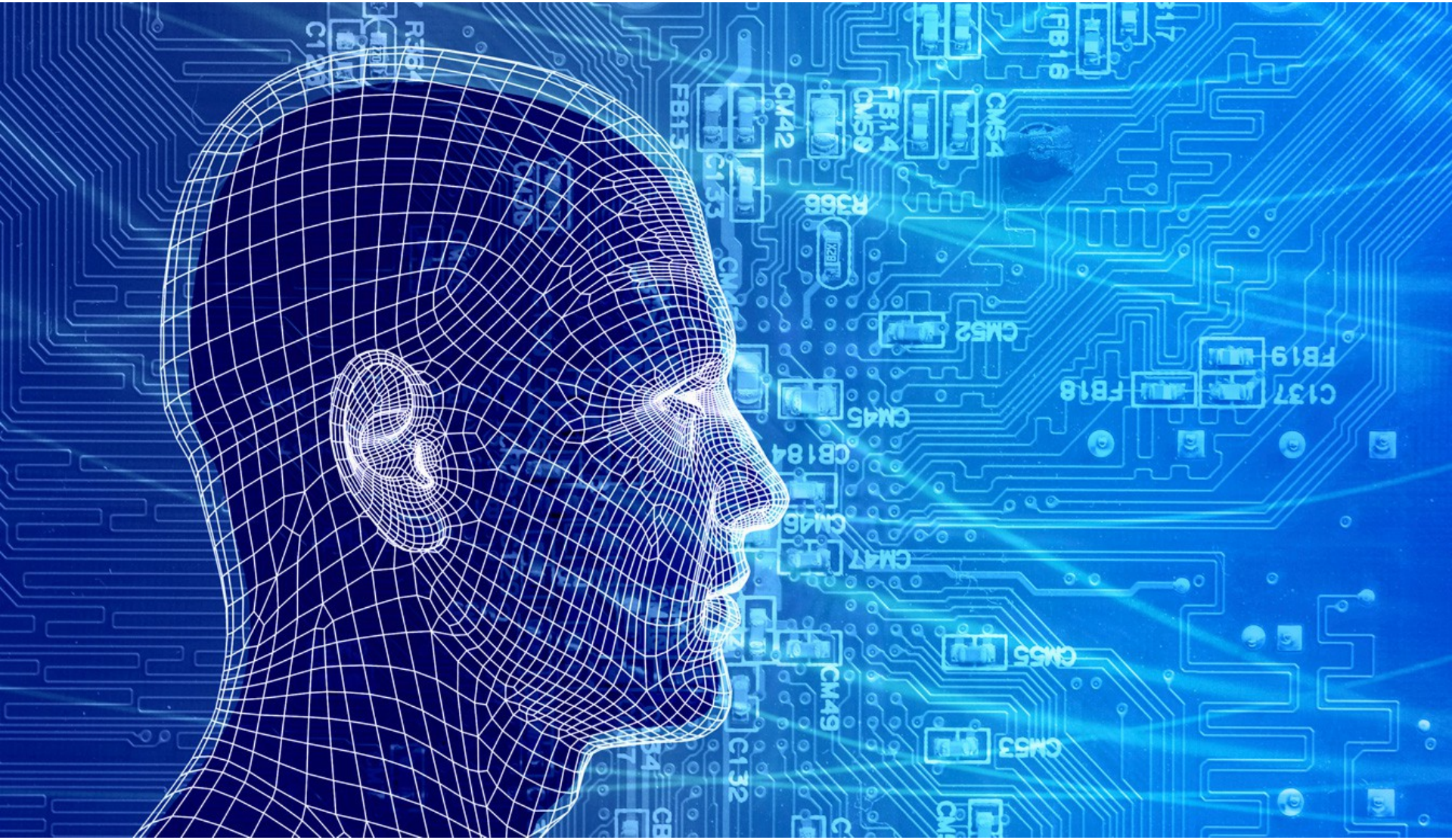
Can machines think like human beings and beyond!



Lets Learn Python

Course Contents

- About Python, Why Python?
- Running Python
- Working with basic types
- Mutable Vs Immutable
- Types and Operators
- List , Tuples and Strings
- Indexing, Slicing
- Basic Statements
- Functions
- Scope Rules (Locality and Context)

History [edit]

Main article: History of Python

Guido van Rossum, the creator of Python

Python was conceived in the late 1980s,[29] and its implementation began in December 1989[30] by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language (itself inspired by SETL)[31] capable of exception handling and interfacing with the Amoeba operating system.[7] Van Rossum remains Python's principal author. His continuing central role in Python's development is reflected in the title given to him by the Python community: *Benevolent Dictator For Life* (BDFL).

On the origins of Python, Van Rossum wrote in 1996:[32]

> ...In December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of *Monty Python's Flying Circus*).
>
> — Guido van Rossum

Python 2.0 was released on 16 October 2000 and had many major new features, including a cycle-detecting garbage collector and support for Unicode. With this release, the development process became more transparent and community-backed.[33]

Python 3.0 (initially called Python 3000 or py3k) was released on 3 December 2008 after a long testing period. It is a major revision of the language that is not completely backward-compatible with previous versions.[34] However, many of its major features have been backported to the Python 2.6.x[35] and 2.7.x version series, and releases of Python 3 include the `2to3` utility, which automates the translation of Python 2 code to Python 3.[36]

Python 2.7's end-of-life date was initially set at 2015, then postponed to 2020 out of concern that a large body of existing code could not easily be forward-ported to Python 3.[37][38]

Python 3.6 had changes regarding UTF-8 (in Windows, PEP 528 and PEP 529) and Python 3.7.0b1 (PEP 540) adds a new "UTF-8 Mode" (and overrides POSIX

Why Use Python?

Python is object-oriented

Structure supports such concepts as polymorphism, operation overloading, and multiple inheritance

It's free (open source)

Downloading and installing Python is free and easy

Source code is easily accessible

Free doesn't mean unsupported! Online Python community is huge

It's portable

Python runs virtually every major platform used today

As long as you have a compatible Python interpreter installed, Python programs will run in exactly the same manner, irrespective of platform

It's powerful

Dynamic typing

Built-in types and tools

Library utilities

Third party utilities (e.g. Numeric, NumPy, SciPy)

Automatic memory management

Why Use Python?

It's mixable

Python can be linked to components written in other languages easily

- Linking to fast, compiled code is useful to computationally intensive problems
- Python is good for code steering and for merging multiple programs in otherwise conflicting languages

Python/C integration is quite common

It's easy to use

Rapid turnaround: no intermediate compile and link steps as in C or C++

Python programs are compiled automatically to an intermediate form called *bytecode*, which the interpreter then reads

This gives Python the development speed of an interpreter without the performance loss inherent in purely interpreted languages

It's easy to learn

Structure and syntax are pretty intuitive and easy to grasp

Running Python

```
Apples-MacBook-Pro:objectDetectionAI apple$ python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 12:04:33)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on
darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print('hello')
hello
>>>
```

In addition to being a programming language, Python is also an interpreter. The interpreter reads other Python programs and commands, and executes them. Note that Python programs are compiled automatically before being scanned into the interpreter. The fact that this process is hidden makes Python faster than a pure interpreter.

First pieces of code

Playing with the interpreter

```
>>> 3+4
7
>>> 270*5+3*(200/56)
1360.7142857142858
>>> 12*7+13*5
149
```

A Code Sample

```
x = 34 - 23              # A comment.
y = "Hello"              # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"     # String concat.
print x
print y
```

Running Python Script file

Python scripts can be written in text files with the suffix .py. The scripts can be read into the interpreter in several ways:

Examples:

```
$ python testscript.py
# This will simply execute the script and return to the terminal afterwards
$ python -i testscript.py
# The -i flag keeps the interpreter open after the script is finished running
$ python
>>> execfile('testscript.py')
# The execfile command reads in scripts and executes them immediately, as though they had been typed into the interpreter directly
$ python
>>> import script # DO NOT add the .py suffix. Script is a module here
# The import command runs the script, displays any unstored outputs, and creates a lower level (or context) within the program.
```

Run first Python Script

Suppose the file myFirstProgram.py contains the following lines:

```
print 'Hello world'
myTestArray = [0,1,2,3,4]
```

Let's run this script in each of the ways described on the last slide:

```
$ python myFirstProgram.py
Hello world
$
```

The script is executed and the interpreter is immediately closed. x is lost.

```
$ python -i myFirstProgram.py
Hello world
>>> print(myTestArray)
[0,1,2,3,4]
>>>
```

“Hello world” is printed, x is stored and can be called later, and the interpreter is left open

Types and Operators: Types of Numbers

Python supports several different numeric types

Integers

Examples: `0, 1, 1234, -56`

Integers are implemented as C longs

Note: dividing an integer by another integer will return only the integer part of the quotient, e.g. typin

Long integers

(Deprecated in python3.6)

Example: `99999999999999999999L`

Must end in either `l` or `L`

Can be arbitrarily long

Floating point numbers

Examples: `0., 1.0, 1e10, 3.14e-2, 6.99E4`

Implemented as C doubles

Division works normally for floating point numbers: `7./2. = 3.5`

Operations involving both floats and integers will yield floats:

`6.4 - 2 = 4.4`

Types and Operators: Operations on Numbers

Basic algebraic operations

Four arithmetic operations: `a+b, a-b, a*b, a/b`

Exponentiation: `a**b`

Other elementary functions are not part of standard Python, but included in packages like NumPy and SciP

Comparison operators

Greater than, less than, etc.: `a < b, a > b, a <= b, a >= b`

Identity tests: `a == b, a != b`

Bitwise operators

Bitwise or: `a | b`

Bitwise exclusive or: `a ^ b` # Don't confuse this with exponentiation

Bitwise and: `a & b`

Shift `a` left o

Other

Not surprisingly, Python follows the basic PEMDAS order of operations

Python supports mixed-type math. The final answer will be of the most complicated type used.

The Whitespace is so important!

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- **Use a newline to end a line of code.**
  - Use `\` when must go to next line prematurely.
- **No braces `{ }` to mark blocks of code in Python… Use *consistent* indentation instead.**
  - The first line with *less* indentation is outside of the block.
  - The first line with *more* indentation starts a nested block
- **Often a colon appears at the start of a new block. (E.g. for function and class definitions.)**

Comments

- Start comments with # – the rest of line is ignored.
- Can include a "documentation string" as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it's good style to include one.

```
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

Assignment

- ***Binding a variable*** **in Python means setting a *name* to hold a *reference* to some *object*.**
  - *Assignment creates references, not copies*

- **Names in Python do not have an intrinsic type. Objects have types.**
  - Python determines the type of the reference automatically based on the data object assigned to it.

- **You create a name the first time it appears on the left side of an assignment expression:**

```
x = 3
```

- **A reference is deleted via garbage collection after any names bound to it have passed out of scope.**

Naming Rules

- **Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.**

  ```
  bob   Bob   _bob   _2_bob_   bob_2   BoB
  ```

- **There are some reserved words:**

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

Understanding Reference Semantics

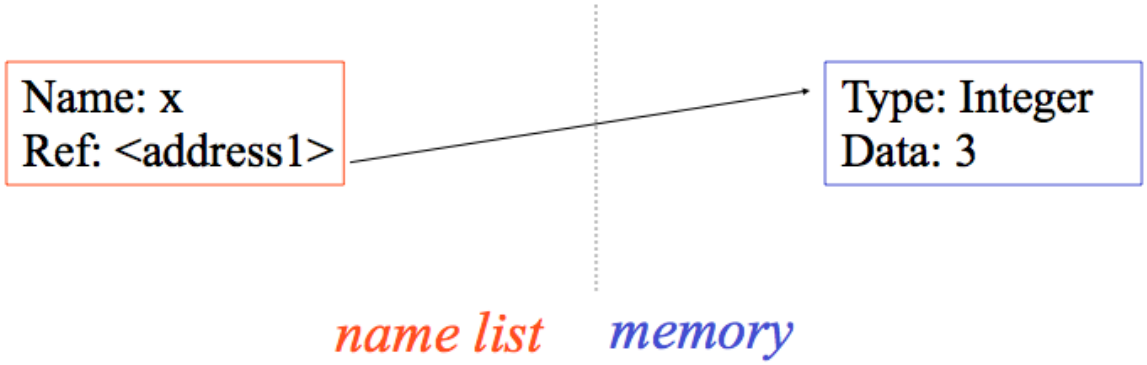
- **Assignment manipulates references**
  - —x = y **does not make a copy** of the object y references
  - —x = y makes x **reference** the object y references
- **Very useful; but beware!**
- **Example:**

```
>>> a = [1, 2, 3]    # a now references the list [1, 2, 3]
>>> b = a            # b now references what a references
>>> a.append(4)      # this changes the list a references
>>> print b          # if we print what b references,
[1, 2, 3, 4]         # SURPRISE!  It has changed…
```

Why??

Understanding Reference Semantics II

- **There is a lot going on when we type:**
  `x = 3`
- **First, an integer *3* is created and stored in memory**
- **A name *x* is created**
- **An *reference* to the memory location storing the *3* is then assigned to the name *x***
- **So: When we say that the value of *x* is *3***
- **we mean that *x* now refers to the integer *3***



Understanding Reference Semantics III

- The data 3 we created is of type integer. In Python, the datatypes integer, float, and string (and tuple) are "immutable."
- This doesn't mean we can't change the value of x, i.e. *change what x refers to* …
- For example, we could increment x:

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

Built in data types

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print x        # No effect on x, still ref 3.
3
```

- **For other data types (lists, dictionaries, user-defined types), assignment works differently.**
  - These datatypes are **"mutable."**
  - When we change these data, we do it *in place*.
  - We don't copy them into a new memory address each time.
  - If we type y=x and then modify y, both x and y are changed.

immutable

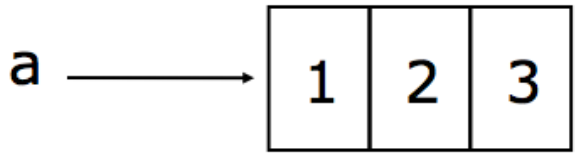
```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

mutable

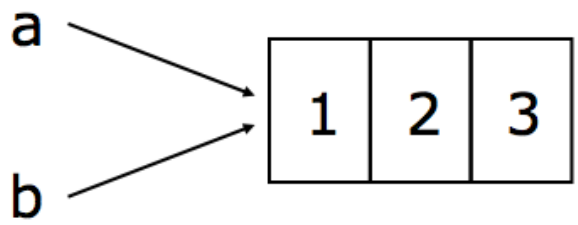
```
x = some mutable object
y = x
make a change to y
look at x
x will be changed as well
```

Why? Changing a Shared List

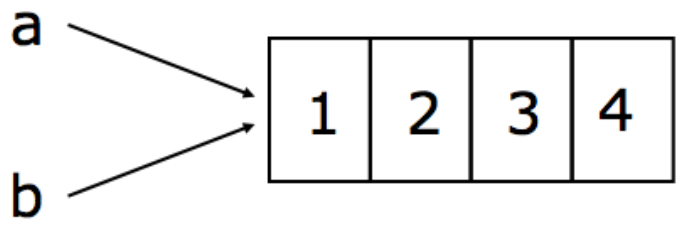
a = [1, 2, 3]



b = a



a.append(4)



Lets test our hypothesis

```
 mytest=[1,2,3]
newtest=mytest
mytest.append(4)
print(newtest)
[1, 2, 3, 4]

newtest.append(5)
print(mytest)
[1, 2, 3, 4, 5]
```

Sequence Types

1. Tuple
   - A simple ***immutable*** ordered sequence of items
   - Items can be of mixed types, including collection types

2. Strings
   - ***Immutable***
   - **Conceptually very much like a tuple**

3. List
   - ***Mutable*** ordered sequence of items of mixed types

String, List and Tuple

- **Tuples are defined using parentheses (and commas).**

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- **Lists are defined using square brackets (and commas).**

```
>>> li = ["abc", 34, 4.34, 23]
```

- **Strings are defined using quotes (", ', or """).**

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Indexing(list, tuple, String)

- **We can access individual members of a tuple, list, or string using square bracket "array" notation.**
- ***Note that all are 0 based…***

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```


```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```


Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

To make a *copy* of an entire sequence, you can use `[:]`.

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1     # 2 names refer to 1 ref
                      # Changing one affects both

>>> list2 = list1[:] # Two independent copies, two refs
```

The + Operator

- **The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.**

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

The * Operator

- **The * operator produces a *new* tuple, list, or string that "repeats" the original content.**

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> "Hello" * 3
'HelloHelloHello'
```

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
   ['abc', 45, 4.34, 23]
```

- **We can change lists *in place.***
- **Name *li* still points to the same memory reference when we're done.**
- **The mutability of lists means that they aren't as fast as tuples.**

Tuples vs. Lists

- **Lists slower but more powerful than tuples.**
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- **To convert between tuples and lists use the list() and tuple() functions:**

```
li = list(tu)
tu = tuple(li)
```

Types and Operators: Arrays (1)

Note: arrays are not a built-in python type; they are included in third-party packages such as Numeric and NumPy. However, they are very useful to computational math and physics, so I will include a discussion of them here.

Basic useage:

Loading in array capabilities: # from here on, all operations involving arrays assume you have already made this step

```
>>> from numpy import *
```

Creating an array:

```
>>> vec = array([1,2,3])
```

Creating a 3x3 matrix:

```
>>> mat = array([[1,2,3],[4,5,6],[7,8,9]])
```

If you need to initialize a dummy array whose terms will be altered later, the `zeros` and `ones` commands are useful;

`zeros((m,n),'typecode')` will create an m-by-n array of zeros, which can be integers, floats, double precision floats etc. depending on the

Arrays

Similarities between arrays and lists:

Both are mutable: both can have elements reassigned in place

Arrays and lists are indexed and sliced identically

The `len` command works just as well on arrays as anything else

Arrays and lists both have `sort` and `reverse` attributes

Differences between arrays and lists:

With arrays, the + and * signs do not refer to concatenation or repetition

Examples:

```
>>> ar1 = array([2,4,6])
>>> ar1+2 # Adding a constant to an array adds the constant to each term
[4,6,8,]   # in the array
>>> ar1*2 # Multiplying an array by a constant multiplies each term in

[4,8,12,] # the array by that constant
```

Types and Operators: Arrays (3)

More differences between arrays and lists:

Adding two arrays is just like adding two vectors

```
>>> ar1 = array([2,4,6]); ar2 = array([1,2,3])
>>> ar1+ar2
[3,6,9,]
```

Multiplying two arrays multiplies them term by term:

```
>>> ar1*ar2
[2,8,18,]
```

Same for division:

```
>>> ar1/ar2
[2,2,2,]
```

Assuming the function can take vector arguments, a function acting on an array acts on each term in the array

```
>>> ar2**2
[1,4,9,]
>>> ar3 = (pi/4)*arange(3)  # like range, but an array
>>> sin(ar3)
[ 0.         , 0.70710678, 1.         ,]
```

Basic Statements: The If Statement (1)

If statements have the following basic structure:

```
# inside the interpreter
>>> if condition:
...     action
...
>>>
```

```
# inside a script
if condition:
    action
```

Subsequent indented lines are assumed to be part of the if statement. The same is true for most other types of python statements. A statement typed into an interpreter ends once an empty line is entered, and a statement in a script ends once an unindented line appears. The same is true for defining functions.

If statements can be combined with else if (elif) and else statements as follows:

```
if condition1:      # if condition1 is true, execute action1
  action1
elif condition2:    # if condition1 is not true, but condition2 is, execute
  action2           # action2
else:               # if neither condition1 nor condition2 is true, execute
  action3           # action3
```

Basic Statements: The While Statement

While statements have the following basic structure:

```
# inside the interpreter
>>> while condition:
...     action
...
>>>
```

```
# inside a script
while condition:
    action
```

As long as the condition is true, the while statement will execute the action

Example:

```
>>> x = 1
>>> while x < 4:     # as long as x < 4...
...     print x**2   # print the square of x
...     x = x+1      # increment x by +1
...
1                    # only the squares of 1, 2, and 3 are printed,
                       because
4                    # once x = 4, the condition is false
9
```

Basic Statements: The While Statement

Pitfall to avoid:

While statements are intended to be used with changing conditions. If the condition in a while statement does not change, the program will be stuck in an infinite loop until the user hits ctrl-C.

Example:

```
>>> x = 1
>>> while x == 1:
...     print 'Hello world'
...
```

Since x does not change, Python will continue to print “Hello world” until interrupted

The For Statement

For statements have the following basic structure:

```
for item i in set s:
  action on item i
```

item and set are not statements here; they are merely intended to clarify the relationships between i and s

Example:

```
>>> for i in range(1,7):
...   print i, i**2, i**3, i**4
...
1 1 1 1
2 4 8 16
3 9 27 81
4 16 64 256
5 25 125 625
6 36 216 1296
>>>
```

Functions

Usually, function definitions have the following basic structure:

```
def func(args):
  return values.
```

Functions may be simple one-to-one mappings

```
>>> def f1(x):
...    return x*(x-1)
...
>>> f1(3)
6
```

They may contain multiple input and/or output variables

```
>>> def f2(x,y):
...    return x+y,x-y
...
>>> f2(3,2)
(5,1)
```

Functions (contd)

Functions don't need to contain arguments at all:

```
>>> def f3():
...     print 'Hello world'
...
>>> f3()
Hello world
```

The user can set arguments to default values in function definitions:

```
>>> def f4(x,a=1):
...     return a*x**2
...
>>>
```

If this function is called with only one argument, the default value of 1 is assumed for the second argument

```
>>> f4(2)
4
```

However, the user is free to change the second argument from its default value

```
>>> f4(2,a=2)    # f4(2,2) would also work
8
```

Scope

Python employs the following scoping hierarchy, in decreasing order of breadth:

Built-in (Python)

Predefined names (len, open, execfile, etc.) and types

Global (module)

Names assigned at the top level of a module, or directly in the interpreter

Names declared global in a function

Local (function)

Names assigned inside a function definition or loop

Example:

```
>>> a = 2              # a is assigned in the interpreter, so it's global
>>> def f(x):          # x is in the function's argument list, so it's local
...     y = x+a        # y is only assigned inside the function, so it's local
...     return y       # using the sa
...
>>>
```

Scope Rules

If a module file is read into the interpreter via execfile, any quantities defined in the top level of the module file will be promoted to the top level of the program

As an example: return to our friend from the beginning of the presentation, myFirstProgram.py:

```
print 'Hello world'
```

```
>>> execfile('myFirstProgram.py')
Hello world
>>> x
[0,1,2,3,4]
```

If we had imported script.py instead, the list x would not be defined on the top level. To call x, we would need to explicitly tell Python its scope, or context.

```
>>> import helloPython
Hello world
>>> helloPython.x
[0,1,2,3,4]
```

Scope Rules (contd)

You can use the same names in different scopes

Examples:

```
>>> a = 2
>>> def f5(x,y)
...     a = x+y     # this a has no knowledge of the global a, and vice-versa
...     b = x-y
...     return a**2,b**2
...
>>> a
2
```

The local a is deleted as soon as the function stops running

```
>>> x = 5
>>> import script   # same script as before
Hello world
>>> x
5
>>> script.x        # script.x and x are defined in different scopes, and
[0,1,2]             # are thus different
```

Lets Write a Test Program

Please ask computer to select a random number(between 1 to 99) and ask you to guess it. Every time you guess it tells you if you guessed high or low, until you hit bull's eye.

Here's the code

```
import random
n = random.randint(1, 99)
guess = int(input("Please enter an integer from 1 to 99: "))
while n != "Please guess":
    print
    if guess < n:
        print("Dear, Your guess is low")
        guess = int(input("Enter an integer from 1 to 99: "))
    elif guess > n:
        print("Dear, Your guess is high")
        guess = int(input("Enter an integer from 1 to 99: "))
    else:
        print("Dear, you have guessed it right...yeaahhh!")
        break
    print
```

Using dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user']
'bozo'
>>> d['pswd']
1234
>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo

>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}

>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> del d['user']           # Remove one.
>>> d
{'p':1234, 'i':34}
>>> d.clear()               # Remove all.
>>> d
{}

>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> d.keys()              # List of keys.
['user', 'p', 'i']
>>> d.values()            # List of values.
['bozo', 1234, 34]
>>> d.items()       # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Thank You!