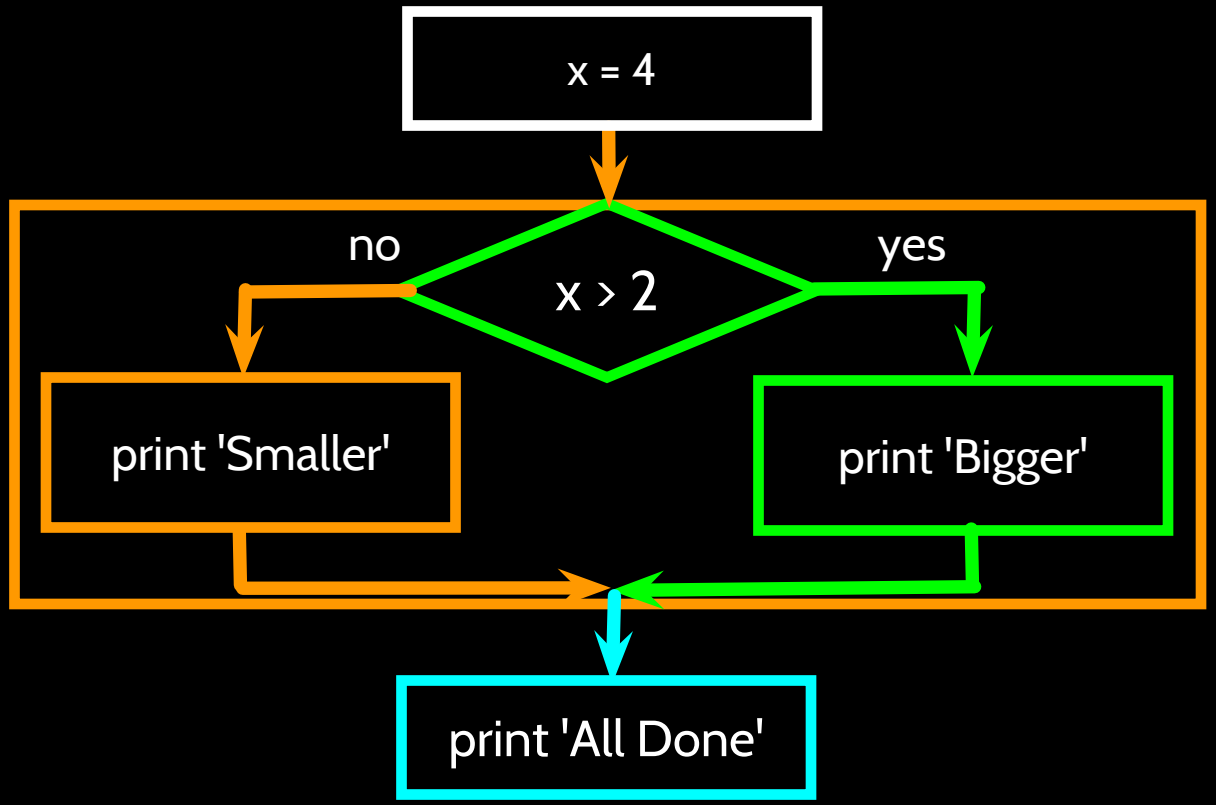
Warning: Turn Off Tabs!!

- Most text editors can turn tabs into spaces - make sure to enable this feature
  - > NotePad++: Settings -> Preferences -> Language Menu/Tab Settings
  - > TextWrangler: TextWrangler -> Preferences -> Editor Defaults
- Python cares a *lot* about how far a line is indented. If you mix tabs and spaces, you may get "indentation errors" even if everything looks fine

Please do this now while you are thinking about it so we can all stay sane...

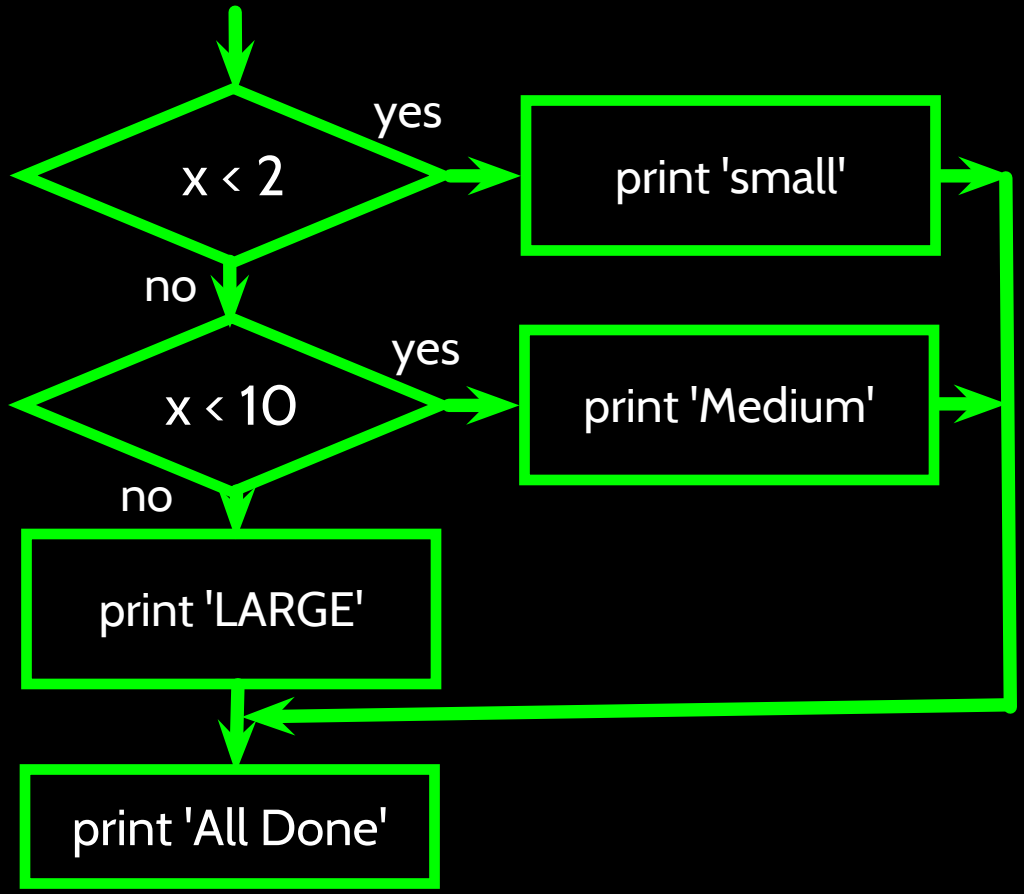
Two-way using else :

```
x = 4

if x > 2 :
    print 'Bigger'
else :
    print 'Smaller'

print 'All done'
```



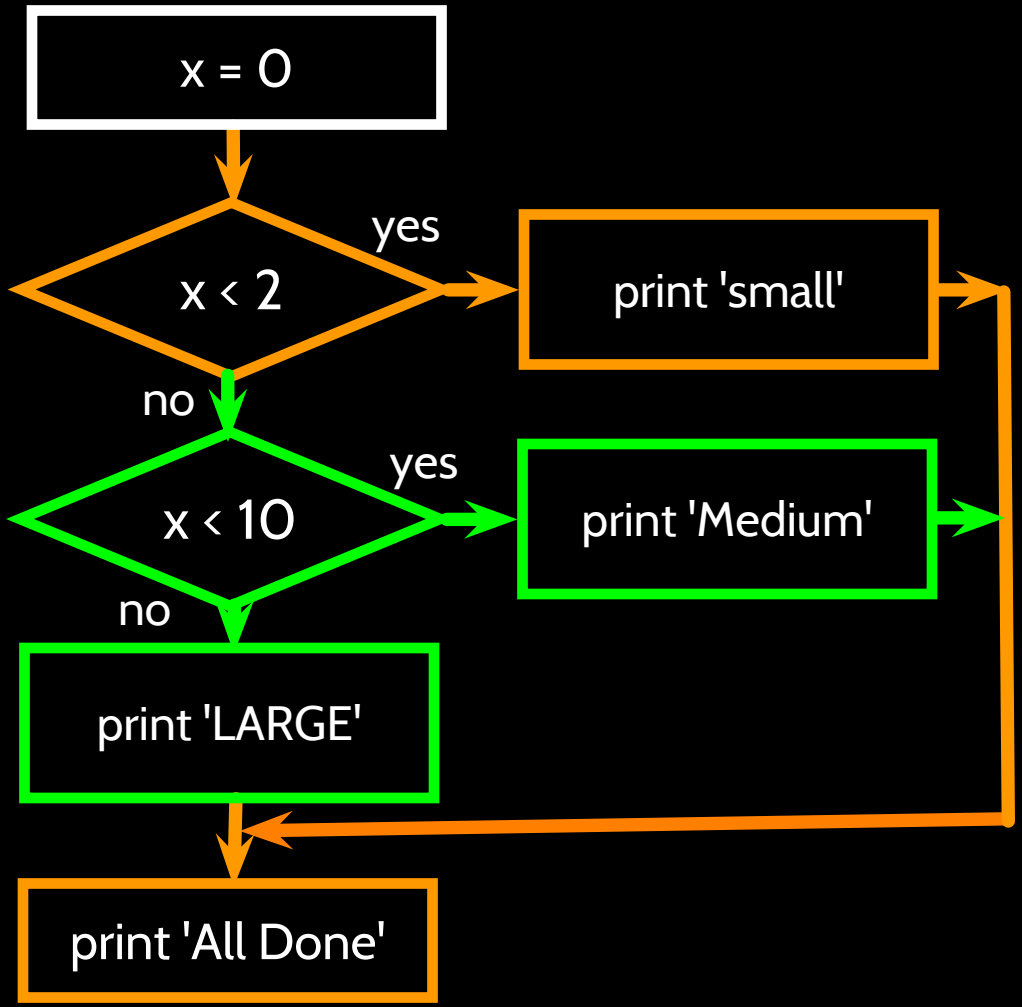
Multi-way

```
if x < 2 :
    print 'small'
elif x < 10 :
    print 'Medium'
else :
    print 'LARGE'
print 'All done'
```



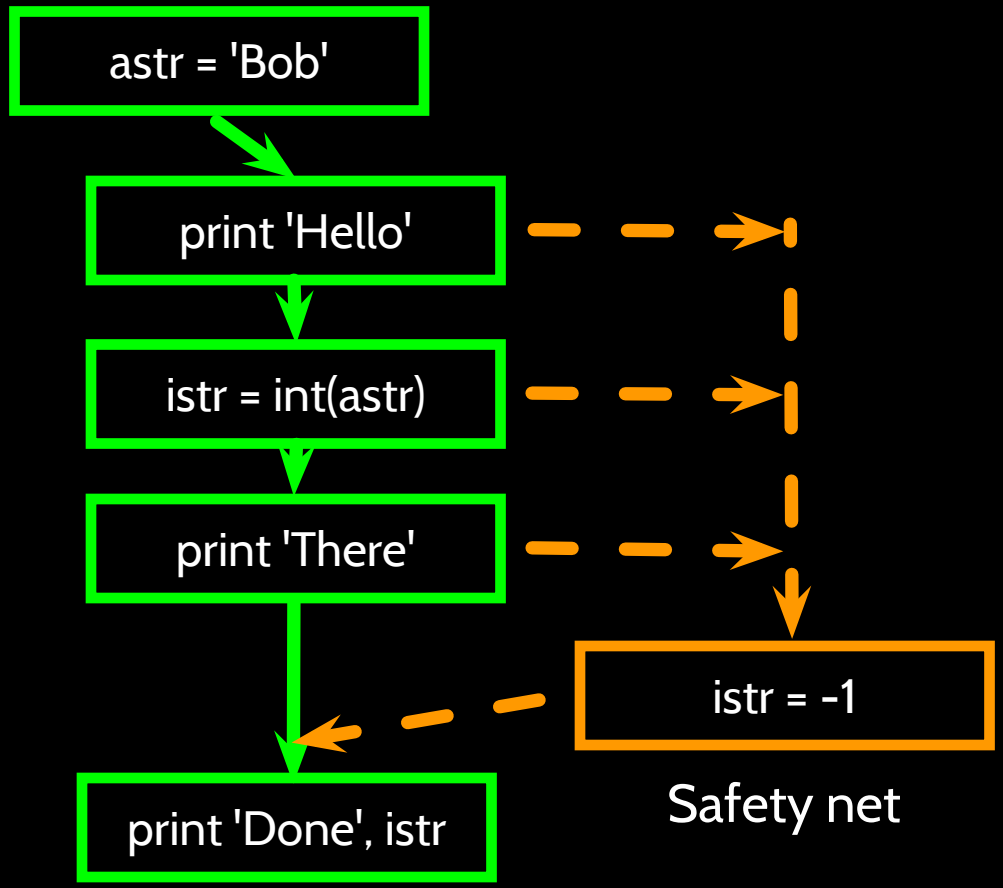
Multi-way

```
x = 0
if x < 2 :
    print 'small'
elif x < 10 :
    print 'Medium'
else :
    print 'LARGE'
print 'All done'
```



try / except

```
astr = 'Bob'
try:
    print 'Hello'
    istr = int(astr)
    print 'There'
except:
    istr = -1

print 'Done', istr
```



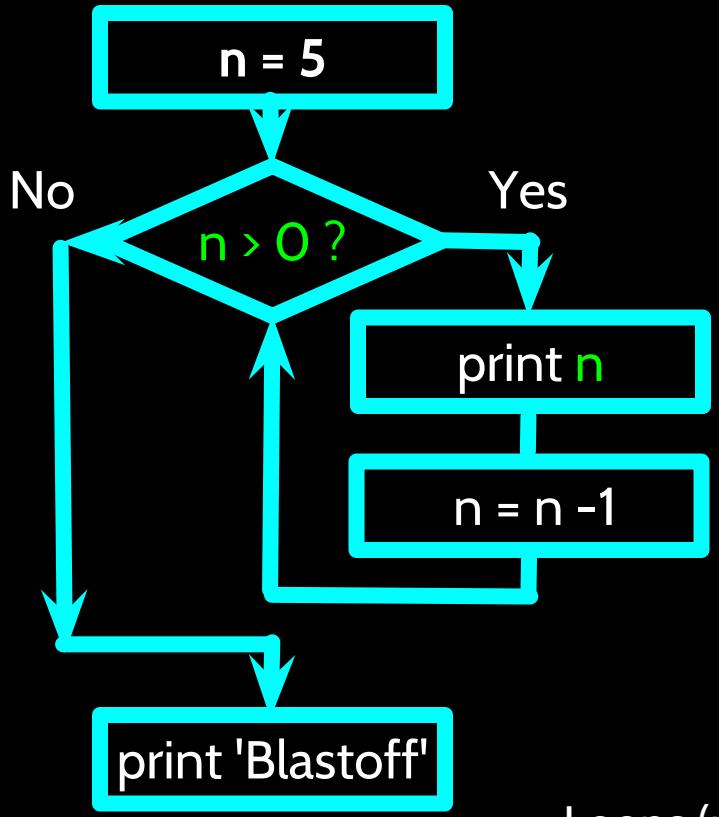
Sample try / except

```
rawstr = raw_input('Enter a number:')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print 'Nice work'
else:
    print 'Not a number'
```

```
$ python trynum.py
Enter a number:42
Nice work
$ python trynum.py
Enter a number:forty-two
Not a number
$
```

Loops and Iteration

Repeated Steps



Program:

```
n = 5
while n > 0 :
    print n
    n = n - 1
print 'Blastoff!'
print n
```

Output:

```
5
4
3
2
1
Blastoff!
0
```

Loops (repeated steps) have iteration variables that change each time through a loop. Often these iteration variables go through a sequence of numbers.

Breaking Out of a Loop

- The break statement ends the current loop and jumps to the statement immediately following the loop

- It is like a loop test that can happen anywhere in the body of the loop

```
while True:
    line = raw_input('> ')
    if line == 'done' :
        break
    print line
print 'Done!'
```

```
> hello there
hello there
> finished
finished
> done
Done!
```

Loop Idioms:
What We Do in Loops

Note: Even though these examples are simple, the patterns apply to all kinds of loops

Making “smart” loops

The trick is “knowing” something about the whole loop when you are stuck writing code that only sees one entry at a time

Set some variables to initial values

for thing in data:

Look for something or do something to each entry separately, updating a variable

Look at the variables

Looping through a Set

```
print 'Before'
for thing in [9, 41, 12, 3, 74, 15] :
     print thing
print 'After'
```

```
$ python
basicloop.py
Before
9
41
12
3
74
15
After
```

Functions

Building our Own Functions

- We create a new function using the def keyword followed by optional parameters in parentheses
- We indent the body of the function
- This defines the function but *does not* execute the body of the function

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all
day.'
```

print_lyrics():

```
print "I'm a lumberjack, and I'm okay."
print 'I sleep all night and I work all day.'
```

```
x = 5
print 'Hello'

def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all
day.'

print 'Yo'
x = x + 2
print x
```

```
Hello
Yo
7
```

Definitions and Uses

- Once we have defined a function, we can call (or invoke) it as many times as we like
- This is the store and reuse pattern

Parameters

A parameter is a variable which we use in the function definition. It is a "handle" that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):
...     if lang == 'es':
...         print 'Hola'
...     elif lang == 'fr':
...         print 'Bonjour'
...     else:
...         print 'Hello'
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

Return Values

Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
def greet():
    return "Hello"

print greet(), "Glenn"
print greet(), "Sally"
```

```
Hello Glenn
Hello Sally
```

To function or not to function…

- Organize your code into “paragraphs” - capture a complete thought and “name it”
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over - perhaps share this with your friends…

Python Lists

A List is a kind of Collection

- A collection allows us to put many values in a single “variable”
- A collection is nice because we can carry all many values around in one convenient package.

```
friends = [ 'Joseph', 'Glenn', 'Sally' ]

carryon = [ 'socks', 'shirt', 'perfume' ]
```

List Constants

- List constants are surrounded by square brackets and the elements in the list are separated by commas
- A list element can be any Python object - even another list
- A list can be empty

```
>>> print [1, 24, 76]
[1, 24, 76]
>>> print ['red', 'yellow', 'blue']
['red', 'yellow', 'blue']
>>> print ['red', 24, 98.6]
['red', 24, 98.599999999999994]
>>> print [ 1, [5, 6], 7]
[1, [5, 6], 7]
>>> print []
[]
```

Lists are Mutable

- Strings are "immutable" - we *cannot* change the contents of a string - we must make a new string to make any change
- Lists are "mutable' - we *can* change an element of a list using the index operator

```
>>> fruit = 'Banana'
>>> fruit[0] = 'b'
Traceback
TypeError: 'str' object does not
support item assignment
>>> x = fruit.lower()
>>> print x
banana
>>> lotto = [2, 14, 26, 41, 63]
>>> print lotto
[2, 14, 26, 41, 63]
>>> lotto[2] = 28
>>> print lotto
[2, 14, 28, 41, 63]
```

How Long is a List?

- The len() function takes a list as a parameter and returns the number of *elements* in the list
- Actually len() tells us the number of elements of *any* set or sequence (such as a string...)

```
>>> greet = 'Hello Bob'
>>> print len(greet)
9
>>> x = [ 1, 2, 'joe', 99]
>>> print len(x)
4
>>>
```

Using the range function

- The range function returns a list of numbers that range from zero to one less than the parameter
- We can construct an index loop using for and an integer iterator

```
>>> print range(4)
[0, 1, 2, 3]
>>> friends = ['Joseph', 'Glenn',
'Sally']
>>> print len(friends)
3
>>> print range(len(friends))
[0, 1, 2]
>>>
```

A List is an Ordered Sequence

- A list can hold many items and keeps those items in the order until we do something to change the order
- A list can be sorted (i.e., change its order)
- The sort method (unlike in strings) means "sort yourself"

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally'
]
>>> friends.sort()
>>> print friends
['Glenn', 'Joseph', 'Sally']
>>> print friends[1]
Joseph
>>>
```

Python Objects

Object Oriented

- A program is made up of many cooperating objects
- Instead of being the “whole program” - each object is a little “island” within the program and cooperatively working with other objects.
- A program is made up of one or more objects working together - objects make use of each other’s capabilities

Object

- An Object is a bit of self-contained Code and Data
- A key aspect of the Object approach is to break the problem into smaller understandable parts (divide and conquer)
- Objects have boundaries that allow us to ignore un-needed detail
- We have been using objects all along: String Objects, Integer Objects, Dictionary Objects, List Objects...

```
movies = list()
movie1 = dict()
movie1['Director'] = 'James Cameron'
movie1['Title'] = 'Avatar'
movie1['Release Date'] = '18 December
2009'
movie1['Running Time'] = '162 minutes'
movie1['Rating'] = 'PG-13'
movies.append(movie1)
movie2 = dict()
movie2['Director'] = 'David Fincher'
movie2['Title'] = 'The Social Network'
movie2['Release Date'] = '01 October 2010'
movie2['Running Time'] = '120 min'
movie2['Rating'] = 'PG-13'
movies.append(movie2)
```

Definitions

- Class – a template - Dog
- Method or Message – A defined capability of a class – bark()
- Field or attribute– A bit of data in a class - length
- Object or Instance – A particular instance of a class – Lassie

Terminology: Class

Defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). One might say that a class is a blueprint or factory that describes the nature of something. For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors).

http://en.wikipedia.org/wiki/Object-oriented_programming

Terminology: Class

A pattern (exemplar) of a class. The class of Dog defines all possible dogs by listing the characteristics and behaviors they can have; the object Lassie is one particular dog, with particular versions of the characteristics. A Dog has fur; Lassie has brown-and-white fur.

http://en.wikipedia.org/wiki/Object-oriented_programming

Terminology: Instance

One can have an instance of a class or a particular object. The instance is the actual object created at runtime. In programmer jargon, the Lassie object is an instance of the Dog class. The set of values of the attributes of a particular object is called its state. The object consists of state and the behavior that's defined in the object's class.

Object and Instance are often used interchangeably.

http://en.wikipedia.org/wiki/Object-oriented_programming

Terminology: Method

An object's abilities. In language, methods are verbs. Lassie, being a Dog, has the ability to bark. So bark() is one of Lassie's methods. She may have other methods as well, for example sit() or eat() or walk() or save_timmy(). Within the program, using a method usually affects only one particular object; all Dogs can bark, but you need only one particular dog to do the barking

Method and Message are often used interchangeably.

http://en.wikipedia.org/wiki/Object-oriented_programming

class is a reserved word.

Each PartyAnimal object has a bit of code.

Tell the object to run the party() code.

```
class PartyAnimal:
  x = 0

  def party(self) :
    self.x = self.x + 1
    print "So far",self.x

an = PartyAnimal()

an.party()
an.party()
an.party()
```

This is the template for making PartyAnimal objects.

Each PartyAnimal object has a bit of data.

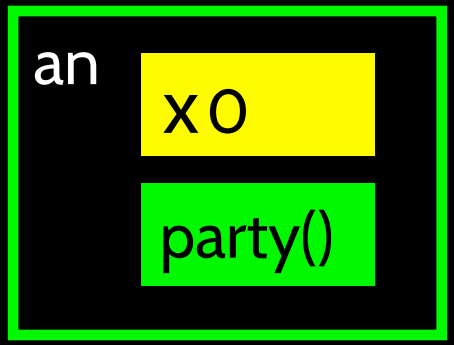
Create a PartyAnimal object.

PartyAnimal.party(an)

run party() *within* the object an

```
class PartyAnimal:
  x = 0

  def party(self) :
    self.x = self.x + 1
    print "So far",self.x

an = PartyAnimal()

an.party()
an.party()
an.party()
```

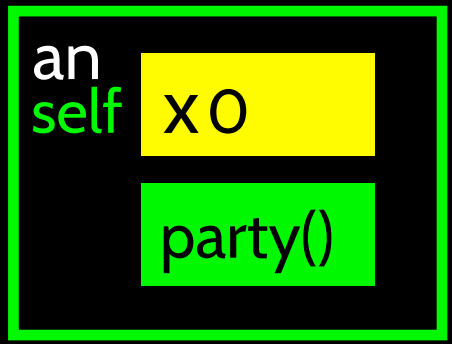
$ python party1.py



```
class PartyAnimal:
  x = 0

  def party(self) :
    self.x = self.x + 1
    print "So far",self.x

an = PartyAnimal()

an.party()
an.party()
an.party()
```

"self" is a formal argument that refers to the object itself.

self.x is saying "x within self"



self is "global within this object"

Definitions Review

- Class - a template - Dog
- Method or Message - A defined capability of a class - bark()
- Object or Instance - A particular instance of a class - Lassie

Object Lifecycle

- Objects are created, used and discarded
- We have special blocks of code (methods) that get called
  - At the moment of creation (constructor)
  - At the moment of destruction (destructor)
- Constructors are used a lot
- Destructors are seldom used

Constructor

- The primary purpose of the constructor is to set up some instance variables to have the proper initial values when the object is created

```
class PartyAnimal:
   x = 0

   def __init__(self):
     print "I am constructed"

   def party(self) :
     self.x = self.x + 1
     print "So far",self.x

   def __del__(self):
     print "I am destructed",
self.x

an = PartyAnimal()
an.party()
an.party()
an.party()
```

```
$ python party2.py
I am constructed
So far 1
So far 2
So far 3
I am destructed 3
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

THANK YOU !