

Python Basics

Running Interactively on UNIX

On Unix...

```
% python
```

```
>>> 3+3
```

```
6
```

- Python prompts with '>>>'.
- To exit Python (not Idle):
 - In Unix, type CONTROL-D
 - In Windows, type CONTROL-Z + <Enter>
 - Evaluate exit()

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print ('print me')
print me
>>>
```

Terminology

- **“Integrated development environment”** (IDE) is a program for writing programs: Text-editor, debugger, etc.
 - e.g. “Spyder”
 - Python IDEs are often, themselves, written in Python, i.e. they are Python packages.

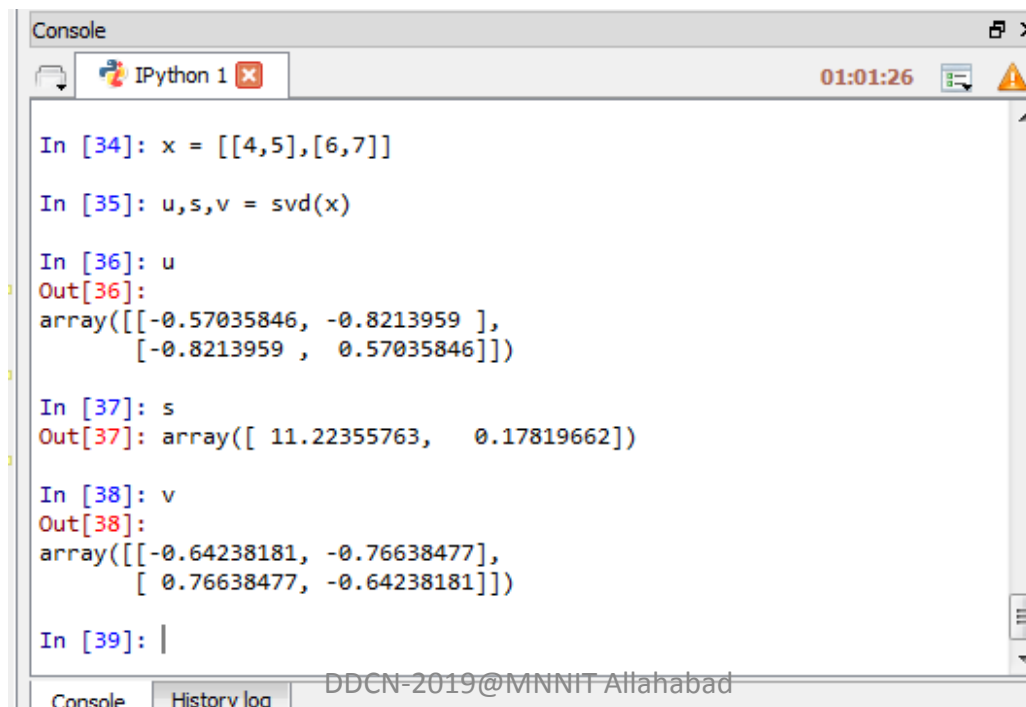
Spyder, an IDE

The screenshot displays the Spyder IDE interface with three main panels:

- Editor:** Shows a Python script named `fresnel.py`. The script defines a function `position_resolved` that calculates the Poynting vector and absorbed energy density for a multilayer structure. It also includes a `find_in_structure` function. The script is currently at line 290.
- Object Inspector:** Displays the `fresnel.fresnel_main` object. It provides a detailed description of the `fresnel_main` function, including its parameters: `pol` (light polarization), `n_list` (refractive indices), `d_list` (layer thicknesses), `th_0` (angle of incidence), and `lam_vac` (vacuum wavelength).
- Console:** Shows the output of the `pv_sim.testt()` function. The output includes various calculated values such as `ISC = 4.103 mA/cm2`, `EQE for 400-800nm = 15.8%`, and `Reflection into air = 62.5%`.

Console versus Modules

- Simplest calculations can be done directly in an interactive console. (“Consoles” are usually powered by “IPython”.)



```
Console
IPython 1 x 01:01:26

In [34]: x = [[4,5],[6,7]]

In [35]: u,s,v = svd(x)

In [36]: u
Out[36]:
array([[ -0.57035846, -0.8213959 ],
       [ -0.8213959 ,  0.57035846]])

In [37]: s
Out[37]: array([ 11.22355763,  0.17819662])

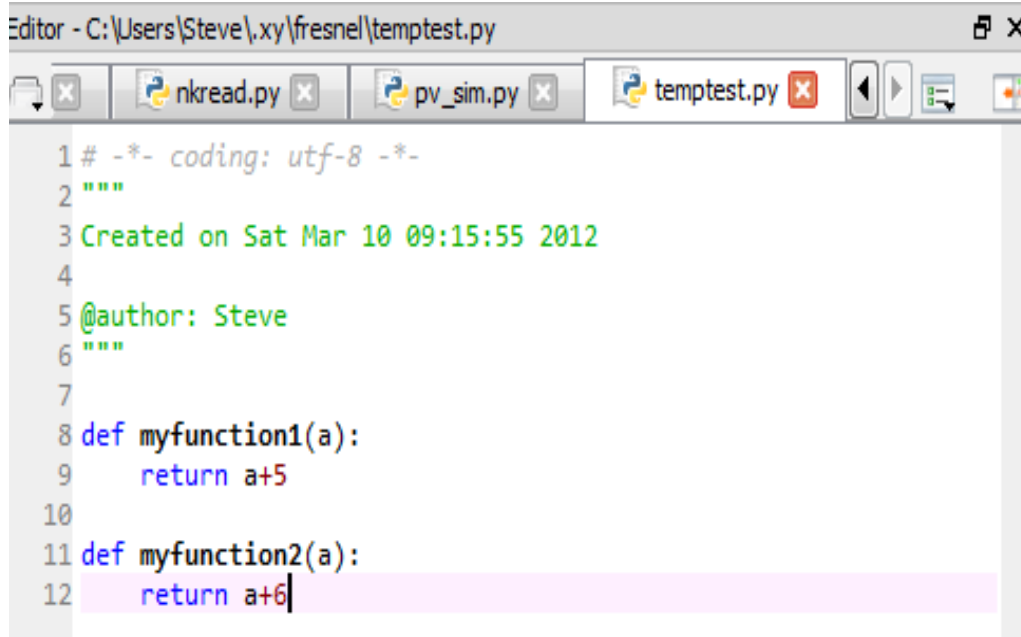
In [38]: v
Out[38]:
array([[ -0.64238181, -0.76638477],
       [  0.76638477, -0.64238181]])

In [39]: |
```

Console History log DDCN-2019@MNNIT Allahabad

Modules

- For more complicated things, you write one or more programs / functions in a “module”, “filename.py”



```
Editor - C:\Users\Steve\.xy\fresnel\temptest.py
nkread.py x pv_sim.py x temptest.py x
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Mar 10 09:15:55 2012
4
5 @author: Steve
6 """
7
8 def myfunction1(a):
9     return a+5
10
11 def myfunction2(a):
12     return a+6
```

In: `from temptest import *`

In: `myfunction1(10)`

Out: 15

In: `myfunction2(10)`

Out: 16

Modules: Imports

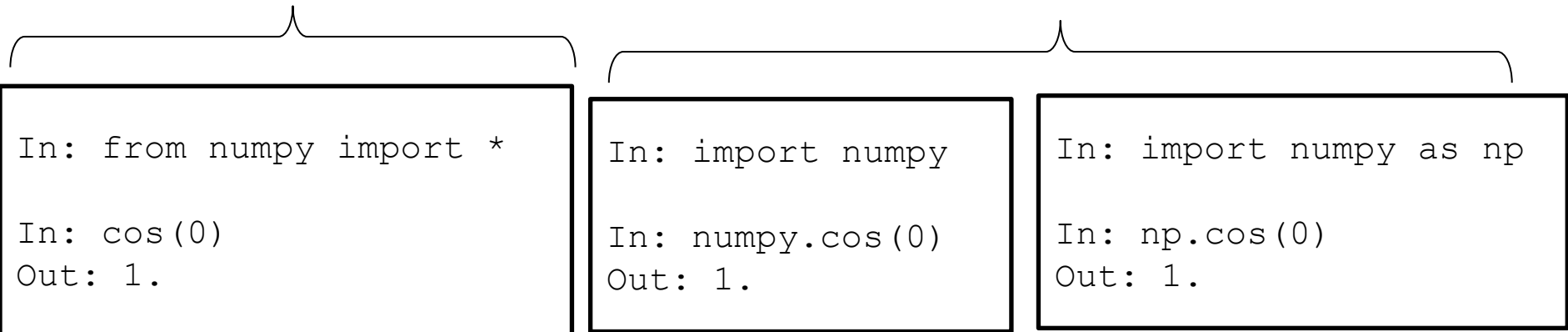
<code>import mymodule</code>	Brings all elements of mymodule in, but must refer to as mymodule.<elem>
<code>from mymodule import x</code>	Imports x from mymodule right into this namespace
<code>from mymodule import *</code>	Imports all elements of mymodule into this namespace

Modules

- Most normal math stuff like cos, conjugate, pi, etc., are actually in NumPy, not Python itself.

Can use...

Everyday use



No Braces

- Python uses **indentation** instead of braces to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This **forces** the programmer to use proper indentation since the indenting is part of the program!

Grouping Indentation

In Python:

```
for i in range(20):  
    if i%3 == 0:  
        print (i)  
    if i%5 == 0:  
        print ("Bingo!")  
print ("---")
```

In C:

```
for (i = 0; i < 20; i++)  
{  
    if (i%3 == 0) {  
        printf("%d\n", i);  
        if (i%5 == 0) {  
            printf("Bingo!\n"); }  
        }  
    printf("---\n");  
}
```

The print Statement

- Elements separated by commas print with a space between them
- A comma at the end of the statement (`print 'hello',`) will not print a newline character

```
>>> print ('hello')  
hello  
>>> print ('hello', 'there')  
hello there
```

Documentation

The '#' starts a line comment

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

Variables

- Are not declared, just assigned
- The variable is created the first time you assign it a value
- Are references to objects
- Type information is with the object, not the reference
- Everything in Python is an object

Everything is an object

- Everything means everything, including functions and classes (more on this later!)
- Data type is a property of the object and not of the variable

```
>>> x = 7
>>> x
7
>>> x = 'hello'
>>> x
'hello'
>>>
```

Numbers: Integers

- Integer – the equivalent of a C long
- Long Integer – an unbounded integer value.

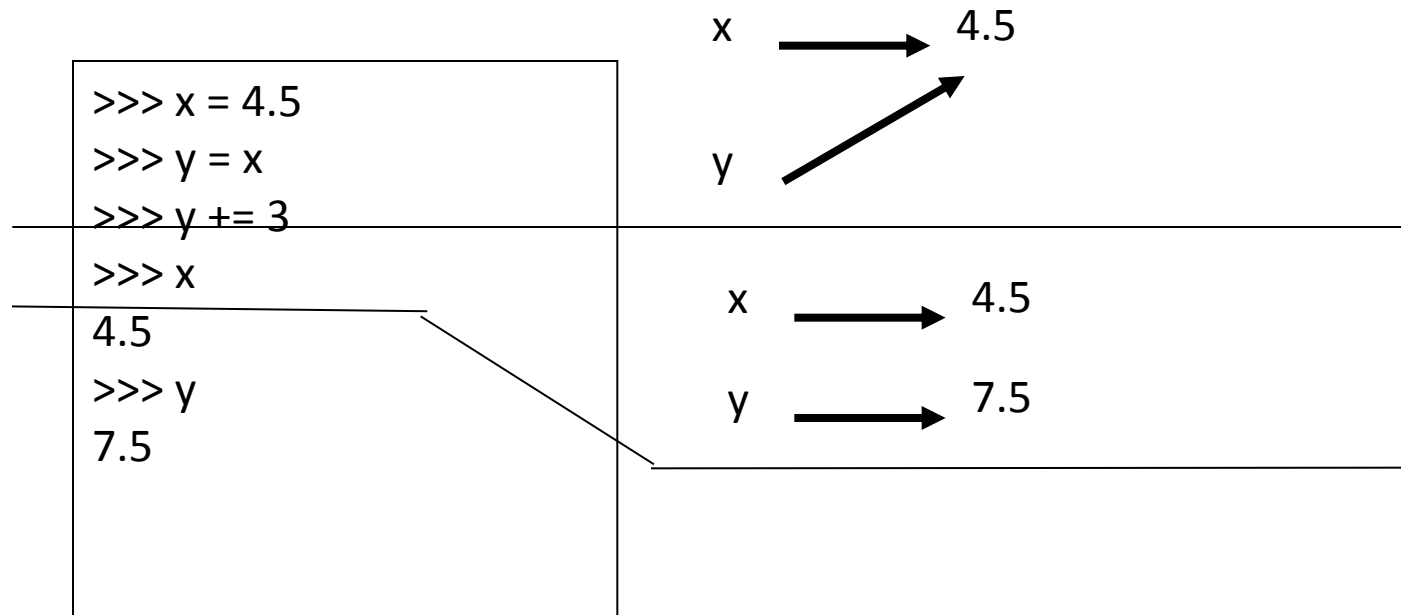
```
>>> 132224
132224
>>> 132323 ** 2
17509376329L
>>>
```


Numbers: Complex

- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

Numbers are *immutable*

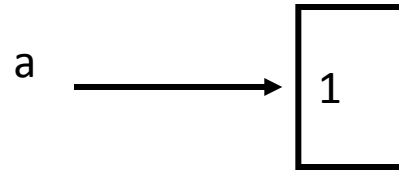


Numbers

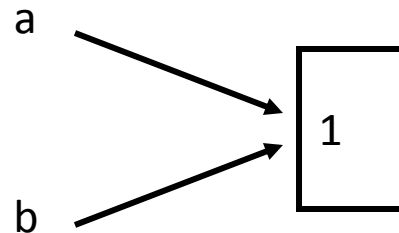
- The usual suspects
 - 12, 3.14, 0xFF, 0377, $(-1+2)^{3/4^{**5}}$, $\text{abs}(x)$, $0 < x \leq 5$
- C-style shifting & masking
 - $1 < < 16$, $x \& 0xFF$, $x | 1$, $\sim x$, x^y
- Integer division truncates :-(
 - $1/2 \rightarrow 0$ # $1./2. \rightarrow 0.5$, $\text{float}(1)/2 \rightarrow 0.5$
 - Will be fixed in the future
- Long (arbitrary precision), complex
 - $2L^{**100} \rightarrow 1267650600228229401496703205376L$
 - In Python 2.2 and beyond, 2^{**100} does the same thing
 - $1j^{**2} \rightarrow (-1+0j)$

Changing an Integer

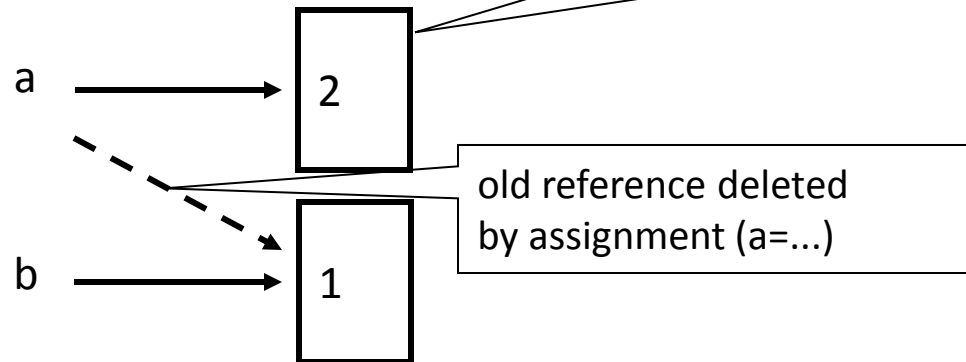
`a = 1`



`b = a`



`a = a+1`



Numbers: Floating Point

- `int(x)` converts `x` to an integer
- `float(x)` converts `x` to a floating point
- The interpreter shows a lot of digits

```
>>> 1.23232
1.232320000000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

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)
```

Enough to Understand the Code

- Indentation matters to code meaning
 - Block structure indicated by indentation
- First assignment to a variable creates it
 - Variable types don't need to be declared.
 - Python figures out the variable types on its own.
- Assignment is `=` and comparison is `==`
- For numbers `+` `-` `*` `/` `%` are as expected
 - Special use of `+` for string concatenation and `%` for string formatting (as in C's `printf`)
- Logical operators are words (`and`, `or`, `not`)
not symbols
- The basic printing command is `print`

Basic Datatypes

- Integers (default for numbers)

`z = 5 / 2 # Answer 2, integer division`

- Floats

`x = 3.456`

- Strings

- Can use `"""` or `"` to specify with `"abc" == 'abc'`
- Unmatched can occur within the string: `"matt's"`
- Use triple double-quotes for multi-line strings or strings than contain both `'` and `"` inside of them:

`"""a 'b' c"""`

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, use *consistent* indentation instead

- First line with *less* indentation is outside of the block
- First line with *more* indentation starts a nested block

- Colons start of a new block in many constructs, e.g. function definitions, then clauses

Comments

- Start comments with `#`, rest of line is ignored
- Can include a “documentation string” as the first line of a new function or class you define
- Development environments, debugger, and other tools use it: it’s good style to include one

```
def fact(n) :  
    """fact(n) assumes n is a positive  
    integer and returns factorial of n."""  
    assert(n>0)  
    return 1 if n==1 else n*fact(n-1)
```

Variables

- No need to declare
- Need to assign (initialize)
 - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```
- ***Everything*** is a "variable":
 - Even functions, classes, modules

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

Naming conventions

The Python community has these recommended naming conventions

- `joined_lower` for functions, methods and, attributes
- `joined_lower` or `ALL_CAPS` for constants
- `StudlyCaps` for classes
- `camelCase` only to conform to pre-existing conventions
- Attributes: `interface`, `_internal`, `__private`

Accessing Non-Existent Name

Accessing a name before it's been properly created (by placing it on the left side of an assignment), raises an error

```
>>> y
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#16>", line 1, in -toplevel-
```

```
    y
```

```
NameError: name 'y' is not defined
```

```
>>> y = 3
```

```
>>> y
```

```
3
```

String Literals

- Strings are *immutable*
- There is no char type like in C++ or Java
- + is overloaded to do concatenation

```
>>> x = 'hello'  
>>> x = x + ' there'  
>>> x  
'hello there'
```

String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the others :)'
>>> print s
And me too!
though I am much longer
than the others :)'
```


String Formatting

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Strings

- "hello"+"world" "helloworld" # concatenation
- "hello"*3 "hellohellohello" # repetition
- "hello"[0] "h" # indexing
- "hello"[-1] "o" # (from end)
- "hello"[1:4] "ello" # slicing
- len("hello") 5 # size
- "hello" < "jello" 1 # comparison
- "e" in "hello" 1 # search
- "escapes: \n etc, \033 etc, \if etc"
- 'single quotes' ""triple quotes"" r"raw strings"

Substrings and Methods

```
>>> s = '012345'  
>>> s[3]  
'3'  
>>> s[1:4]  
'123'  
>>> s[2:]  
'2345'  
>>> s[:4]  
'0123'  
>>> s[-2]  
'4'
```

- **len**(String) – returns the number of characters in the String
- **str**(Object) – returns a String representation of the Object

```
>>> len(x)  
6  
>>> str(10.3)  
'10.3'
```

Booleans

- 0 and None are false
- Everything else is true
- True and False are aliases for 1 and 0 respectively

Boolean Expressions

- Compound boolean expressions short circuit
- and and or return one of the elements in the expression
- Note that when None is returned the interpreter does not print anything

```
>>> True and False
False
>>> False or True
True
>>> 7 and 14
14
>>> None and 2
>>> None or 2
2
```

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1, 'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists

- Flexible arrays, *not* Lisp-like linked lists
 - `a = [99, "bottles of beer", ["on", "the", "wall"]]`
- Same operators as for strings
 - `a+b`, `a*3`, `a[0]`, `a[-1]`, `a[1:]`, `len(a)`
- Item and slice assignment
 - `a[0] = 98`
 - `a[1:2] = ["bottles", "of", "beer"]`
 `-> [98, "bottles", "of", "beer", ["on", "the", "wall"]]`
 - `del a[-1]` `# -> [98, "bottles", "of", "beer"]`

Lists are 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.

Tuples are 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')
```

- *The immutability of tuples means they're faster than lists.*

Lists: Modifying Content

- **x[i] = a** reassigns the *i*th element to the value *a*
- Since *x* and *y* point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Operations on Lists Only

```
>>> li = [1, 11, 3, 4, 5]
```

```
>>> li.append('a')    # Note the  
    method syntax
```

```
>>> li  
[1, 11, 3, 4, 5, 'a']
```

```
>>> li.insert(2, 'i')  
>>> li  
[1, 11, 'i', 3, 4, 5, 'a']
```

The *extend* method vs *+*

- *+* creates a fresh list with a new memory ref
- *extend* operates on list `li` in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- *Potentially confusing*:
 - *extend* takes a list as an argument.
 - *append* takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only

Lists have many methods, including index, count, remove, reverse, sort

```
>>> li = ['a', 'b', 'c', 'b']
```

```
>>> li.index('b')    # index of 1st  
occurrence
```

1

```
>>> li.count('b')    # number of  
occurrences
```

2

```
>>> li.remove('b')   # remove 1st  
occurrence
```

```
>>> li  
['a', 'c', 'b']
```

Operations on Lists Only

```
>>> li = [5, 2, 6, 8]
```

```
>>> li.reverse()      # reverse the list *in place*
```

```
>>> li
[8, 6, 2, 5]
```

```
>>> li.sort()         # sort the list *in place*
```

```
>>> li
[2, 5, 6, 8]
```

```
>>> li.sort(some_function)
# sort in place using user-defined comparison
```


More List Operations

```
>>> a = range(5)           # [0,1,2,3,4]
>>> a.append(5)            # [0,1,2,3,4,5]
>>> a.pop()                # [0,1,2,3,4]
5
>>> a.insert(0, 42)        # [42,0,1,2,3,4]
>>> a.pop(0)               # [0,1,2,3,4]
5.5
>>> a.reverse()            # [4,3,2,1,0]
>>> a.sort()               # [0,1,2,3,4]
```

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:
‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```



Tuple details

- The comma is the tuple creation operator, not parens

```
>>> 1,  
(1,)
```

- Python shows parens for clarity (best practice)

```
>>> (1,)   
(1,)
```

- Don't forget the comma!

```
>>> (1)   
1
```

- Trailing comma only required for singletons
others

- Empty tuples have a special syntactic form

```
>>> ()   
()  
>>> tuple()  
()
```

Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Summary: Tuples vs. Lists

- Lists slower but more powerful than tuples
 - Lists can be modified, and they have lots of handy operations and methods
 - 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)
```

Sequence Types

1. Tuple: ('john', 32, [CMSC])

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

2. Strings: "John Smith"

- *Immutable*
- Conceptually very much like a tuple

3. List: [1, 2, 'john', ('up', 'down')]

- *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
 - Tuples and strings are *immutable*
 - Lists are *mutable*
- The operations shown in this section can be applied to *all* sequence types
 - most examples will just show the operation performed on one

Sequence Types 1

- Define tuples using parentheses and commas

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

- Define lists are using square brackets and commas

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

- Define strings using quotes ("', or """).

```
>>> st = "Hello World"
```

```
>>> st = 'Hello World'
```

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

Sequence Types 2

- 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 index: count from right, starting with -1

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

Slicing: return copy of a subset

```
>>> 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 second.

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

Negative indices count from end

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

Slicing: return copy of a =subset

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

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

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

Omit second index to make copy starting at first index and going to end

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

Copying the Whole Sequence

- `[:]` makes a *copy* of an entire sequence

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

- Note the difference between these two lines for mutable sequences

```
>>> l2 = l1 # Both refer to 1 ref,  
            # changing one affects  
            both
```

```
>>> l2 = l1[ : ] # Independent copies,  
                two refs
```

The 'in' Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*

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'
```

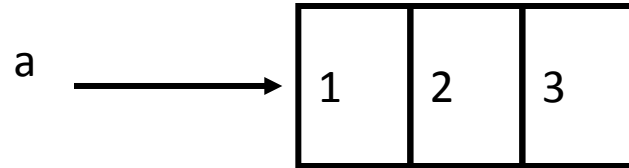
Reference Semantics

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

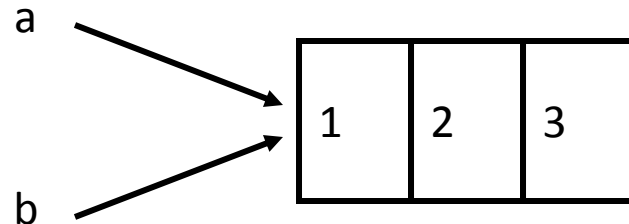
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Changing a Shared List

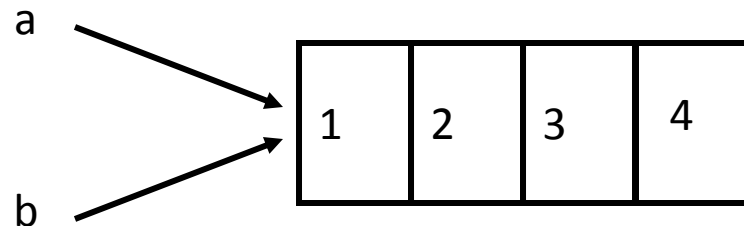
`a = [1, 2, 3]`



`b = a`



`a.append(4)`



Dictionaries

- Hash tables, "associative arrays"
 - `d = {"duck": "eend", "water": "water"}`
- Lookup:
 - `d["duck"] -> "eend"`
 - `d["back"]` # raises `KeyError` exception
- Delete, insert, overwrite:
 - `del d["water"]` # `{"duck": "eend", "back": "rug"}`
 - `d["back"] = "rug"` # `{"duck": "eend", "back": "rug"}`
 - `d["duck"] = "duik"` # `{"duck": "duik", "back": "rug"}`

More Dictionary Ops

- Keys, values, items:
 - `d.keys()` -> ["duck", "back"]
 - `d.values()` -> ["duik", "rug"]
 - `d.items()` -> [("duck","duik"), ("back","rug")]
- Presence check:
 - `d.has_key("duck")` -> 1; `d.has_key("spam")` -> 0
- Values of any type; keys almost any
 - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

Dictionary Details

- Keys must be **immutable**:
 - numbers, strings, tuples of immutables
 - these cannot be changed after creation
 - reason is *hashing* (fast lookup technique)
 - **not** lists or other dictionaries
 - these types of objects can be changed "in place"
 - no restrictions on values
- Keys will be listed in **arbitrary order**
 - again, because of hashing

Tuples

- `key = (lastname, firstname)`
- `point = x, y, z` # parentheses optional
- `x, y, z = point` # unpack
- `lastname = key[0]`
- `singleton = (1,)` # trailing comma!!!
- `empty = ()` # parentheses!
- tuples vs. lists; tuples immutable

Data Type Summary

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references

Data Type Summary

- Integers: 2323, 3234L
- Floating Point: 32.3, 3.1E2
- Complex: 3 + 2j, 1j
- Lists: l = [1,2,3]
- Tuples: t = (1,2,3)
- Dictionaries: d = {'hello' : 'there', 2 : 15}

Control Structures

if condition:

statements

[elif condition:

statements] ...

else:

statements

while condition:

statements

for var in sequence:

statements

break

continue

Python programming: if

```
if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

If Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30 :
    y = x + 30
else :
    y = x
Print( 'y = ',)
print (math.sin(y) )
```

```
>>> import ifstatement
y = 0.999911860107
>>>
```

In interpreter

In file ifstatement.py

For Loops

- Similar to perl for loops, iterating through a list of values

forloop1.py

```
for x in [1,7,13,2] :  
    print( x)
```

```
~: python forloop1.py  
1  
7  
13  
2
```

forloop2.py

```
for x in range(5) :  
    print (x)
```

```
~: python forloop2.py  
0  
1  
2  
3  
4
```

range(N) generates a list of numbers [0,1, ..., n-1]

For Loops

- **For** loops also may have the optional **else** clause

```
for x in range(5):  
    print( x)  
    break  
else :  
    print ('i got here')
```

```
~: python elseforloop.py  
1
```

elseforloop.py

Python programming: for, range

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

There are “continue” and “break” commands for for loops too.

Note: take print(x) instead of print x as we are using python3

```
In [72]: for i in range(5):
.....:     print i
.....:
0
1
2
3
4
```

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

While Loops

```
x = 1
while x < 10 :
    print (x)
    x = x + 1
```

In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print (x)
    x = x + 1
else:
    print( 'hello')
```

```
~: python whileelse.py
1
2
hello
```

Run from the command line

In whileelse.py

The Loop Else Clause

```
x = 1
while x < 5 :
    print( x)
    x = x + 1
    break
else :
    print ('i got here')
```

```
~: python whileelse2.py
1
```

whileelse2.py

Python programming: “White space”

- For “if”, “for”, “def”, “else”, etc. [commands ending in ‘:’], the associated code is whatever is indented afterwards.

```
for n in range(2, 10):  
    for x in range(2, n):  
        if n % x == 0:  
            print n, 'equals', x, '*', n/x  
            break  
    else:  
        # loop fell through without finding a factor  
        print n, 'is a prime number'
```

Note: take
print(n)

“else” goes with
“for” not “if”.

Python programming: “White space”

- The end of a line is the end of a command.
 - For longer commands, use parentheses...when there's dangling parentheses / brackets, Python assumes the command is continuing to the next line. [Alternative: End a line with a backslash.]

```
292     #Poynting vector
293     if(pol=='s'):
294         poyn = ((n*cos(th)*conj(Ef+Eb)*(Ef-Eb)).real) / (n_0*cos(th_0)).real
295     elif(pol=='p'):
296         poyn = (((n*conj(cos(th))*(Ef+Eb)*conj(Ef-Eb)).real)
297                 / (n_0*conj(cos(th_0))).real)
298
299     #absorbed energy density
300     if(pol=='s'):
301         absor = (n*cos(th)*kz*abs(Ef+Eb)**2).imag / (n_0*cos(th_0)).real
302     elif(pol=='p'):
303         absor = (n*conj(cos(th))*
304                 (kz*abs(Ef-Eb)**2-conj(kz)*abs(Ef+Eb)**2)
305                 ).imag / (n_0*conj(cos(th_0))).real
```

Importing data – an example.

[File has unknown number of header rows.]

“csv” is a standard Python package for reading data-files.

```
23 import csv
24 import numpy as np
25
26 def readcomsolfile(filename):
27     """
28     filename is 2D COMSOL output file, with header rows then data of the form:
29
30     x    y    z
31
32     This function skips header rows then puts the data into a NumPy array of
33     floats.
34     """
35     A=[]
36     myreader = csv.reader(open(filename, 'r'), delimiter=' ',
37                            skipinitialspace=True)
38     for row in myreader:
39         rowfloat = []
40         try:
41             for elem in row:
42                 rowfloat.append(float(elem))
43             A.append(rowfloat)
44         except (TypeError, ValueError): #Header row -- skip it.
45             pass
46     A = np.asarray(A)
47     return A
```

Good practice: Start each function with a block-quote describing it.

Consecutive spaces are treated as just one separator.

Convert each item to a real number and put it in a list, then append the list as a new row in “A”.

If you can’t convert to a real number, then it’s a header row. Don’t do anything.

Convert “A” from a list-of-lists to a NumPy 2D array.

Interactive “Shell”

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations: IDLE (GUI),
python (command line)
- Type statements or expressions at prompt:

```
>>> print( "Hello, world")
```

```
Hello, world
```

```
>>> x = 12**2
```

```
>>> x/2
```

```
72
```

```
>>> # this is a comment
```

Warning: Integer division

```
In: 7/3
```

```
Out: 2
```

```
In: 7./3
```

```
Out: 2.3333333
```

```
In: 7/3.
```

```
Out: 2.3333333
```


Function Basics

```
def max(x,y) :  
    if x < y :  
        return x  
    else :  
        return y
```

functionbasics.py

```
>>> import functionbasics  
>>> max(3,5)  
5  
>>> max('hello', 'there')  
'there'  
>>> max(3, 'hello')  
'hello'
```

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
>>> lst[1](4)
16
```

Functions, Procedures

```
def name(arg1, arg2, ...):  
    """documentation"""           # optional doc string  
    statements  
  
    return                          # from procedure  
    return expression             # from function
```

Define a function

- Define a function

```
def f(a,b):  
    c = a * b  
    return abs(c**2)
```

- Use a function

```
x = f(3,5)
```

Example Function

```
def gcd(a, b):  
    "greatest common divisor"  
    while a != 0:  
        a, b = b%a, a    # parallel assignment  
    return b
```

```
>>> gcd.__doc__  
'greatest common divisor'  
>>> gcd(12, 20)  
4
```

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print ('hello')
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Functions as Parameters

```
def foo(f, a) :  
    return f(a)  
  
def bar(x) :  
    return x * x
```

funcasparam.py

```
>>> from funcasparam import *  
>>> foo(bar, 3)  
9
```

Note that the function **foo** takes two parameters and applies the first as a function with the second as its parameter

Higher-Order Functions

map(func,seq) – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x):  
    return 2*x
```

highorder.py

```
>>> from highorder import *  
>>> lst = range(10)  
>>> lst  
[0,1,2,3,4,5,6,7,8,9]  
>>> map(double,lst)  
[0,2,4,6,8,10,12,14,16,18]
```

Higher-Order Functions

filter(boolfunc,seq) – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x):  
    return ((x%2 == 0))
```

highorder.py

```
>>> from highorder import *  
>>> lst = range(10)  
>>> lst  
[0,1,2,3,4,5,6,7,8,9]  
>>> filter(even,lst)  
[0,2,4,6,8]
```

Higher-Order Functions

reduce(func,seq) – applies func to the items of seq, from left to right, two-at-time, to reduce the seq to a single value.

```
def plus(x,y):  
    return (x + y)
```

highorder.py

```
>>> from highorder import *  
>>> lst = ['h','e','l','l','o']  
>>> reduce(plus,lst)  
'hello'
```

Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :  
    def bar (z) :  
        return z * 2  
    return bar(x) + y
```

```
>>> from funcinfunc import *  
>>> foo(2,3)  
7
```

funcinfunc.py

Functions Returning Functions

```
def foo (x) :  
    def bar(y) :  
        return x + y  
    return bar  
# main  
f = foo(3)  
print (f)  
print (f(2))
```

```
~: python funcreturnfunc.py  
<function bar at 0x612b0>  
5
```

funcreturnfunc.py

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn't limit the type of a parameter

```
>>> def foo(x = 3) :  
...     print x  
...  
>>> foo()  
3  
>>> foo(10)  
10  
>>> foo('hello')  
hello
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :  
...     print (a, b, c)  
...  
>>> foo(c = 10, a = 2, b = 14)  
2 14 10  
>>> foo(3, c = 2, b = 19)  
3 19 2
```

Warning: Array copying

```
In: a=array([[1,2],[3,4]])  
  
In: b=a  
  
In: b[0,0] = 100  
  
In: a  
Out: array([[100,2],[3,4]])
```

The behavior makes sense if you think of “a” as *NOT* a list of numbers but *INSTEAD* as a description of where I should look, in the computer’s RAM, to find a certain list of numbers.

In the bottom-left example, b is a “view” of the data in a.

```
In: a=array([[1,2],[3,4]])  
  
In: b=a[:,0]  
  
In: b  
Out: array([1, 3])  
  
In: b[0] = 100  
  
In: a  
Out: array([[100,2],[3,4]])
```

FIXED

```
In: a=array([[1,2],[3,4]])  
  
In: b=a.copy()  
  
In: b[0,0] = 100  
  
In: a  
Out: array([[1,2],[3,4]])
```


Warning: Arrays in functions

```
def messwitharray(a):  
    a[0] = 57  
    return a[1]+a[2]
```

```
In: a = array([1,2,3])
```

```
In: b = messwitharray(a)
```

```
In: b  
Out: 5
```

```
In: a  
Out: array([57,2,3])
```

Solution: Put `a2=a.copy()` at the start of the function, then you can freely mess around with `a2`.

FIXED

```
def messwitharray(a_temp):  
    a = a_temp.copy()  
    a[0] = 57  
    return a[1]+a[2]
```

.....OR.....

```
In: b = messwitharray(a.copy())
```

The behavior makes sense if you think of “a” as *NOT* a list of numbers but *INSTEAD* as a description of where I should look, in the computer’s RAM, to find a certain list of numbers.

When in doubt, `copy()` !!

```
#define an array  
a = array([[1,2],[3,4],[5,6]])  
  
#pull out the first two rows  
b = a[0:2].copy()  
  
#also need the transpose  
c = b.T.copy()  
  
#run a function  
d = f(b.copy(), c.copy())
```

You can always take
them out later on!

The exact same warnings and suggestions apply to any
“mutable object”, including built-in python arrays.

Define a function with multiple return values. A few options:

Return a “Python list”

```
import numpy as np

def polar(z):
    phi = np.angle(z)
    abs_val = abs(z)
    return [phi, abs_val]

[t, r] = polar(4+8j)
```

Return a “Python tuple”

```
import numpy as np

def polar(z):
    phi = np.angle(z)
    abs_val = abs(z)
    return (phi, abs_val)

t, r = polar(4+8j)
```

```
import numpy as np

def polar(z):
    phi = np.angle(z)
    abs_val = abs(z)
    return {'angle':phi, 'abs':abs_val}

results = polar(4+8j)
t = results['angle']
r = results['abs']
```

My favorite: Return a “Python dictionary”. Code is easier to understand and less prone to error.

[Even fancier options: Return an “object” in a custom “class”; return a “named tuple”]

Classes

`class name:`

`"documentation"`

`statements`

-or-

`class name(base1, base2, ...):`

`...`

Most, *statements* are method definitions:

`def name(self, arg1, arg2, ...):`

`...`

May also be *class variable* assignments

Example Class

```
class Stack:
    "A well-known data structure..."
    def __init__(self):                # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)          # the sky is the limit
    def pop(self):
        x = self.items[-1]             # what happens if it's empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0    # Boolean result
```

Using Classes

- To create an instance, simply call the class object:

```
x = Stack() # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty() # -> 1
```

```
x.push(1) # [1]
```

```
x.empty() # -> 0
```

```
x.push("hello") # [1, "hello"]
```

```
x.pop() # -> "hello" # [1]
```

- To inspect instance variables, use dot notation:

```
x.items # -> [1]
```

Subclassing

```
class FancyStack(Stack):  
    "stack with added ability to inspect inferior stack items"  
  
    def peek(self, n):  
        "peek(0) returns top; peek(-1) returns item below that; etc."  
        size = len(self.items)  
        assert 0 <= n < size                # test precondition  
        return self.items[size-1-n]
```

Subclassing (2)

```
class LimitedStack(FancyStack):  
    "fancy stack with limit on stack size"  
  
    def __init__(self, limit):  
        self.limit = limit  
        FancyStack.__init__(self)           # base class constructor  
  
    def push(self, x):  
        assert len(self.items) < self.limit  
        FancyStack.push(self, x)           # "super" method call
```


Class / Instance Variables

```
class Connection:
```

```
    verbose = 0                                # class variable
```

```
    def __init__(self, host):
```

```
        self.host = host                        # instance variable
```

```
    def debug(self, v):
```

```
        self.verbose = v                        # make instance variable!
```

```
    def connect(self):
```

```
        if self.verbose:                        # class or instance variable?
```

```
            print "connecting to", self.host
```

Instance Variable Rules

- On use via instance (`self.x`), search order:
 - (1) instance, (2) class, (3) base classes
 - this also works for method lookup
- On assignment via instance (`self.x = ...`):
 - always makes an instance variable
- Class variables "default" for instance variables
- But...!
 - mutable *class* variable: one copy *shared* by all
 - mutable *instance* variable: each instance its own

Modules

- Collection of stuff in *foo.py* file
 - functions, classes, variables
- Importing modules:
 - `import re; print (re.match("[a-z]+", s))`
 - `from re import match; print (match("[a-z]+", s))`
- Import with rename:
 - `import re as regex`
 - `from re import match as m`
 - Before Python 2.0:
 - `import re; regex = re; del re`

Catching Exceptions

```
def foo(x):  
    return 1/x
```

```
def bar(x):  
    try:  
        print( foo(x))  
    except ZeroDivisionError, message:  
        print ("Can't divide by zero:", message)
```

```
bar(0)
```

Try-finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()      # always executed
    print ("OK")   # executed on success only
```

Raising Exceptions

- `raise IndexError`
- `raise IndexError("k out of range")`
- `raise IndexError, "k out of range"`
- `try:`
 something
`except: # catch everything`
 `print ("Oops")`
 `raise # reraise`

More on Exceptions

- User-defined exceptions
 - subclass Exception or any other standard exception
- Old Python: exceptions can be strings
 - WATCH OUT: compared by object identity, not ==
- Last caught exception info:
 - `sys.exc_info() == (exc_type, exc_value, exc_traceback)`
- Last uncaught exception (traceback printed):
 - `sys.last_type, sys.last_value, sys.last_traceback`
- Printing exceptions: traceback module

File Objects

- `f = open(filename[, mode[, buffersize]])`
 - mode can be "r", "w", "a" (like C stdio); default "r"
 - append "b" for text translation mode
 - append "+" for read/write open
 - buffersize: 0=unbuffered; 1=line-buffered; buffered
- methods:
 - `read([nbytes])`, `readline()`, `readlines()`
 - `write(string)`, `writelines(list)`
 - `seek(pos[, how])`, `tell()`
 - `flush()`, `close()`
 - `fileno()`

Standard Library

- Core:
 - os, sys, string, getopt, StringIO, struct, pickle, ...
- Regular expressions:
 - re module; Perl-5 style patterns and matching rules
- Internet:
 - socket, rfc822, httpplib, htmlplib, ftplib, smtplib, ...
- Miscellaneous:
 - pdb (debugger), profile+pstats
 - Tkinter (Tcl/Tk interface), audio, *dbm, ...

Running Programs on UNIX

- Call python program via the python interpreter

```
% python fact.py
```

- Make a python file directly executable by
 - Adding the appropriate path to your python interpreter as the first line of your file

```
#!/usr/bin/python
```

- Making the file executable

```
% chmod a+x fact.py
```

- Invoking file from Unix command line

```
% fact.py
```

Example 'script': fact.py

```
#!/usr/bin/python
```

```
def fact(x):
```

```
    """Returns the factorial of its argument, assumed to be a posint"""
```

```
    if x == 0:
```

```
        return 1
```

```
    return x * fact(x - 1)
```

```
Print()
```

```
print ('N fact(N)')
```

```
print ("-----")
```

```
for n in range(10):
```

```
    print (n, fact(n))
```

Python Scripts

- When you call a python program from the command line the interpreter evaluates each expression in the file
- Familiar mechanisms are used to provide command line arguments and/or redirect input and output
- Python also has mechanisms to allow a python program to act both as a script and as a module to be imported and used by another python program

Example of a Script

```
#!/usr/bin/python
""" reads text from standard input and outputs any
    email
    addresses it finds, one to a line.
"""

import re
from sys import stdin

# a regular expression ~ for a valid email address
pat = re.compile(r'[-\w][-\.\w]* @[-\w][-\w.]+[a-zA-Z]{2,4}')

for line in stdin.readlines():
    for address in pat.findall(line):
        print address
```

Getting a unique, sorted list

```
import re
from sys import stdin

pat = re.compile(r'[-\w][-\w]* @[-\w][-\w.]+[a-zA-Z]{2,4}')

# found is an initially empty set (a list w/o duplicates)
found = set( )
for line in stdin.readlines():
    for address in pat.findall(line):
        found.add(address)

# sorted() takes a sequence, returns a sorted list of its elements
for address in sorted(found):
    print address
```

Simple functions: ex.py

```
"""factorial done recursively and iteratively"""
```

```
def fact1(n):  
    ans = 1  
    for i in range(2, n):  
        ans = ans * i  
    return ans
```

```
def fact2(n):  
    if n < 1:  
        return 1  
    else:  
        return n * fact2(n - 1)
```

Simple functions: ex.py

```
671> python
```

```
Python 2.5.2 ...
```

```
>>> import ex
```

```
>>> ex.fact1(6)
```

```
1296
```

```
>>> ex.fact2(200)
```

```
78865786736479050355236321393218507...000000L
```

```
>>> ex.fact1
```

```
<function fact1 at 0x902470>
```

```
>>> fact1
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'fact1' is not defined
```

```
>>>
```


Input

- The **input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Input: Example

```
print ("What's your name?")
name = input("> ")

print ("What year were you born?")
birthyear = int(input("> "))

print ("Hi %s! You are %d years old!" % (name, 2011 - birthyear))
```

```
~: python input.py
What's your name?
> Michael
What year were you born?
>1980
Hi Michael! You are 31 years old!
```

Files: Input

<code>inflobj = open('data', 'r')</code>	Open the file 'data' for input
<code>S = inflobj.read()</code>	Read whole file into one String
<code>S = inflobj.read(N)</code>	Reads N bytes ($N \geq 1$)
<code>L = inflobj.readlines()</code>	Returns a list of line strings

Files: Output

<code>outflobj = open('data', 'w')</code>	Open the file 'data' for writing
<code>outflobj.write(S)</code>	Writes the string S to file
<code>outflobj.writelines(L)</code>	Writes each of the strings in list L to file
<code>outflobj.close()</code>	Closes the file

Moving to Files

- The interpreter is a good place to try out some code, but what you type is not reusable
- Python code files can be read into the interpreter using the **import** statement

Moving to Files

- In order to be able to find a module called `myscripts.py`, the interpreter scans the list `sys.path` of directory names.
- The module must be in one of those directories.

```
>>> import sys
>>> sys.path
['C:\\Python26\\Lib\\idlelib', 'C:\\WINDOWS\\system32\\python26.zip',
'C:\\Python26\\DLLs', 'C:\\Python26\\lib', 'C:\\Python26\\lib\\plat-win',
'C:\\Python26\\lib\\lib-tk', 'C:\\Python26', 'C:\\Python26\\lib\\site-
packages']
>>> import myscripts
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    import myscripts.py
```

ImportError: No module named `myscripts.py`