PYTHON PROGRAMMING FOR DATA SCIENCE

Anasua Sarkar
Computer Science & Engineering Department
Jadavpur University
anasua.sarkar@jadavpuruniversity.in

Data Science – A Definition

Data Science is the science which uses computer science, statistics and machine learning, visualization and human-computer interactions to collect, clean, integrate, analyze, visualize, interact with data to create data products.

Python : Introduction

- Most recent popular (scripting/extension) language
  - although origin ~1991
- heritage: teaching language (ABC)
  - Tcl: shell
  - perl: string (regex) processing
- object-oriented
- It includes modules for creating graphical user interfaces, connecting to relational databases, generating pseudorandom numbers, arithmetic with arbitrary-precision decimals, manipulating regular expressions, and unit testing.
- Large organizations that use Python include Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram and Spotify. The social news networking site Reddit is written entirely in Python.

What's Python?

- Python is a general-purpose, interpreted high-level programming language.
- Its syntax is clear and emphasize readability.
- Python has a large and comprehensive standard library.
- Python supports multiple programming paradigms, primarily but not limited to object-oriented, imperative and, to a lesser extent, functional programming styles.
- It features a fully dynamic type system and automatic memory management

@Yang Li

Timeline

- Python born, name picked - Dec 1989
  - By Guido van Rossum, now at GOOGLE
- First public release (USENET) - Feb 1991
- python.org website - 1996 or 1997
- 2.0 released - 2000
- Python Software Foundation - 2001
- …
- 2.4 released - 2004
- 2.5 released – 2006
- 3.0 released - 2008
- Current versions: Python 2.7.13 and Python 3.6.0.

It’s Named After Monty Python

- Despite all the reptile icons in the Python world, the truth is that Python creator Guido van Rossum named it after the BBC comedy series Monty Python’s Flying Circus. He is a big fan of Monty Python, as are many software developers (indeed, there seems to almost be a symmetry between the two fields).
- This legacy inevitably adds a humorous quality to Python code examples. For instance, the traditional “foo” and “bar” for generic variable names become “spam” and “eggs” in the Python world. The occasional “Brian,” “ni,” and “shrubbery” likewise owe their appearances to this namesake. It even impacts the Python community at large: talks at Python conferences are regularly billed as “The Spanish Inquisition.”
- All of this is, of course, very funny if you are familiar with the show, but less so otherwise.

Key features of Python

- Python programs can run on any platform, you can carry code created in Windows machine and run it on Mac or Linux.
- Python has inbuilt large library with prebuilt and portable functionality, also known as the standard library.
- Python is an expressive language.
- Python is free and open source.
- Python code is about one third of the size of equivalent C++ and Java code.
- Python can be both dynamically and strongly typed—
  - Dynamically typed means it is a type of variable that is interpreted at runtime, which means, in Python, there is no need to define the type (int or float) of the variable.

What is python?

- Object oriented language
- Interpreted language
- Supports dynamic data type
- Independent from platforms
- Focused on development time
- Simple and easy grammar
- High-level internal object data types
- Automatic memory management
- It's free (open source)!

Advantages of Python

- Simple
- Easy to study
- Free and open source
- High-level programming language
- Portability
- Expansibility
- Embedability
- Large and comprehensive standard libraries
- Canonical code

High-level data types

- Numbers: int, long, float, complex
- Strings: immutable
- Lists and dictionaries: containers
- Other types for e.g. binary data, regular expressions
- Extension modules can define new “built-in” data types

Why learn python? (cont.)

- Reduce development time
- Reduce code length
- Easy to learn and use as developers
- Easy to understand codes
- Easy to do team projects
- Easy to extend to other languages

Where to use python?

- System management (i.e., scripting)
- Systems Programming-shell tools, searching directories,socket, process,thread, Stackless python-multiprocessing system
- Graphic User Interface (GUIs) - Tk GUI API called tkinter, PMW, wxPython,
- GUI support in other toolkits in Python - Qt with PyQt, GTK with PyGTK, MFC with PyWin32, .NET with IronPython, and Swing with Jython or JType
- Internet programming -CGI scripts, XML, FTP, XML-RPC, TelNet : HTMLGen, modPython, Django, TurboGears, web2py, Pylons, Zope, and WebWare – MVC, AJAX
- Database (DB) programming – pickle, ZODB, SQLObject and SQLAlchemy, SQLite
- Component integration – Cython
- Rapid Prototyping
- Text data processing-NLTK
- Distributed processing
- Numerical operations – NumPy, SciPy and ScientificPython
- Graphics –pygame, PIL, PyOpenGL, Blender, Maya
- PyRo , PySol
- And so on…

- Implementations of Python
  - Major implementations include CPython, Jython, IronPython, MicroPython, and PyPy.

Python vs. Perl

- Easier to learn
  - important for occasional users
- More readable code
  - improved code maintenance
- Fewer “magical” side effects
- More “safety” guarantees
- Better Java integration
- Less Unix bias

Python vs. Java

- Code 5-10 times more concise
- Dynamic typing
- Much quicker development
  - no compilation phase
  - less typing
- Yes, it runs slower
  - but development is so much faster!
- Similar (but more so) for C/C++

- Use Python with Java: JPython!

Compare Python to languages - Perl, Tcl, and Java

- Is more powerful than Tcl. Python's support for "programming in the large" makes it applicable to the development of larger systems.
- Has a cleaner syntax and simpler design than Perl, which makes it more readable and maintainable and helps reduce program bugs.
- Is simpler and easier to use than Java. Python is a scripting language, but Java inherits much of the complexity and syntax of systems languages such as C++.
- Is simpler and easier to use than C++, but it doesn't often compete with C++; as a scripting language, Python typically serves different roles.
- Is both more powerful and more cross-platform than Visual Basic. Its open source nature also means it is not controlled by a single company.
- Is more readable and general-purpose than PHP. Python is sometimes used to construct websites, but it's also widely used in nearly every other computer domain, from robotics to movie animation.
- Is more mature and has a more readable syntax than Ruby. Unlike Ruby and Java, OOP is an option in Python—Python does not impose OOP on users or projects to which it may not apply.
- Has the dynamic flavor of languages like SmallTalk and Lisp, but also has a simple, traditional syntax accessible to developers as well as end users of customizable systems.

Python philosophy

- Coherence
  - not hard to read, write and maintain
- power
- scope
  - rapid development + large systems
- objects
- integration
  - hybrid systems

Python features

Lutz, *Programming Python*

no compiling or linking	rapid development cycle
no type declarations	simpler, shorter, more flexible
automatic memory management	garbage collection
high-level data types and operations	fast development
object-oriented programming	code structuring and reuse, C++
embedding and extending in C	mixed language systems
classes, modules, exceptions	"programming-in-the-large" support
dynamic loading of C modules	simplified extensions, smaller binaries
dynamic reloading of C modules	programs can be modified without stopping

Python features

Lutz, *Programming Python*

universal "first-class" object model	fewer restrictions and rules
run-time program construction	handles unforeseen needs, end-user coding
interactive, dynamic nature	incremental development and testing
access to interpreter information	metaprogramming, introspective objects
wide portability	cross-platform programming without ports
compilation to portable byte-code	execution speed, protecting source code
built-in interfaces to external services	system tools, GUIs, persistence, databases, etc.

What not to use Python (and kin) for

- most scripting languages share these
- not as efficient as C
  - but sometimes better built-in algorithms (e.g., hashing and sorting)
- delayed error notification
- lack of profiling tools

Python

- elements from C++, Modula-3 (modules), ABC, Icon (slicing)
- same family as Perl, Tcl, Scheme, REXX, BASIC dialects

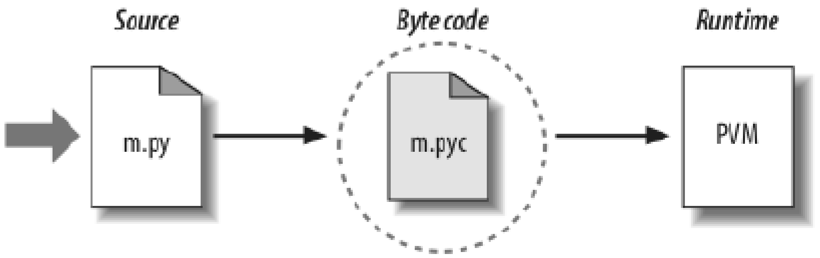
Install Python

Though there are various ways to install Python, the one I would suggest for use in data science is the Anaconda distribution, which works similarly whether you use Windows, Linux, or Mac OS X. The Anaconda distribution comes in two flavors:

- Miniconda gives you the Python interpreter itself, along with a command-line tool called *conda* that operates as a cross-platform package manager geared toward Python packages, similar in spirit to the apt or yum tools that Linux users might be familiar with.
- Anaconda includes both Python and conda, and additionally bundles a suite of other preinstalled packages geared toward scientific computing. Because of the size of this bundle, expect the installation to consume several gigabytes of disk space.

```
[~]$ conda install numpy pandas scikit-learn matplotlib seaborn ipython-notebook
```

Using python



- /usr/local/bin/python
  - `#! /usr/bin/env python`
- interactive use

```
Python 1.6 (#1, Sep 24 2000, 20:40:45)  [GCC 2.95.1 19990816 (release)] on sunos5
Copyright (c) 1995-2000 Corporation for National Research Initiatives.
All Rights Reserved.
Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>>
```

- `python –c` *`command`* `[`*`arg`*`] ...`
- `python –i` *`script`*
  - read script first, then interactive

Running Python Interactively

- Start python by typing "python"
  - /afs/isis/pkg/isis/bin/python
- ^D (control-D) exits
  - % python
  - >>> ^D
  - %
- Comments start with '#'
  - >>> 2+2 #Comment on the same line as text
  - 4
  - >>> 7/3 #Numbers are integers by default
  - 2
  - >>> x = y = z = 0 #Multiple assigns at once
  - >>> z
  - 0

- city='London' # A string variable assignment.
- money = 100.75 # A floating point number assignment
- count=4 #An integer assignment

Running Python Programs

- In general
    - % python ./myprogram.py
- Can also create executable scripts
  - Compose the code in an editor like `vi/emacs`
    - % vi ./myprogram.py # Python scripts with the suffix .py.
  - Then you can just type the script name to execute
    - % python ./myprogram.py
- The first line of the program tells the OS how to execute it:
    - #! /afs/isis/pkg/isis/bin/python
  - Make the file executable:
    - % chmod +x ./myprogram.py
  - Then you can just type the script name to execute
    - % ./myprogram.py

Running Python Programs Interactively

Suppose the file `script.py` contains the following lines:

```
print 'Hello world'
x = [0,1,2]
```

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

-
  ```
  python -i script.py
  Hello world
  >>> x
  [0,1,2]
  ```

-
  ```
  $ python
  >>> execfile('script.py')
  >>> x
  [0,1,2]
  ```

Running Python Programs Interactively

Suppose the file `script.py` contains the following lines:

```
print 'Hello world'
x = [0,1,2]
```

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

- python

```
>>> import script  # DO NOT add the .py suffix.  Script is a module here
>>> x
    Traceback (most recent call last):
    File "<stdin>", line 1, in ?
    NameError: name 'x' is not defined
>>> script.x      # to make use of x, we need to let Python know which
                  #module it came from, i.e. give Python its context
    [0,1,2]
```

Running Python Programs Interactively

# Pretend that script.py contains multiple stored quantities. To promote x(and only x) to the top level context, type the following:

- `$ python`

```
>>> from script import x
Hello world
>>> x
[0,1,2]
>>>
```

# To promote all quantities in `script.py` to the top level context, type `from script import *` into the interpreter. Of course, if that's what you want, you might as well type `python -i script.py` into the terminal.

```
>>> from script import *
```

File naming conventions

- python files usually end with the suffix `.py`
- but executable files usually don’t have the `.py` extension
- modules (later) should <u>always</u> have the `.py` extension

Comments

- Start with # and go to end of line
- What about C, C++ style comments?
  - NOT supported!

- #This is a single line comment in Python
- print "Hello World"
- #This is a single comment in Python

- """ For multi-line
- comment use three
- double quotes
- ...
- """
- print "Hello World!"

Python structure

- modules: Python source files
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
- objects
  - everything is an object
  - automatically reclaimed when no longer needed

First example

```
#!/usr/local/bin/python
# import systems module
import sys
marker = ':::::::'
for name in sys.argv[1:]:
  input = open(name, 'r')
 print marker + name
  print input.read()
```

Python Syntax

- Much of it is similar to C syntax
- Exceptions:
  - missing operators: `++`, `--`
  - no curly brackets, `{}`, for blocks; uses whitespace
  - different keywords
  - lots of extra features
  - no type declarations!

Printing

- print “Hello World!”
- print 'Hello World! ‘
- print ‘ ‘ ‘ I am mad in love

  do you think

  I am doing

  the right thing ’ ’ ’
- print “““ I am mad in love

  do you think

  I am doing

  the right thing ”””
- print "Hello \

  world “
- print 'old world "but" still good'

- print 'Hey there it's a cow'
- print "Hey there it's a cow”
- print "Only way to join" + "two strings”
- print "John Doe", 80.67, "27”
- print ("%s %14.2f %11d" % ("John Doe", 80.67, 27))

Indentation

- Indentation not only makes Python code readable, but also distinguishes each block of code from the other.
- def fun():

```
def fun():
        pass
        for each in "Australia":
                pass
```

- Here, most people get confused whether to use a tab or space. It is advisable to stick to only one type and follow the same across the whole code.
- If the indentation is not strictly implemented, then code execution will throw an error.

Variables

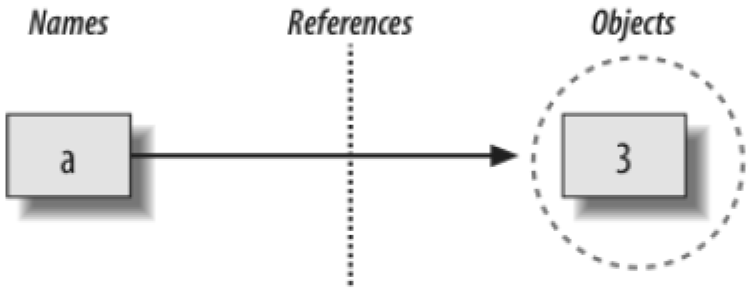
- No need to declare. Variable names can begin with _, $, or a letter.
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed

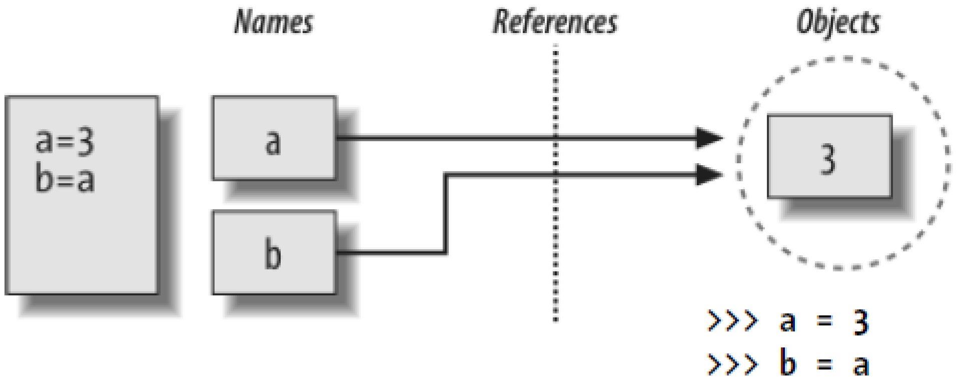
```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

- ***Everything*** is a variable:
  - functions, modules, classes
- Depending on the data type of the variable, the interpreter allocates memory and makes a decision to store a particular data type in the reserved memory.
- Python variables are usually dynamically typed, that is, the type of the variable is interpreted during runtime and you need not specifically provide a type to the variable name, unlike what other programming languages require.

Variables

- Variables are created when they are first assigned values.
- Variables are replaced with their values when used in expressions.
- Variables must be assigned before they can be used in expressions.
- Variables refer to objects and are never declared ahead of time.
- Variables are entries in a system table, with spaces for links to objects.
- Objects are pieces of allocated memory, with enough space to represent the values for which they stand.
- References are automatically followed pointers from variables to objects.





```
>>> a = 3
>>> b = a
```

- The answer is that in Python, whenever a name is assigned to a new object, the space held by the prior object is reclaimed (if it is not referenced by any other name or object). This automatic reclamation of objects' space is known as **garbage collection**.

Reference semantics

- Internally, Python accomplishes this feat by keeping a counter in every object that keeps track of the number of references currently pointing to that object. As soon as (and exactly when) this counter drops to zero, the object's memory space is automatically reclaimed. however, it also has a component that detects and reclaims objects with ***cyclic references*** in time. This component can be disabled if you're sure that your code doesn't create cycles, but it is enabled by default.
- 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]
```

@ Shubin Liu

Simple data types: operators

- **`+ - * / %`** (like C) // (floor division)
- **`+= -=`** etc. (no **`++`** or **`--`**)
- Assignment using **`=`**
  - but semantics are different!

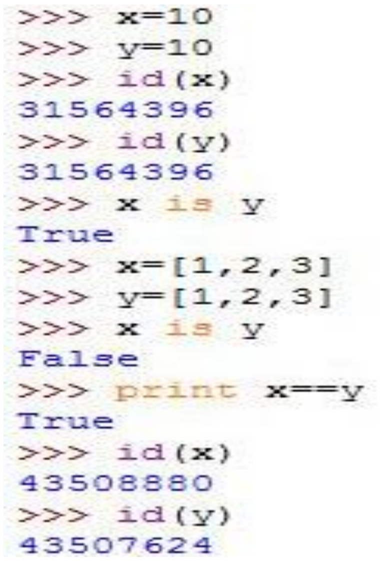
```
a = 1
a = "foo"  # OK
```

- Can also use **`+`** to concatenate strings
- Infix operator
- Walrus operator := in Python 3.8

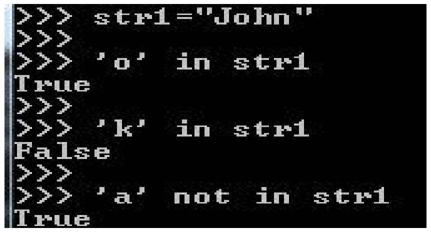
@ Shubin Liu

Operators

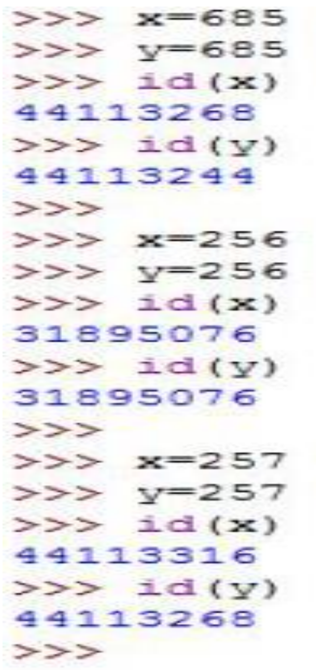
- Arithmetic operators.
- Comparison operators - ==, <, >, <=, >=, !=, <>
- Assignment operators - = x=y , += x+=y , -= x-=y,
  = x=y, /= x/=y, **= x**=y
- Bitwise operators
- Logical operators – and, or, not
- Membership operators – in, not in
- Identity operators – is, is not



```
>>> x=10
>>> y=10
>>> id(x)
31564396
>>> id(y)
31564396
>>> x is y
True
>>> x=[1,2,3]
>>> y=[1,2,3]
>>> x is y
False
>>> print x==y
True
>>> id(x)
43508880
>>> id(y)
43507624
```



```
>>> str1="John"
>>>
>>> 'o' in str1
True
>>>
>>> 'k' in str1
False
>>>
>>> 'a' not in str1
True
```



```
>>> x=685
>>> y=685
>>> id(x)
44113268
>>> id(y)
44113244
>>>
>>> x=256
>>> y=256
>>> id(x)
31895076
>>> id(y)
31895076
>>>
>>> x=257
>>> y=257
>>> id(x)
44113316
>>> id(y)
44113268
>>>
```

`is`	Returns `True` if two variables point to the same object and `False`, otherwise
`is not`	Returns `False` if two variables point to the same object and `True`, otherwise

- The id() function returns the *identity* of an object. This is an integer (or long integer), which is guaranteed to be unique and constant for this object during its lifetime. It is similar to memory addresses in C language.

Bitwise Operators

>>> **X = 0b0001** # *Binary literals*

>>> **X << 2** # *Shift left*

4

>>> **bin(X << 2)** # *Binary digits string*

'0b100'

>>> **bin(X | 0b010)** # *Bitwise OR*

'0b11'

>>> **bin(X & 0b1)** # *Bitwise AND*

'0b1'

>>> **X = 0xFF** # *Hex literals*

~ Performs binary *XOR* operation

^ Performs binary one's complement operation

>>> **bin(X)**

'0b11111111'

>>> **X ^ 0b10101010** # *Bitwise XOR*

85

>>> **bin(X ^ 0b10101010)**

'0b1010101'

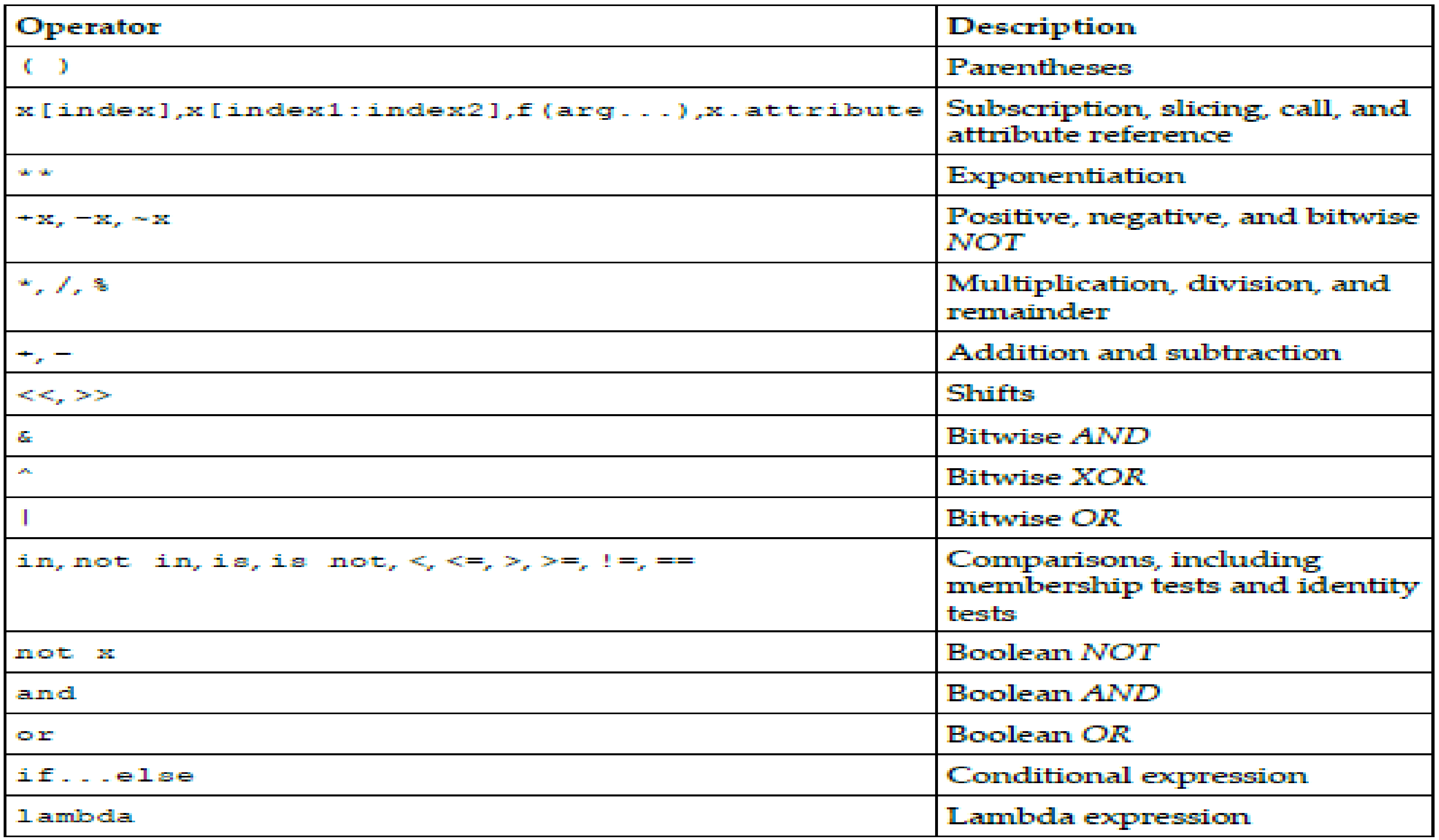
>>> **int('1010101', 2)** # *String to int per base*

85

>>> **hex(85)** # *Hex digit string*

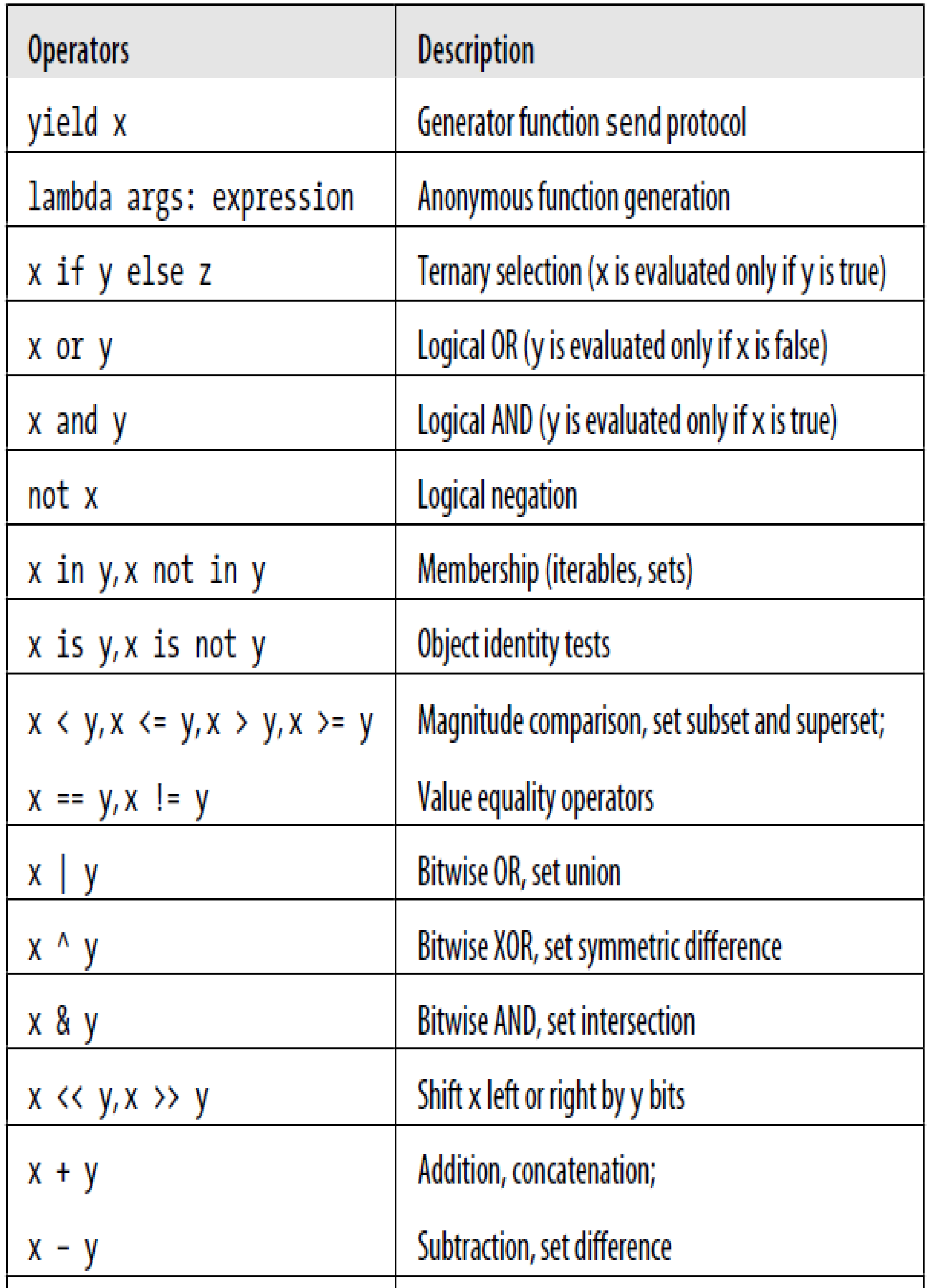
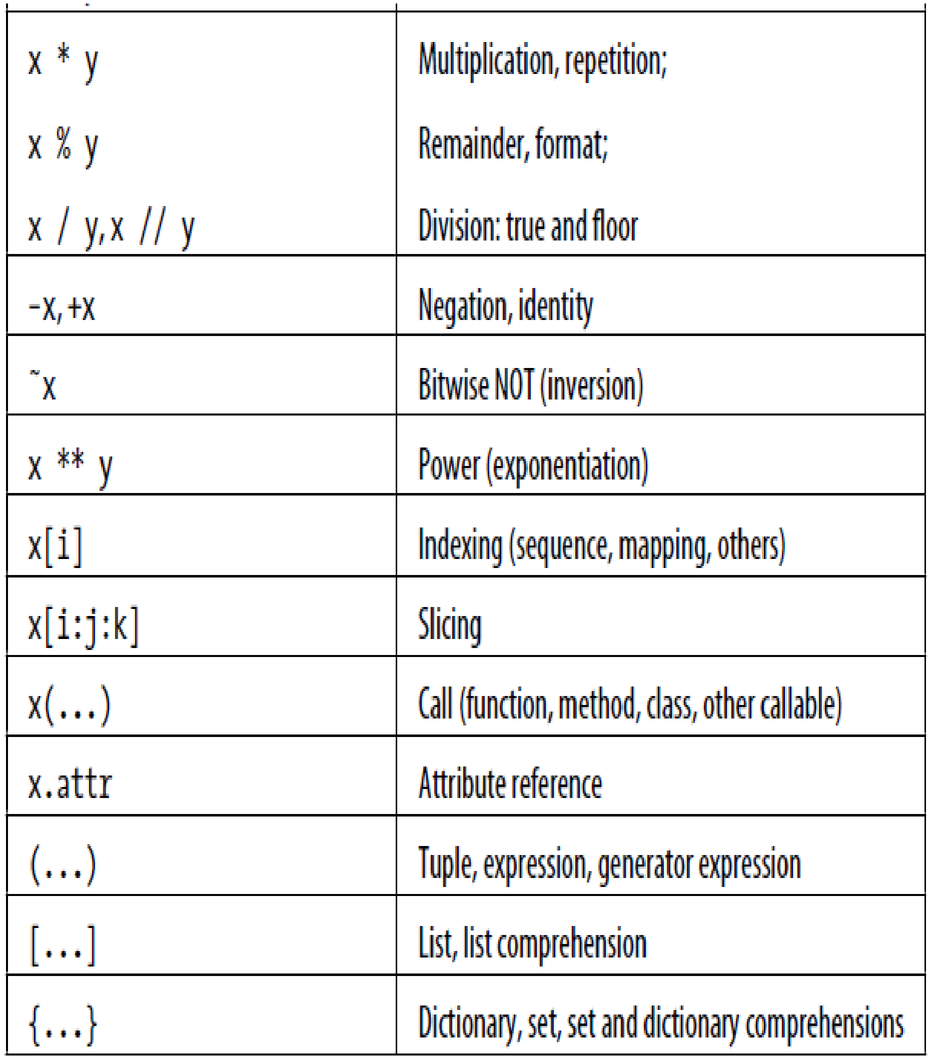
'0x55'

Operator Precedence



Operator	Description
( )	Parentheses
x[index],x[index1:index2],f(arg...),x.attribute	Subscription, slicing, call, and attribute reference
**	Exponentiation
+x, −x, ~x	Positive, negative, and bitwise *NOT*
*, /, %	Multiplication, division, and remainder
+, −	Addition and subtraction
<<, >>	Shifts
&	Bitwise *AND*
^	Bitwise *XOR*
|	Bitwise *OR*
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
not x	Boolean *NOT*
and	Boolean *AND*
or	Boolean *OR*
if...else	Conditional expression
lambda	Lambda expression

Operators that have the same precedence are evaluated from left to right, except for comparisons and exponentiation. Comparisons can be chained arbitrarily.

Operators	Description
yield x	Generator function send protocol
lambda args: expression	Anonymous function generation
x if y else z	Ternary selection (x is evaluated only if y is true)
x or y	Logical OR (y is evaluated only if x is false)
x and y	Logical AND (y is evaluated only if x is true)
not x	Logical negation
x in y,x not in y	Membership (iterables, sets)
x is y,x is not y	Object identity tests
x < y,x <= y,x > y,x >= y	Magnitude comparison, set subset and superset;
x == y,x != y	Value equality operators
x | y	Bitwise OR, set union
x ^ y	Bitwise XOR, set symmetric difference
x & y	Bitwise AND, set intersection
x << y,x >> y	Shift x left or right by y bits
x + y	Addition, concatenation;
x - y	Subtraction, set difference
x * y	Multiplication, repetition;
x % y	Remainder, format;
x / y,x // y	Division: true and floor
-x, +x	Negation, identity
~x	Bitwise NOT (inversion)
x ** y	Power (exponentiation)
x[i]	Indexing (sequence, mapping, others)
x[i:j:k]	Slicing
x(...)	Call (function, method, class, other callable)
x.attr	Attribute reference
(...)	Tuple, expression, generator expression
[...]	List, list comprehension
{...}	Dictionary, set, set and dictionary comprehensions

Simple Data Types

- is operator - tests object identity (i.e., address in memory, a strict form
- of equality)
- lambda - creates unnamed functions
- Triple quotes useful for multi-line strings

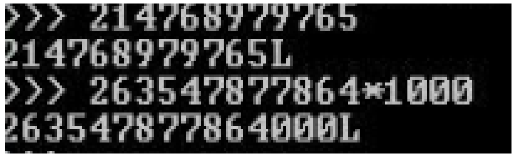
```
>>> s = """ a long
... string with "quotes" or anything else"""
>>> s
' a long\012string with "quotes" or anything
else'
>>> len(s)
45
```

Comparisons

- In Python 2.X, magnitude comparisons of mixed types—converting numbers to a common type, and ordering other mixed types according to the type name—are allowed. In Python 3.0, nonnumeric mixed-type magnitude comparisons are not allowed and raise exceptions; this includes sorts by proxy.
- Magnitude comparisons for dictionaries are also no longer supported in Python 3.0 (though equality tests are); comparing *sorted(dict.items())* is one possible replacement.
- **Mixed operators follow operator precedence**
- In mixed-type numeric expressions, Python first **converts operands up** to the type of the most complicated operand, and then performs the math on same-type operands. This behavior is similar to type conversions in the C language.
- All Python operators may be **overloaded** (i.e., implemented) by Python classes and C extension types to work on objects you create.
- **Polymorphism** = + operator performs addition when applied to numbers but performs concatenation when applied to sequence objects such as strings and lists.

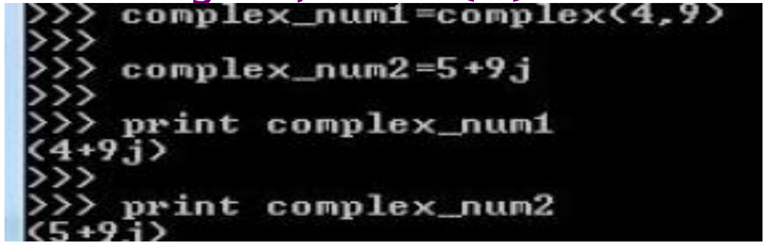
Simple data types

- Numbers
- integers,
  - int : from -2^{31} to (2^{31}-1)
- long integers - interpreter will add L



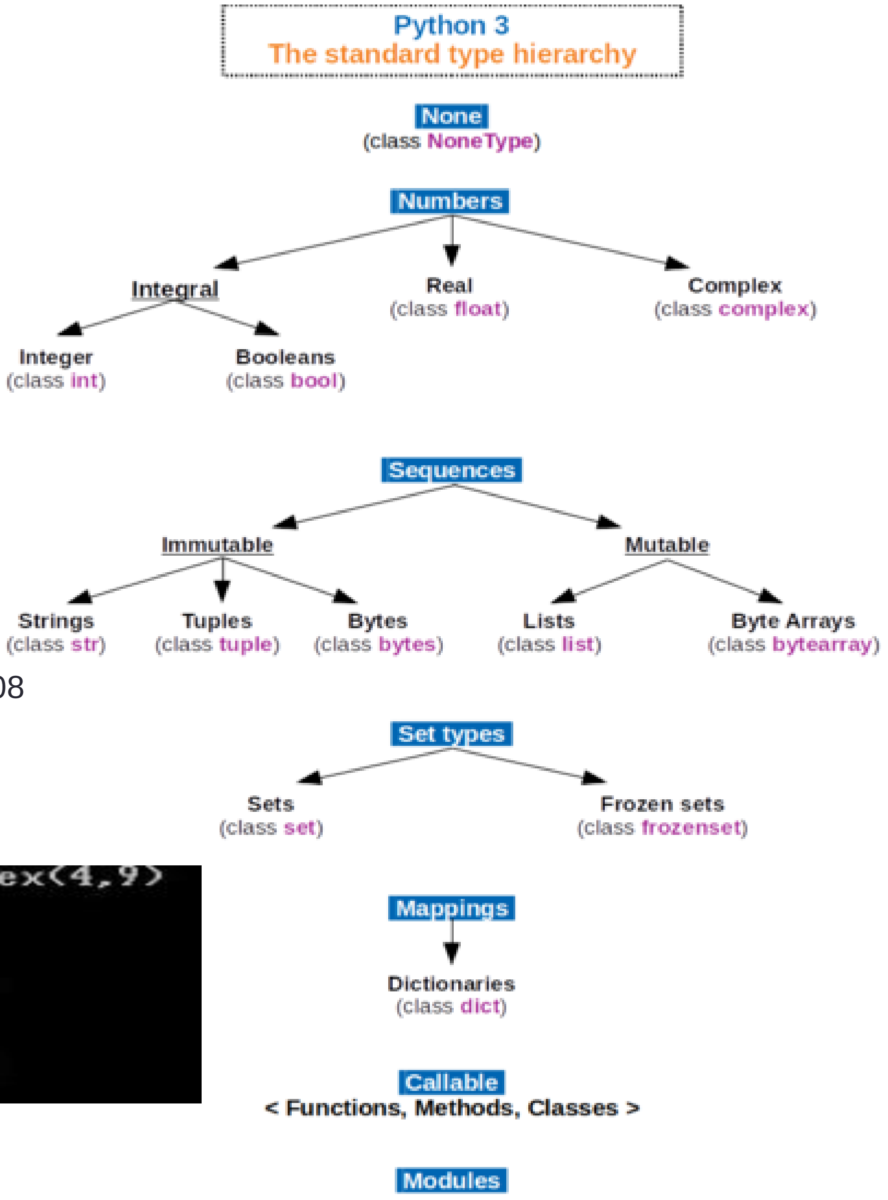
```
>>> 2147689779765
2147689779765L
>>> 26354787786 4*1000
263547877864000L
```

- floating point numbers,
  - ranges approximately from -10 to 10^^{308}
  - 16 digits of precision
- complex numbers - `1j+3, abs(z)`



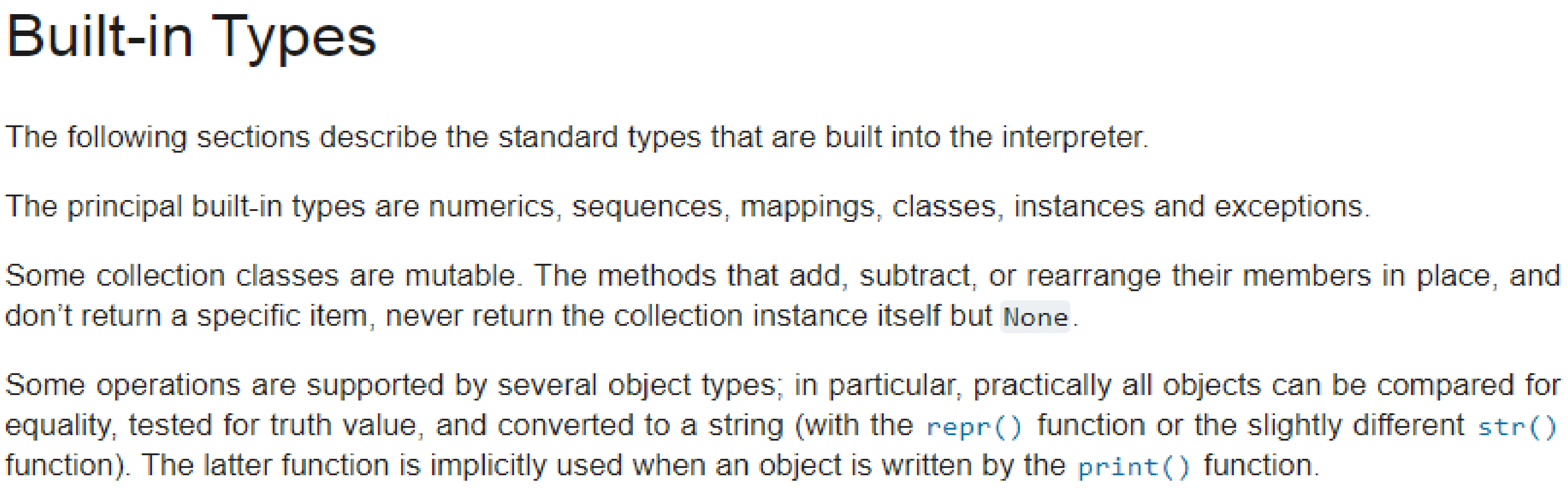
```
>>> complex_num1=complex(4,9)
>>>
>>> complex_num2=5+9j
>>>
>>> print complex_num1
(4+9j)
>>>
>>> print complex_num2
(5+9j)
```

- Booleans are **`False`** or **`True`**



Numeric Types

- Integer
- Boolean
- Real - Decimal type
- Fraction type
- Complex



Built-in Types

The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don't return a specific item, never return the collection instance itself but `None`.

Some operations are supported by several object types; in particular, practically all objects can be compared for equality, tested for truth value, and converted to a string (with the `repr()` function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print()` function.

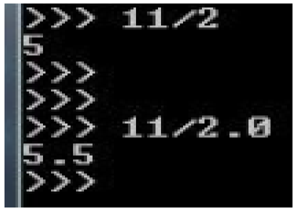
Numbers

- The usual notations and operators
  - 12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
  - 1<<16, x&0xff, x|1, ~x, x^y
- Integer division truncates :-(
  - 1/2 -> 0   # float(1)/2 -> 0.5
- Long (arbitrary precision), complex
  - 2L**100 -> 1267650600228229401496703205376L
  - 1j**2 -> (-1+0j)
- Boolean - 'True' or 'False'

Arithmetic Expression

- **Bracket, Of, Division, Multiplication, Addition, and Subtraction (BODMAS)**
- The decreasing precedence order is as follows:
  - Exponent
  - Unary negation
  - Multiplication, division, modulus
  - Addition, subtraction
- Addition of two int data types can produce a long integer.
- **Mixed mode conversion**
- int(), float()

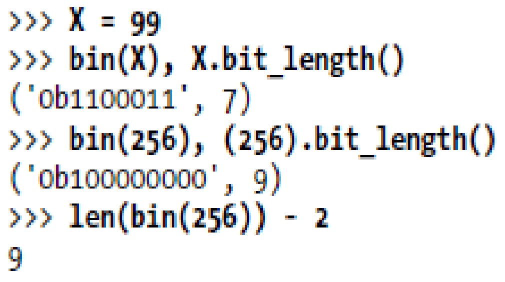
Operator
**
*
/
%
+
−



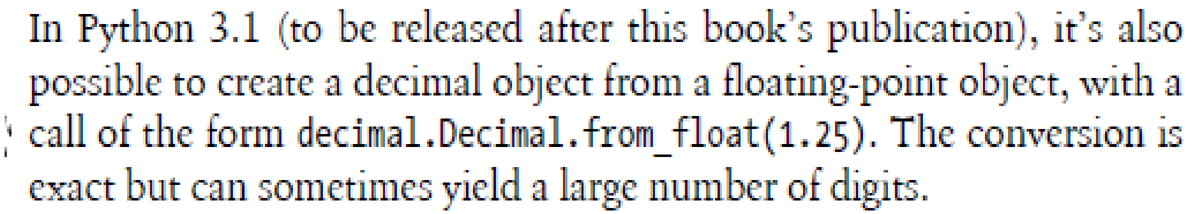
```
>>> 11/2
5
>>>
>>>
>>> 11/2.0
5.5
>>>
```

Numeric Types

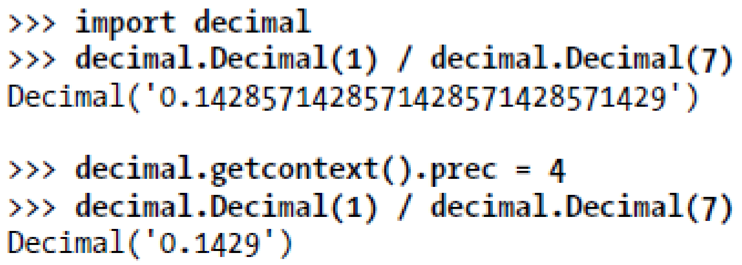
- Floating point type
  - as_integer_ratio
  - is_integer
- Integer type
  - bit_length
- Decimal type
  - Decimal Context manager
- Fraction type
- Sets
- Boolean



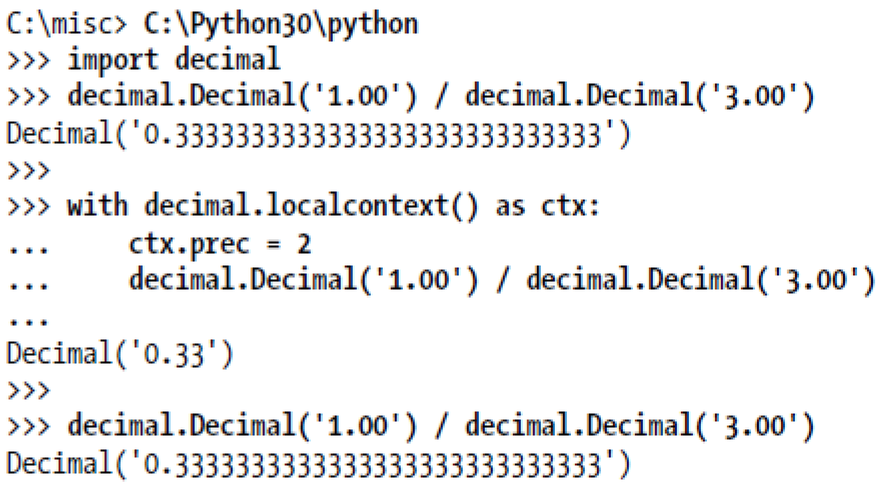
```
>>> X = 99
>>> bin(X), X.bit_length()
('0b1100011', 7)
>>> bin(256), (256).bit_length()
('0b100000000', 9)
>>> len(bin(256)) - 2
9
```



In Python 3.1 (to be released after this book's publication), it's also possible to create a decimal object from a floating-point object, with a call of the form decimal.Decimal.from_float(1.25). The conversion is exact but can sometimes yield a large number of digits.



```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1428571428571428571428571429')

>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
```



```
C:\misc> C:\Python30\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.3333333333333333333333333333')
>>>
>>> with decimal.localcontext() as ctx:
...     ctx.prec = 2
...     decimal.Decimal('1.00') / decimal.Decimal('3.00')
...
Decimal('0.33')
>>>
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.3333333333333333333333333333')
```

Built in constants

A small number of constants live in the built-in namespace. They are:

`False`

The false value of the `bool` type. Assignments to `False` are illegal and raise a `SyntaxError`.

`True`

The true value of the `bool` type. Assignments to `True` are illegal and raise a `SyntaxError`.

`None`

The sole value of the type `NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a `SyntaxError`.

```
>>> 4.0 / 3
1.3333333333333333
>>> (4.0 / 3).as_integer_ratio()                    # Precision loss from float
(6004799503160661, 4503599627370496)

>>> x
Fraction(1, 3)
>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())
>>> a
Fraction(22517998136852479, 13510798882111488)

>>> 22517998136852479 / 13510798882111488.          # 5 / 3 (or close to it!)
1.6666666666666667

>>> a.limit_denominator(10)                         # Simplify to closest fraction
Fraction(5, 3)
```

Display Formatting

```
>>> num = 1 / 3.0
>>> num # Echoes
0.33333333333333331
>>> print(num) # print rounds
0.333333333333
>>> '%e' % num # String formatting expression
'3.333333e-001'
>>> '%4.2f' % num # Alternative floating-point format
'0.33'
>>> '{0:4.2f}'.format(num) # String formatting method (Python 2.6 and 3.0)
'0.33‘
>>> repr(num) # Used by echoes: as-code form
'0.33333333333333331'
>>> str(num) # Used by print: user-friendly form
'0.333333333333'
```

Integer Precision

- Python 3.0 integers support unlimited size:

```
>>> 99999999999999999999999999999999 + 1
100000000000000000000000000000000
```

- Unlimited-precision integers are a convenient built-in tool with performance penalty.

```
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376
```

- Python 2.6 has a separate type for long integers, but it automatically converts any number too large to store in a normal integer to this type.

```
>>> 99999999999999999999999999999999 + 1
100000000000000000000000000000000L
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376L
```

Complex Numbers

- Complex numbers are a distinct core object type in Python.

```
>>> 1j * 1J
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
>>> oct(64), hex(64), bin(64)
('0100', '0x40', '0b1000000')
>>> 0o1, 0o20, 0o377 # Octal literals
(1, 16, 255)
>>> 0x01, 0x10, 0xFF # Hex literals
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111 # Binary literals
(1, 16, 255)
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)
```

Built-in Numeric Tools

```
>>> import math
>>> math.pi, math.e # Common constants
(3.141592653589793, 2.718281828459045)
>>> math.sin(2 * math.pi / 180) # Sine, tangent, cosine
0.03489949670250097
>>> math.sqrt(144), math.sqrt(2) # Square root
(12.0, 1.4142135623730951)
>>> pow(2, 4), 2 ** 4 # Exponentiation (power)
(16, 16)
>>> math.floor(2.567), math.floor(-2.567) # Floor (next-lower integer)
(2, −3)
>>> math.trunc(2.567), math.trunc(−2.567) # Truncate (drop decimal digits)
(2, −2)
>>> import random
>>> random.random()
0.44694718823781876
>>> random.random()
0.28970426439292829
>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
4
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Life of Brian'
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Holy Grail'
```

Other Numeric Types

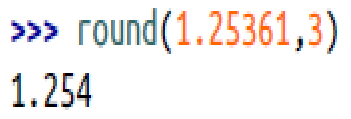
Decimal Type

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')
Decimal('0.00')
>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
>>> 0.1 + 0.1 + 0.1 - 0.3 # This should be zero (close, but not exact)
5.551115123125783e-17
>>> Fraction.from_float(1.75) # Convert float -> fraction: other way
Fraction(7, 4)
```

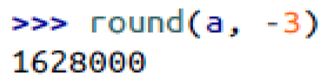
Fraction Type

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3) # Numerator, denominator
>>> y = Fraction(4, 6) # Simplified to 2, 3 by gcd
>>> print(y)
2/3
>>> x * y
Fraction(2, 9)
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)
```

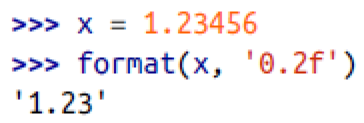
The limitations of floating-point hardware and its inability to exactly represent some values in a limited number of bits.



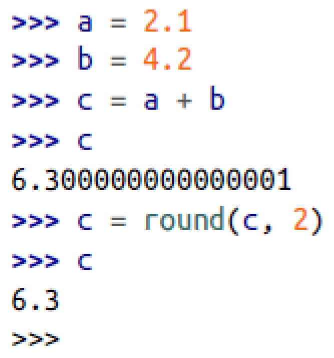
```
>>> round(1.25361,3)
1.254
```



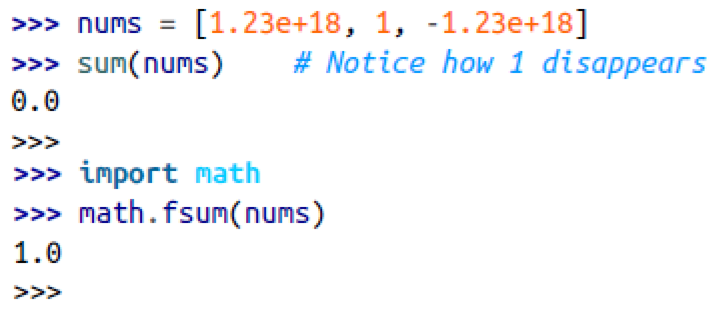
```
>>> round(a, -3)
1628000
```



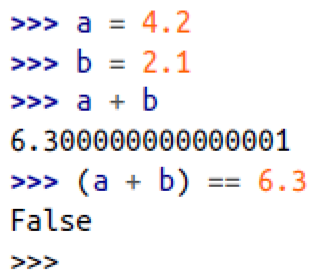
```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
```



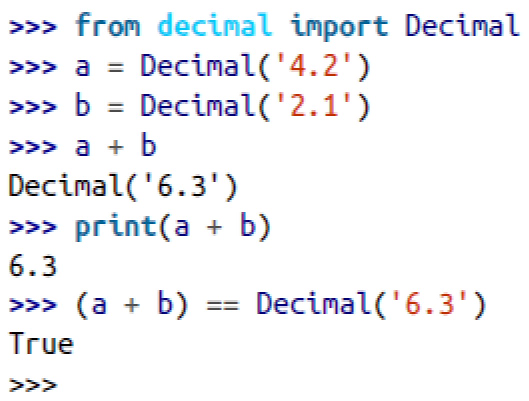
```
>>> a = 2.1
>>> b = 4.2
>>> c = a + b
>>> c
6.300000000000001
>>> c = round(c, 2)
>>> c
6.3
>>>
```



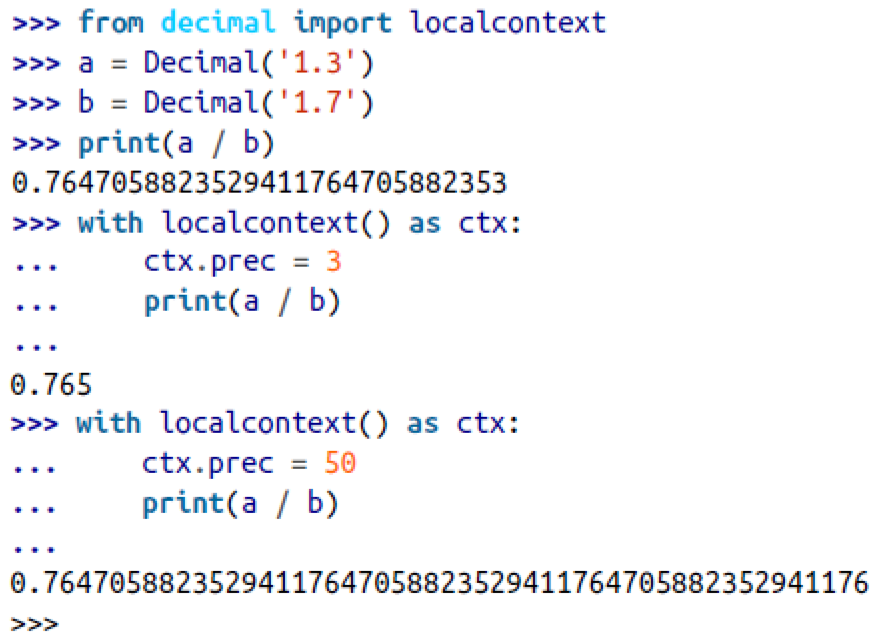
```
>>> nums = [1.23e+18, 1, -1.23e+18]
>>> sum(nums)     # Notice how 1 disappears
0.0
>>>
>>> import math
>>> math.fsum(nums)
1.0
>>>
```



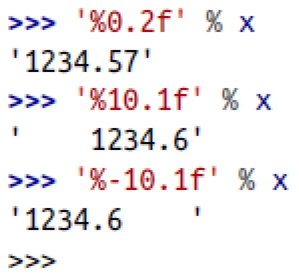
```
>>> a = 4.2
>>> b = 2.1
>>> a + b
6.300000000000001
>>> (a + b) == 6.3
False
>>>
```



```
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
>>>
```



```
>>> from decimal import localcontext
>>> a = Decimal('1.3')
>>> b = Decimal('1.7')
>>> print(a / b)
0.7647058823529411764705882353
>>> with localcontext() as ctx:
...     ctx.prec = 3
...     print(a / b)
...
0.765
>>> with localcontext() as ctx:
...     ctx.prec = 50
...     print(a / b)
...
0.76470588235294117647058823529411764705882352941176
>>>
```



```
>>> '%0.2f' % x
'1234.57'
>>> '%10.1f' % x
'    1234.6'
>>> '%-10.1f' % x
'1234.6    '
>>>
```

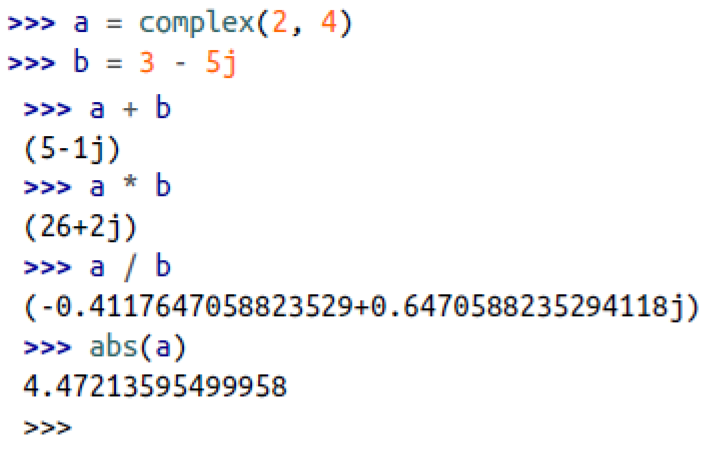
Comparisons

- Normal

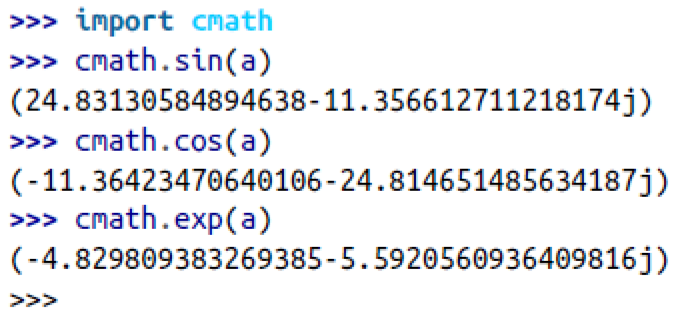
```
>>> 1 < 2                    # Less than
True
>>> 2.0 >= 1                 # Greater than or equal: mixed-type 1 converted to 1.0
True
>>> 2.0 == 2.0               # Equal value
True
>>> 2.0 != 2.0               # Not equal value
False
```

- Chained

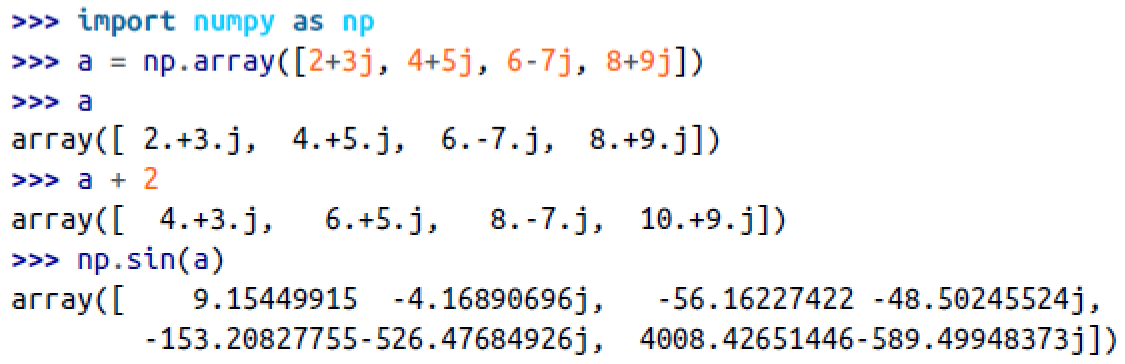
```
>>> X = 2          >>> X < Y < Z              # Chained comparisons: range tests
>>> Y = 4          True
>>> Z = 6          >>> X < Y and Y < Z
                   True

>>> 1 == 2 < 3                 # Same as: 1 == 2 and 2 < 3
False                          # Not same as: False < 3 (which means 0 < 3, which is true)
```



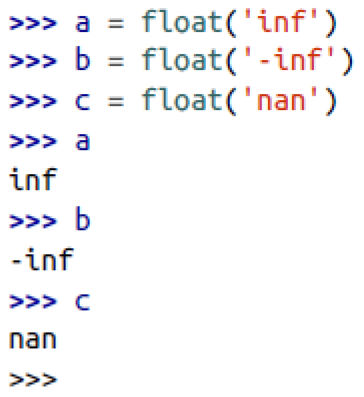
```
>>> a = complex(2, 4)
>>> b = 3 - 5j
 >>> a + b
 (5-1j)
 >>> a * b
 (26+2j)
 >>> a / b
 (-0.4117647058823529+0.6470588235294118j)
 >>> abs(a)
 4.47213595499958
 >>>
```



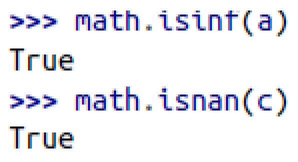
```
>>> import cmath
>>> cmath.sin(a)
(24.83130584894638-11.356612711218174j)
>>> cmath.cos(a)
(-11.36423470640106-24.814651485634187j)
>>> cmath.exp(a)
(-4.829809383269385-5.5920560936409816j)
>>>
```



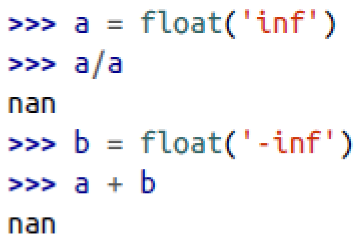
```
>>> import numpy as np
>>> a = np.array([2+3j, 4+5j, 6-7j, 8+9j])
>>> a
array([ 2.+3.j,  4.+5.j,  6.-7.j,  8.+9.j])
>>> a + 2
array([  4.+3.j,   6.+5.j,   8.-7.j,  10.+9.j])
>>> np.sin(a)
array([    9.15449915  -4.16890696j,   -56.16227422 -48.50245524j,
        -153.20827755-526.47684926j,  4008.42651446-589.49948373j])
```



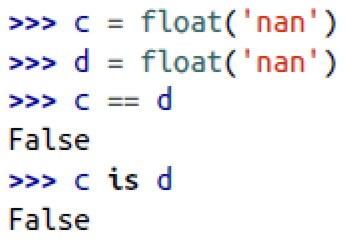
```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```



```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
```



```
>>> a = float('inf')
>>> a/a
nan
>>> b = float('-inf')
>>> a + b
nan
```



```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is d
False
```

Divisions

- In **3.0**, the / now always performs true division, returning a float result that includes any remainder, regardless of operand types. The // performs floor division, which truncates the remainder and returns an integer for integer operands or a float if any operand is a float.
- In **2.6**, the / does classic division, performing truncating integer division if both operands are integers and float division (keeping remainders) otherwise. The // does floor division and works as it does in 3.0, performing truncating division for integers and floor division for floats.

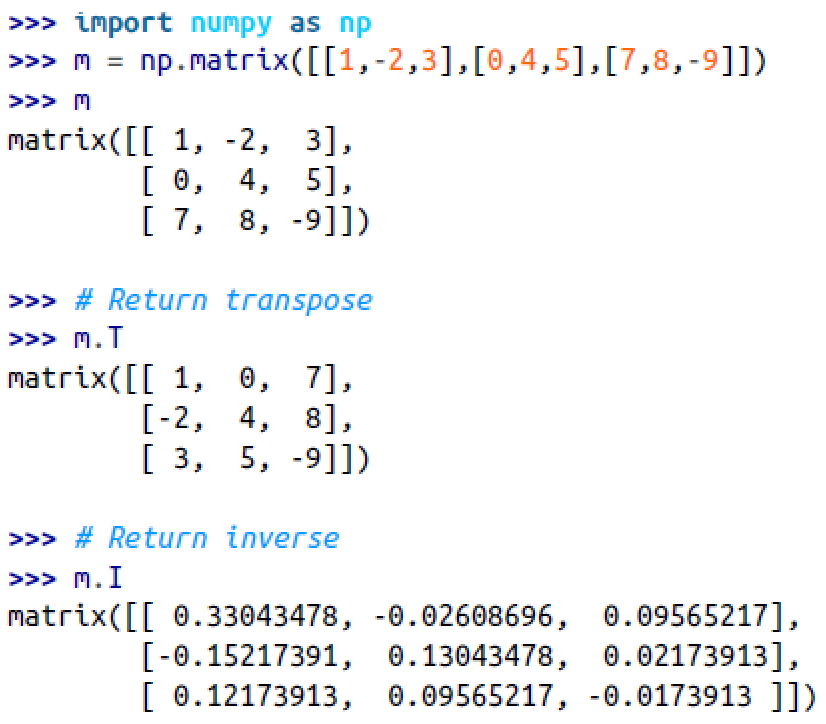
```
C:\misc> C:\Python30\python
>>>
>>> 10 / 4                # Differs in 3.0: keeps remainder
2.5
>>> 10 // 4               # Same in 3.0: truncates remainder
2
>>> 10 / 4.0              # Same in 3.0: keeps remainder
2.5
>>> 10 // 4.0             # Same in 3.0: truncates to floor
2.0
```

```
C:\misc> C:\Python26\python
>>>
>>> 10 / 4
2
>>> 10 // 4
2
>>> 10 / 4.0
2.5
>>> 10 // 4.0
2.0
```

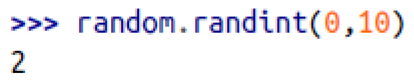
```
C:\misc> C:\Python26\python
>>> from __future__ import division          # Enable 3.0 "/" behavior
>>> 10 / 4
2.5
>>> 10 // 4
2
```

```
>>> from fractions import Fraction
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b)
35/64

>>> # Getting numerator/denominator
>>> c = a * b
>>> c.numerator
35
>>> c.denominator
64

>>> # Converting to a float
>>> float(c)
0.546875

>>> # Limiting the denominator of a value
>>> print(c.limit_denominator(8))
4/7

>>> # Converting a float to a fraction
>>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(15, 4)
>>>
```

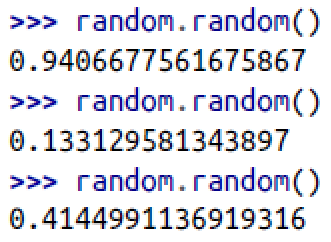
```
>>> # Numpy arrays
>>> import numpy as np
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6,  8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])

>>> np.sqrt(ax)
array([ 1.        ,  1.41421356,  1.73205081,  2.        ])
>>> np.cos(ax)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])

>>> # Broadcast a row vector across an operation on all rows
>>> a + [100, 101, 102, 103]
array([[101, 103, 105, 107],
       [105, 117, 119, 111],
       [109, 121, 123, 115]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 20, 21, 12]])

>>> # Conditional assignment on an array
>>> np.where(a < 10, a, 10)
array([[ 1,  2,  3,  4],
       [ 5, 10, 10,  8],
       [ 9, 10, 10, 10]])
```



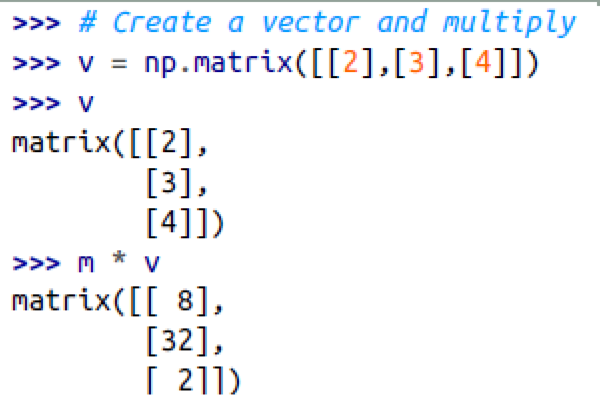
```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2,  3],
        [ 0,  4,  5],
        [ 7,  8, -9]])

>>> # Return transpose
>>> m.T
matrix([[ 1,  0,  7],
        [-2,  4,  8],
        [ 3,  5, -9]])

>>> # Return inverse
>>> m.I
matrix([[ 0.33043478, -0.02608696,  0.09565217],
        [-0.15217391,  0.13043478,  0.02173913],
        [ 0.12173913,  0.09565217, -0.0173913 ]])
```



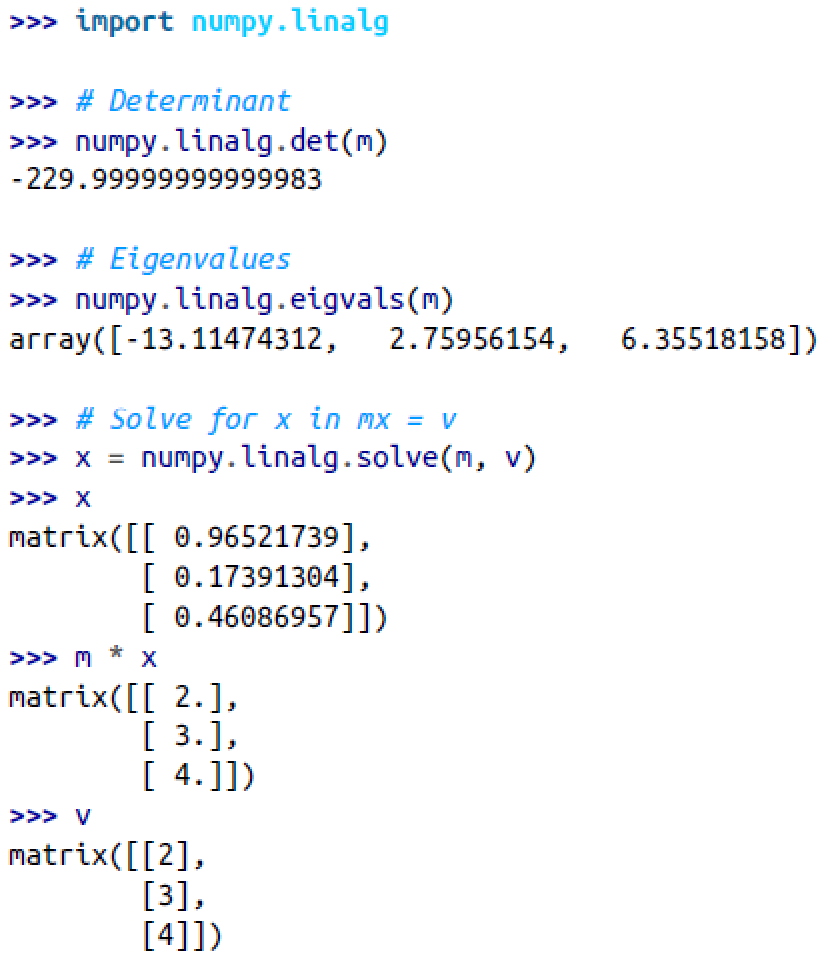
```
>>> random.randint(0,10)
2
```



```
>>> random.random()
0.9406677561675867
>>> random.random()
0.133129581343897
>>> random.random()
0.4144991136919316
```



```
>>> # Create a vector and multiply
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],
        [3],
        [4]])
>>> m * v
matrix([[ 8],
        [32],
        [ 2]])
```



```
>>> import numpy.linalg

>>> # Determinant
>>> numpy.linalg.det(m)
-229.99999999999983

>>> # Eigenvalues
>>> numpy.linalg.eigvals(m)
array([-13.11474312,   2.75956154,   6.35518158])

>>> # Solve for x in mx = v
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
>>> m * x
matrix([[ 2.],
        [ 3.],
        [ 4.]])
>>> v
matrix([[2],
        [3],
        [4]])
```

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
```

```
>>> d = datetime(2012, 12, 21, 9, 30, 0)
>>> print(d)
2012-12-21 09:30:00
>>>

>>> # Localize the date for Chicago
>>> central = timezone('US/Central')
>>> loc_d = central.localize(d)
>>> print(loc_d)
2012-12-21 09:30:00-06:00
```

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
>>> print(d)
2012-12-23 16:31:52.718111

>>> # Next Friday
>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>

>>> # Last Friday
>>> print(d + relativedelta(weekday=FR(-1)))
2012-12-21 16:31:52.718111
```

```
>>> from datetime import datetime
>>> text = '2012-09-20'
>>> y = datetime.strptime(text, '%Y-%m-%d')
>>> z = datetime.now()
>>> diff = z - y
>>> diff
datetime.timedelta(3, 77824, 177393)

>>> z
datetime.datetime(2012, 9, 23, 21, 37, 4, 177393)
>>> nice_z = datetime.strftime(z, '%A %B %d, %Y')
>>> nice_z
'Sunday September 23, 2012'
```

Modules

- import module:
```
import fibo
```
- Use modules via "name space":
```
>>> fibo.fib(1000)
>>> fibo.__name__
'fibo'
```
- can give it a local name:
```
>>> fib = fibo.fib
>>> fib(500)
```

Modules

- Access other code by importing modules

```
import math
print math.sqrt(2.0)
```

- or:

```
from math import sqrt
print sqrt(2.0)
```

Modules

- or:

```
from math import *
print sqrt(2.0)
```

- Can import multiple modules on one line:

```
import sys, string, math
```

- Only one "`from x import y`" per line

Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable
- uses installation-default if not defined, e.g., .:/usr/local/lib/python
- uses sys.path

```
>>> import sys
>>> sys.path
['', 'C:\\PROGRA~1\\Python2.2', 'C:\\Program
  Files\\Python2.2\\DLLs', 'C:\\Program Files\\Python2.2\\lib',
  'C:\\Program Files\\Python2.2\\lib\\lib-tk', 'C:\\Program
  Files\\Python2.2', 'C:\\Program Files\\Python2.2\\lib\\site-
  packages']
```

Example: NumPy Modules

- http://numpy.scipy.org/
- NumPy has many of the features of Matlab, in a free, multiplatform program. It also allows you to do intensive computing operations in a simple way
- Numeric Module: Array Constructors
  - ones, zeros, identity
  - arrayrange
- LinearAlgebra Module: Solvers
  - Singular Value Decomposition
  - Eigenvalue, Eigenvector
  - Inverse
  - Determinant
  - Linear System Solver

Arrays and Constructors

- >>> a = ones((3,3),float)
- >>> print a
- [[1., 1., 1.],
- [1., 1., 1.],
- [1., 1., 1.]]
- >>> b = zeros((3,3),float)
- >>> b = b + 2.*identity(3) #"+" is overloaded
- >>> c = a + b
- >>> print c
- [[3., 1., 1.],
- [1., 3., 1.],
- [1., 1., 3.]]

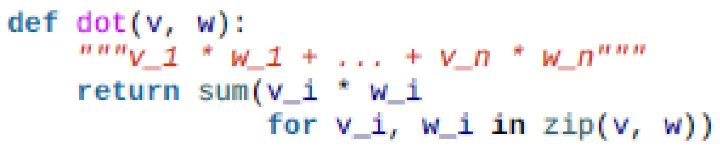
Array functions

- >>> from LinearAlgebra import *
- >>> a = zeros((3,3),float) + 2.*identity(3)
- >>> print inverse(a)
- [[0.5, 0., 0.],
- [0., 0.5, 0.],
- [0., 0., 0.5]]
- >>> print determinant(inverse(a))
- 0.125
- >>> print diagonal(a)
- [0.5,0.5,0.5]
- >>> print diagonal(a,1)
- [0.,0.]

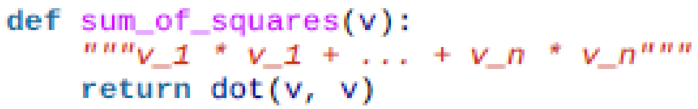
- transpose(a), argsort(), dot()

Vectors

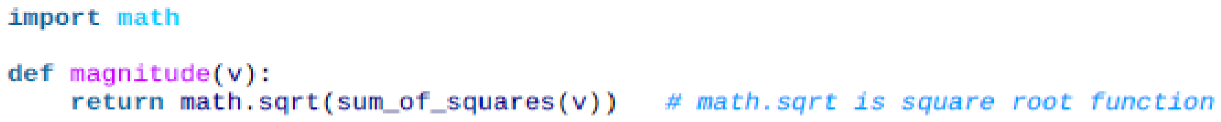
- The dot product of two vectors is the sum of their
- componentwise products:



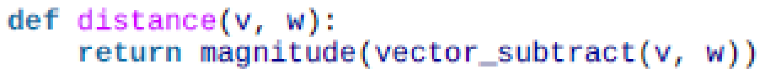
```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i
               for v_i, w_i in zip(v, w))
```



```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
```



```
import math

def magnitude(v):
    return math.sqrt(sum_of_squares(v))   # math.sqrt is square root function
```



```
def distance(v, w):
    return magnitude(vector_subtract(v, w))
```

Eigenvalues

- >>> from LinearAlgebra import *
- >>> val = eigenvalues(c)
- >>> val, vec = eigenvectors(c)
- >>> print val
- [1., 4., 1.]
- >>> print vec
- [[0.816, -0.408, -0.408],
- [0.575, 0.577, 0.577],
- [-0.324, -0.487, 0.811]]

- also solve_linear_equations, singular_value_decomposition, etc.

Least Squares Fitting

- Part of Hinsen's Scientific Python module
  - >>> from LeastSquares import *
  - >>> def func(params,x): # y=ax^2+bx+c
  - return params[0]*x*x + params[1]*x +
  - params[2]
  - >>> data = []
  - >>> for i in range(10):
  - data.append((i,i*i))
  - >>> guess = (3,2,1)
  - >>> fit_params, fit_error =
  - leastSquaresFit(func,guess,data)
  - >>> print fit_params
  - [1.00,0.000,0.00]

FFT

- >>> from FFT import *
- >>> data = array((1,0,1,0,1,0,1,0))
- >>> print fft(data).real
- [4., 0., 0., 0., 4., 0., 0., 0.]]

- Also note that the FFTW package ("fastest Fourier transform in the West") has a python wrapper. See notes at the end
- Python Standard Libraries/Modules:
  - http://docs.python.org/library/
  - http://its2.unc.edu/dci/dci_components/shared_apps/packages/python_packages.html
  - http://pypi.python.org/pypi/

Probability

- Normal distribution

$$f(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

```
def normal_pdf(x, mu=0, sigma=1):
    sqrt_two_pi = math.sqrt(2 * math.pi)
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (sqrt_two_pi * sigma))
```

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1)   for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("Various Normal pdfs")
plt.show()
```

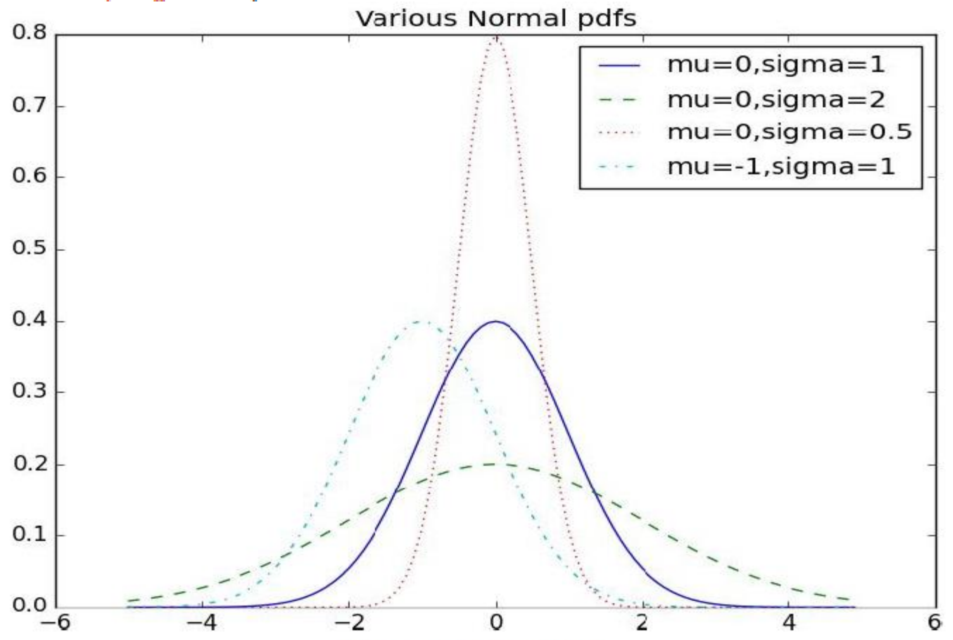
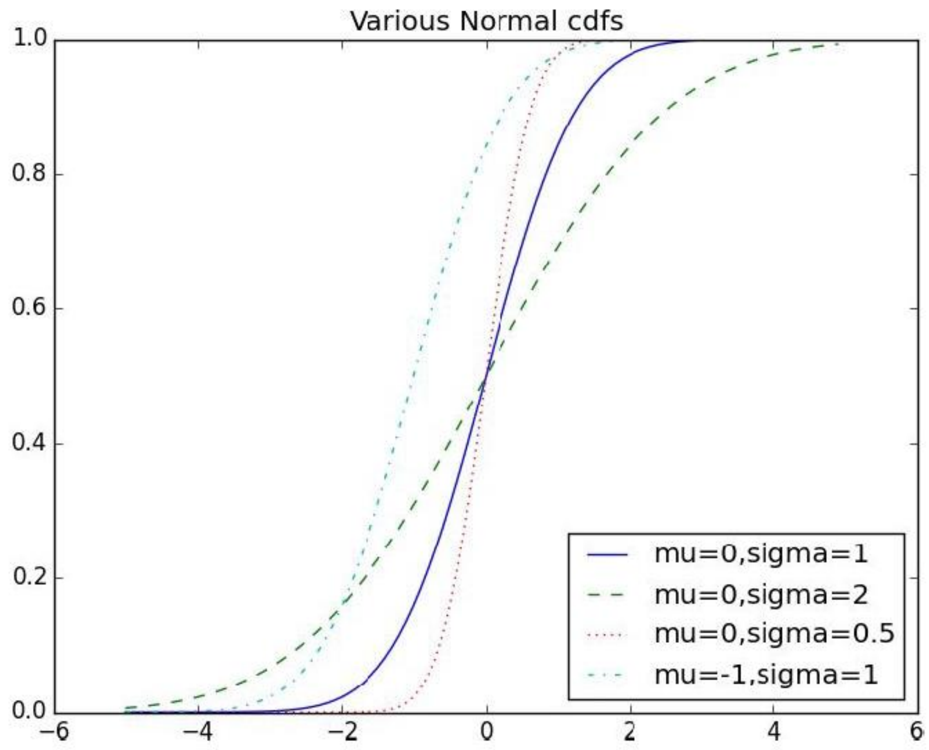


Figure 6-2. Various normal pdfs

CDF

- The **cumulative distribution function** for the normal distribution cannot be written in an “elementary” manner, but we can write it using Python’s **math.erf**.

```
def normal_cdf(x, mu=0,sigma=1):
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2

xs = [x / 10.0 for x in range(-50, 50)]

plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # bottom right
plt.title("Various Normal cdfs")
plt.show()
```



Sorting

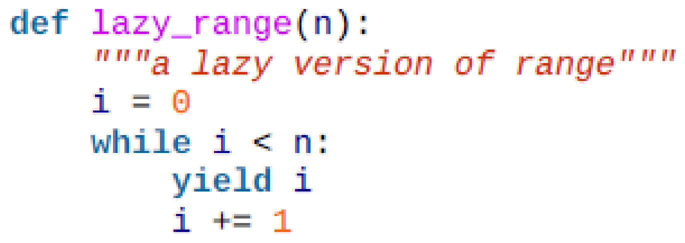
- Every Python list has a ***sort*** method that sorts it in place. If you don’t want to mess up your list, you can use the ***sorted*** function, which returns a new list:

```
x = [4,1,2,3]
y = sorted(x)      # is [1,2,3,4], x is unchanged
x.sort()           # now x is [1,2,3,4]
```

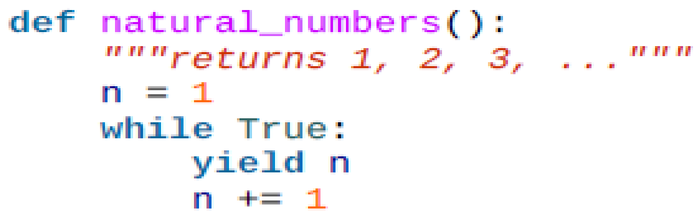
```
# sort the list by absolute value from largest to smallest
x = sorted([-4,1,-2,3], key=abs, reverse=True)  # is [-4,3,-2,1]

# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
            key=lambda (word, count): count,
            reverse=True)
```

Generators and Iterators

- A problem with lists is that they can easily grow very big. range(1000000) creates an actual list of 1 million elements. If you only need to deal with them one at a time, this can be a huge source of inefficiency (or of running out of memory).
- A generator is something that you can iterate over (for us, usually using for) but whose values are produced only as needed (lazily).
- One way to create generators is with functions and the yield operator:



```python
def lazy_range(n):
    """a lazy version of range"""
    i = 0
    while i < n:
        yield i
        i += 1
```



```python
def natural_numbers():
    """returns 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

- This means you could even create an infinite sequence:

Randomness

- As we learn data science, we will frequently need to generate random numbers, which we can do with the ***random*** module.

```
import random

four_uniform_randoms = [random.random() for _ in range(4)]

#  [0.8444218515250481,      # random.random() produces numbers
#   0.7579544029403025,      # uniformly between 0 and 1
#   0.420571580830845,       # it's the random function we'll use
#   0.25891675029296335]     # most often


random.seed(10)          # set the seed to 10
print random.random()    # 0.57140259469
```

Python Libraries for Data Science

Many popular Python toolboxes/libraries:

- NumPy
- SciPy
- Pandas
- SciKit-Learn

Data Science

Visualization libraries

- matplotlib
- Seaborn

and many more …

@Python-for-Data-Analysis.pptx

Python Libraries for Data Science

NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects

- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance

- many other python libraries are built on NumPy

Link:
http://www.numpy.org/

What is Numpy?

- Numpy, Scipy, and Matplotlib provide MATLAB-like functionality in python.
- Numpy Features:
  - Typed multidimentional arrays (matrices)
  - Fast numerical computations (matrix math)
  - High-level math functions
- Python does numerical computations slowly.
- 1000 x 1000 matrix multiply
  - Python triple loop takes > 10 min.
  - Numpy takes ~0.03 seconds

NumPy Overview

1. Arrays
2. Shaping and transposition
3. Mathematical Operations
4. Indexing and slicing
5. Broadcasting

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- **Vectors**
- **Matrices**
- Images
- Tensors
- ConvNets

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Arrays

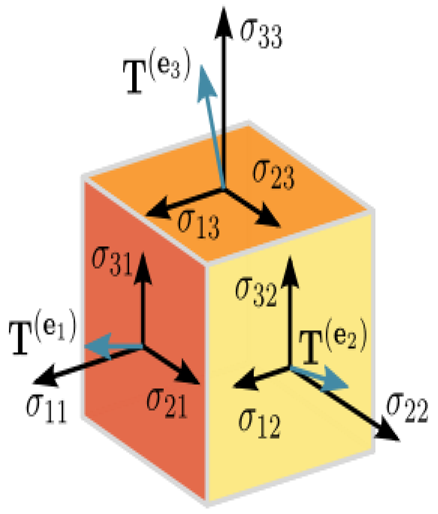
Structured lists of numbers.

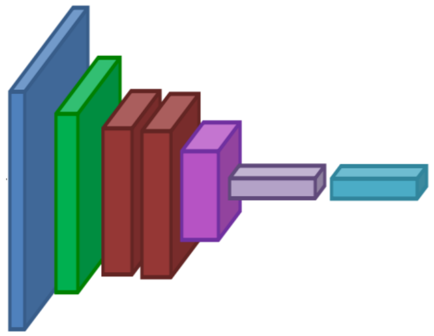
- Vectors
- Matrices
- **Images**
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- **Tensors**
- **ConvNets**





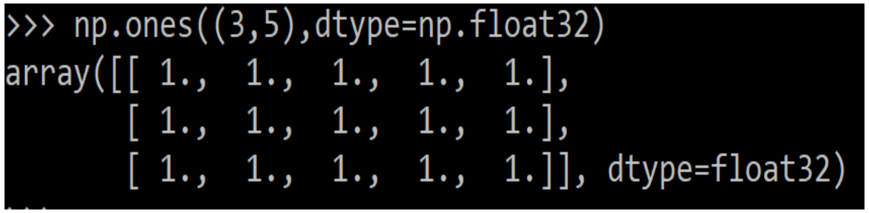
Arrays, Basic Properties

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]],dtype=np.float32)
print a.ndim, a.shape, a.dtype
```

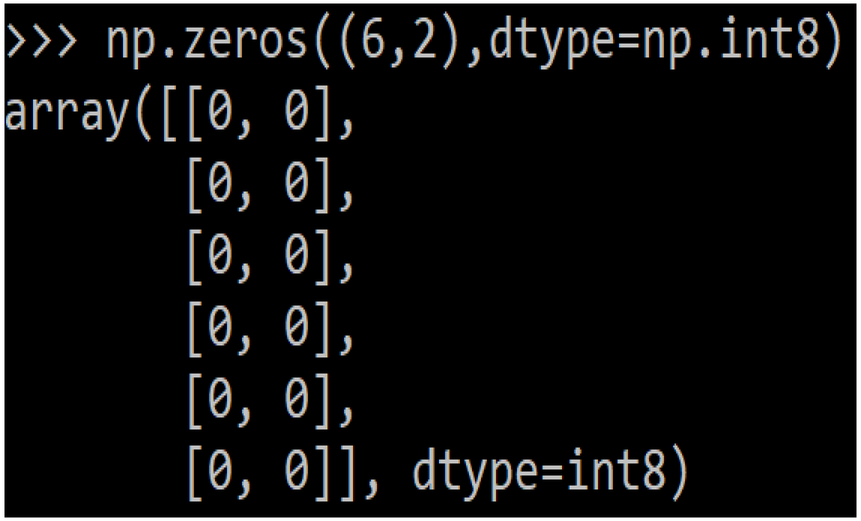
1. Arrays can have any number of dimensions, including zero (a scalar).
2. Arrays are typed: np.uint8, np.int64, np.float32, np.float64
3. Arrays are dense. Each element of the array exists and has the same type.

Arrays, creation

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random



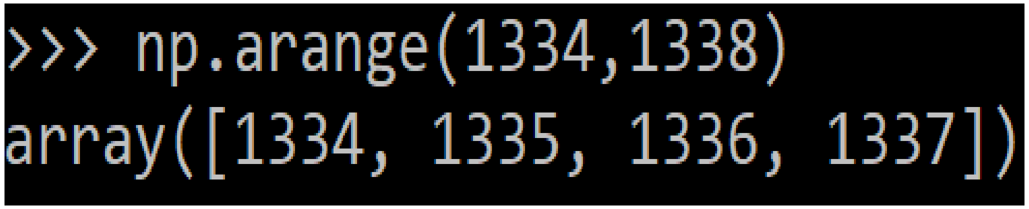
```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```



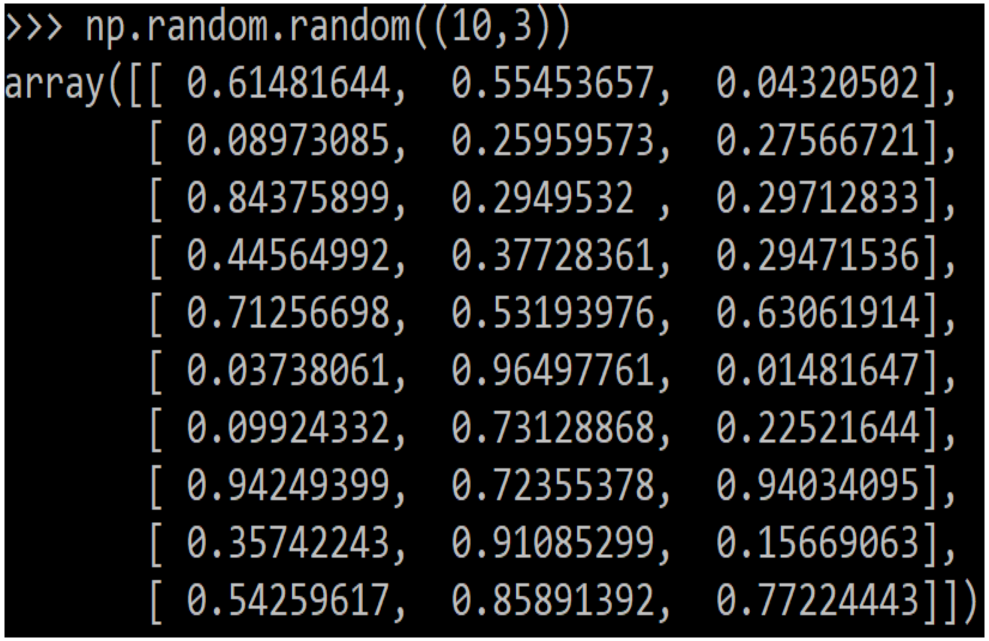
```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

Arrays, creation

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random



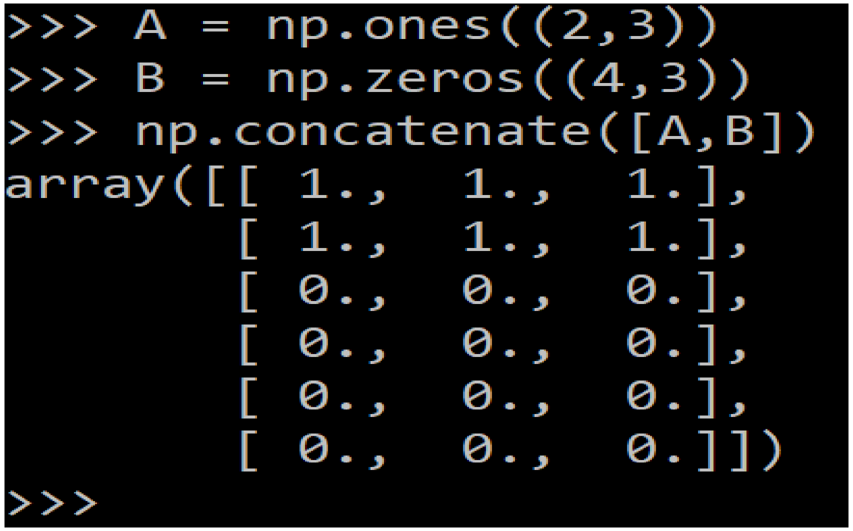
```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```



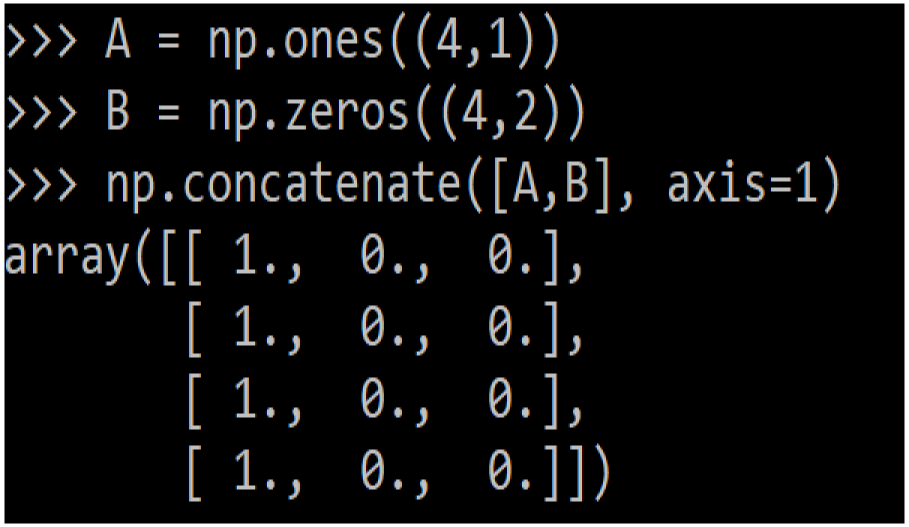
```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random



```
>>> A = np.ones((2,3))
>>> B = np.zeros((4,3))
>>> np.concatenate([A,B])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
```



```
>>> A = np.ones((4,1))
>>> B = np.zeros((4,2))
>>> np.concatenate([A,B], axis=1)
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

Saving and loading arrays

```
np.savez(‘data.npz’, a=a)
data = np.load(‘data.npz’)
a = data[‘a’]
```

1. NPZ files can hold multiple arrays
2. np.savez_compressed similar.

Saving and Loading Images

SciPy: `skimage.io.imread,skimage.io.imsave`

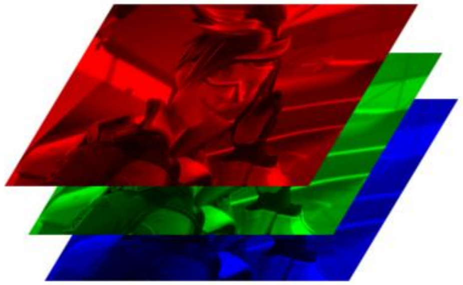
height x width x RGB

PIL / Pillow: `PIL.Image.open, Image.save`

width x height x RGB

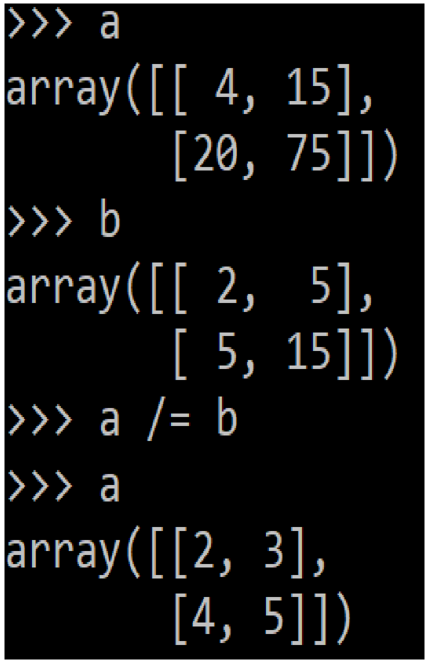
OpenCV: `cv2.imread, cv2.imwrite`

height x width x BGR

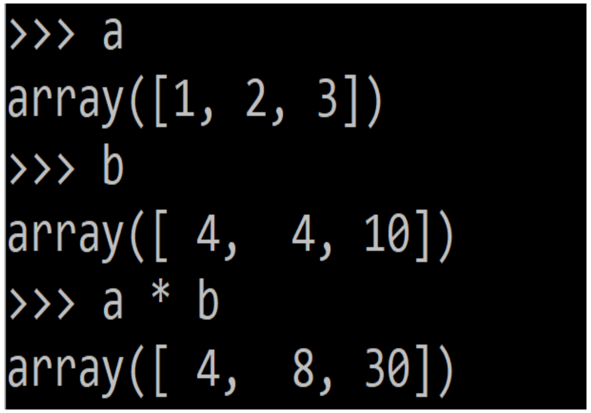


Mathematical operators

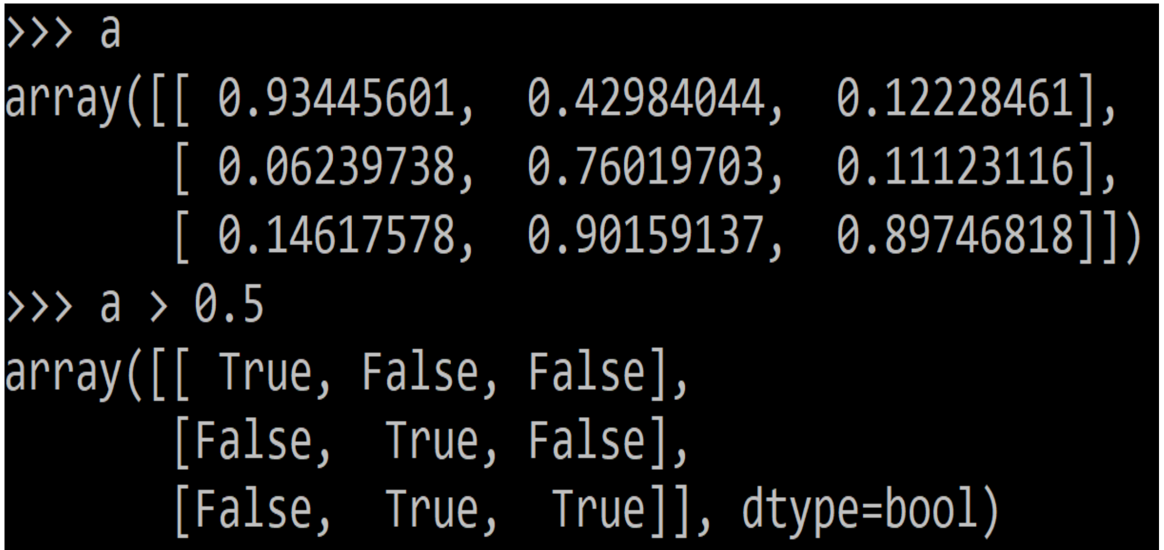
- **Arithmetic operations are element-wise**
- Logical operator return a bool array
- In place operations modify the array



```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```



```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```



```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

Math, upcasting

Just as in Python and Java, the result of a math operator is cast to the more general or precise datatype.

uint64 + uint16 => uint64

float32 / int32 => float32

Warning: upcasting does not prevent overflow/underflow. You must manually cast first.

Use case: images often stored as uint8. You should convert to float32 or float64 before doing math.

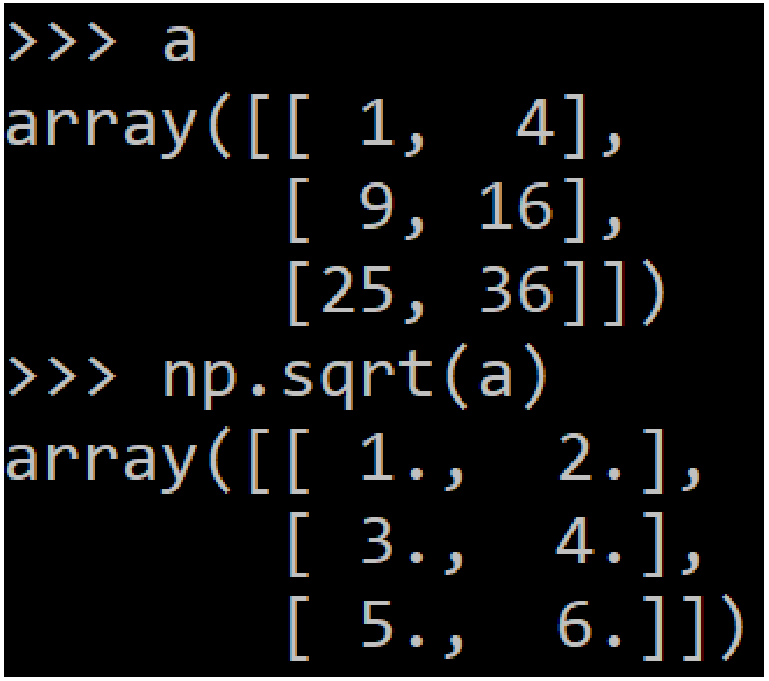
Math, universal functions

Also called ufuncs

Element-wise

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan



```
>>> a
array([[ 1,  4],
       [ 9, 16],
       [25, 36]])
>>> np.sqrt(a)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

Indexing

```
x[0,0]    # top-left element
x[0,-1]   # first row, last column
x[0,:]    # first row (many entries)
x[:,0]    # first column (many entries)
```

Notes:

- Zero-indexing
- Multi-dimensional indices are comma-separated (i.e., a tuple)

Python Slicing

Syntax: start:stop:step

```
a = list(range(10))
a[:3] # indices 0, 1, 2
a[-3:] # indices 7, 8, 9
a[3:8:2] # indices 3, 5, 7
a[4:1:-1] # indices 4, 3, 2 (this one
is tricky)
```

Axes

```
a.sum() # sum all entries
a.sum(axis=0) # sum over rows
a.sum(axis=1) # sum over columns
a.sum(axis=1, keepdims=True)
```

1. Use the axis parameter to control which axis NumPy operates on
2. Typically, the axis specified will disappear, keepdims keeps all dimensions

Broadcasting

```
a = a + 1 # add one to every element
```

When operating on multiple arrays, broadcasting rules are used.

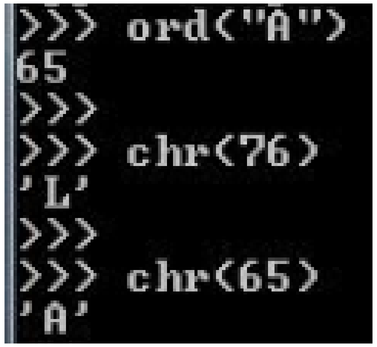
Each dimension must match, from right-to-left

1. Dimensions of size 1 will broadcast (as if the value was repeated).
2. Otherwise, the dimension must have the same shape.
3. Extra dimensions of size 1 are added to the left as needed.

Strings and formatting

- Python string is a contiguous sequence of Unicode characters. Strings - characters are strings of length 1

```
i = 10

d = 3.1415926

s = "I am a string!"

print "%d\t%f\t%s" % (i, d, s)

print “newline\n"

print "no newline“
```



```
>>> ord("A")
65
>>>
>>> chr(76)
'L'
>>>
>>> chr(65)
'A'
```

- In order to convert a character value to ASCII code, the ord() function is used, and for converting ASCII code to character, the chr() function is used

Strings

"hello"+"world"	"helloworld"	# concatenation
"hello"*3	"hellohellohello"	# repetition
"hello"[0]	"h"	# indexing
"hello"[-1]	"o"	# (from end)
"hello"[1:4]	"ell"	# slicing
len("hello")	5	# size
"hello" < "jello"	1	# comparison
"e" in "hello"	1	# search
New line:	"escapes: \n "	
Line continuation:	triple quotes ””	
Quotes:	‘single quotes’, "raw strings"	

Methods in string

- upper()
- lower()
- capitalize()
- count(s)
- find(s)
- rfind(s)
- index(s)

- strip(), lstrip(), rstrip()
- replace(a, b)
- expandtabs()
- split()
- join()
- center(), ljust(), rjust()

Language comparison

		Tcl	Perl	Python	JavaScript	Visual Basic
Speed	development	✓	✓	✓	✓	✓
	regexp	✓	✓	✓		
breadth	extensible	✓		✓		✓
	embeddable	✓		✓		
	easy GUI	✓		✓ (Tk)		✓
	net/web	✓	✓	✓	✓	✓
enterprise	cross-platform	✓	✓	✓	✓	
	I18N	✓		✓	✓	✓
	thread-safe	✓		✓		✓
	database access	✓	✓	✓	✓	✓

Python: Pros & Cons

- Pros
  - Free availability (like Perl, Python is open source).
  - Stability (Python is in release 2.6 at this point and, as I noted earlier, is older than Java).
  - Very easy to learn and use
  - Good support for objects, modules, and other reusability mechanisms.
  - Easy integration with and extensibility using C and Java.
- Cons
  - Smaller pool of Python developers compared to other languages, such as Java
  - Lack of true multiprocessor support
  - Absence of a commercial support point, even for an Open Source project (though this situation is changing)
  - Software performance slow, not suitable for high performance applications