Introduction to Python

Sunil Gorantla

About me

11+ Years of Experience in building AI/ML solutions.

Currently working for one of the big 4 consulting companies as data architect.

Alumni of IIT Patna and ISB Alumni

Expertise include data analytics, AI/ML, NLP.

About Python

Python is an interpreted, high-level, general-purpose programming language.

Created by Guido van Rossum, 1991.

Dynamic type , memory management.

Object-oriented and procedural.

Designed to be highly extensible.

Comprehensive standard library.

Python Software Foundation.

Bit of History

Python 2.0 → October 2000

Python 3.0 → 3 December 2008

British group Monty Python.

Top ten – 2013

Used in

- Wikipedia, Google, CERN, NASA, Facebook, Amazon, Reddit 100%
- web applications, scientific computing, Raspberry Pi single-board computer

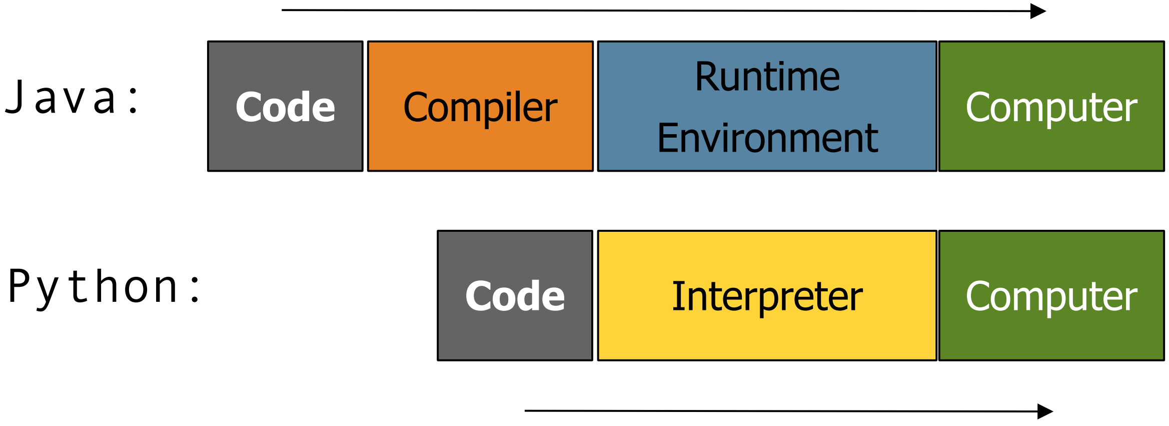
django

pandas

$y_it = \beta' x_{it} + \mu_i + \epsilon_{it}$

interpreted

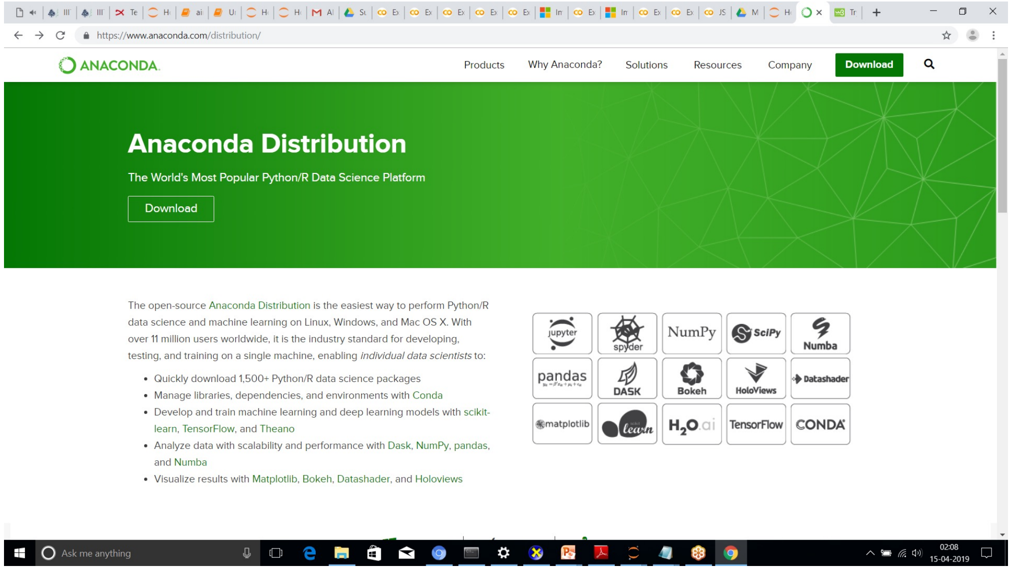
- Not compiled like Java
- Code is written and then directly executed by an **interpreter**
- Type commands into interpreter and see immediate results



Java: Code → Compiler → Runtime Environment → Computer

Python: Code → Interpreter → Computer

Installing Python



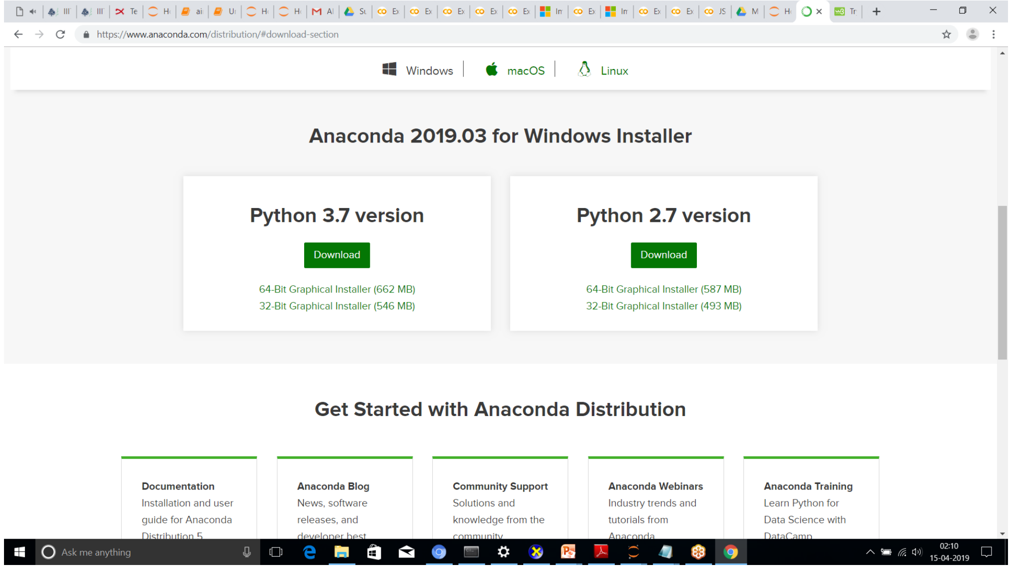
Anaconda Distribution

The World's Most Popular Python/R Data Science Platform

The open-source Anaconda Distribution is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X. With over 11 million users worldwide, it is the industry standard for developing, testing, and training on a single machine, enabling *individual data scientists* to:

- Quickly download 1,500+ Python/R data science packages
- Manage libraries, dependencies, and environments with Conda
- Develop and train machine learning and deep learning models with scikit-learn, TensorFlow, and Theano
- Analyze data with scalability and performance with Dask, NumPy, pandas, and Numba
- Visualize results with Matplotlib, Bokeh, Datashader, and Holoviews

Anaconda



Anaconda 2019.03 for Windows Installer

Python 3.7 version

64-Bit Graphical Installer (662 MB)
32-Bit Graphical Installer (546 MB)

Python 2.7 version

64-Bit Graphical Installer (587 MB)
32-Bit Graphical Installer (493 MB)

Get Started with Anaconda Distribution

Documentation
Installation and user guide for Anaconda Distribution 5

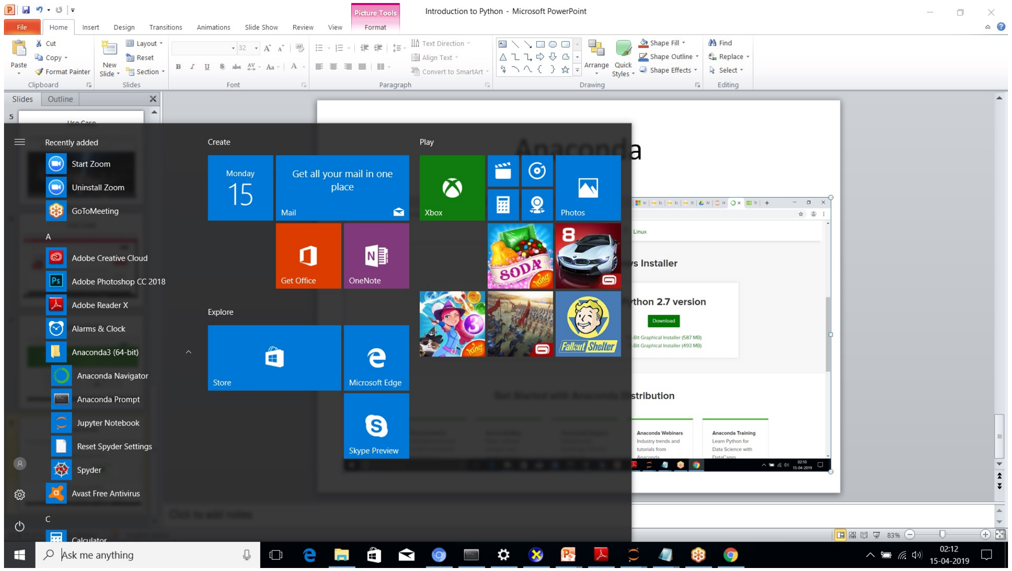
Anaconda Blog
News, software releases, and developer best

Community Support
Solutions and knowledge from the community

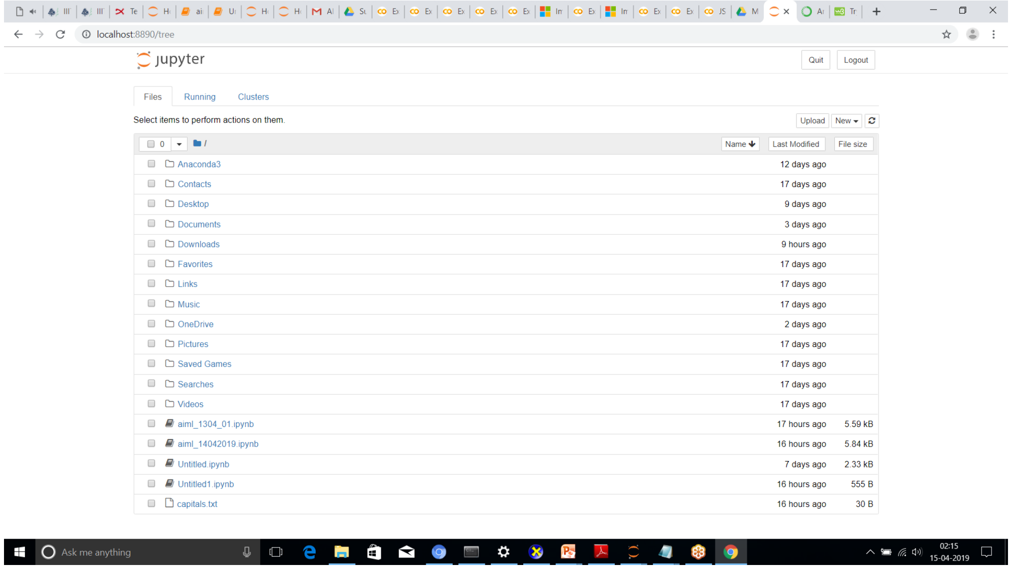
Anaconda Webinars
Industry trends and tutorials from Anaconda

Anaconda Training
Learn Python for Data Science with DataCamp

Starting Python



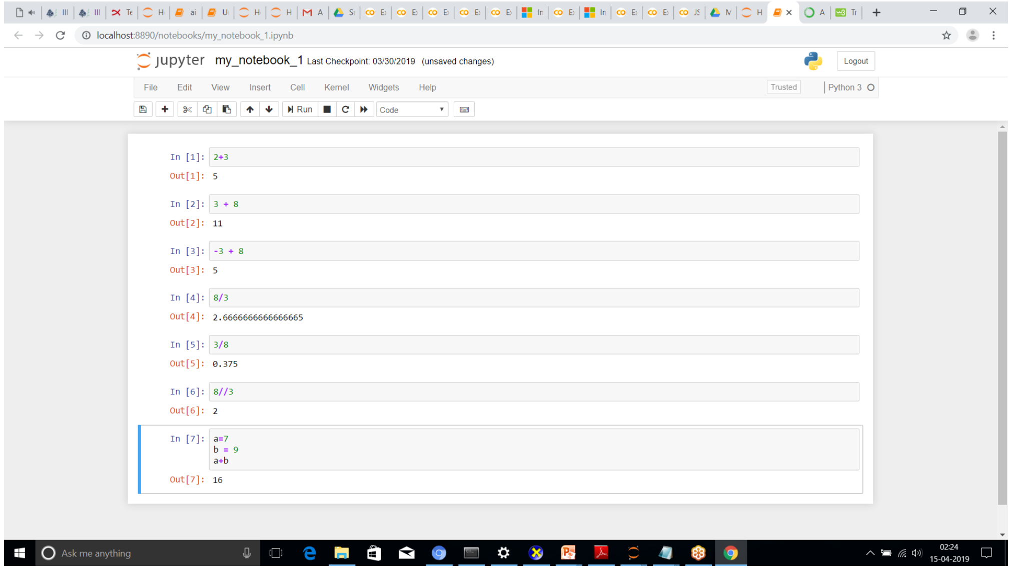
Jupyter Notebook



Select items to perform actions on them.

Name	Last Modified	File size
Anaconda3	12 days ago	
Contacts	17 days ago	
Desktop	9 days ago	
Documents	3 days ago	
Downloads	9 hours ago	
Favorites	17 days ago	
Links	17 days ago	
Music	17 days ago	
OneDrive	2 days ago	
Pictures	17 days ago	
Saved Games	17 days ago	
Searches	17 days ago	
Videos	17 days ago	
aiml_1304_01.ipynb	17 hours ago	5.59 kB
aiml_14042019.ipynb	16 hours ago	5.84 kB
Untitled.ipynb	7 days ago	2.33 kB
Untitled1.ipynb	16 hours ago	555 B
capitals.txt	16 hours ago	30 B

First Interaction


localhost:8890/notebooks/my_notebook_1.ipynb
jupyter my_notebook_1 Last Checkpoint: 03/30/2019 (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code
Ask me anything
02:24
15-04-2019

```
In [1]: 2+3
Out[1]: 5

In [2]: 3 + 8
Out[2]: 11

In [3]: -3 + 8
Out[3]: 5

In [4]: 8/3
Out[4]: 2.6666666666666665

In [5]: 3/8
Out[5]: 0.375

In [6]: 8//3
Out[6]: 2

In [7]: a=7
        b = 9
        a+b
Out[7]: 16
```

The Basics

A Code Sample (in IDLE)

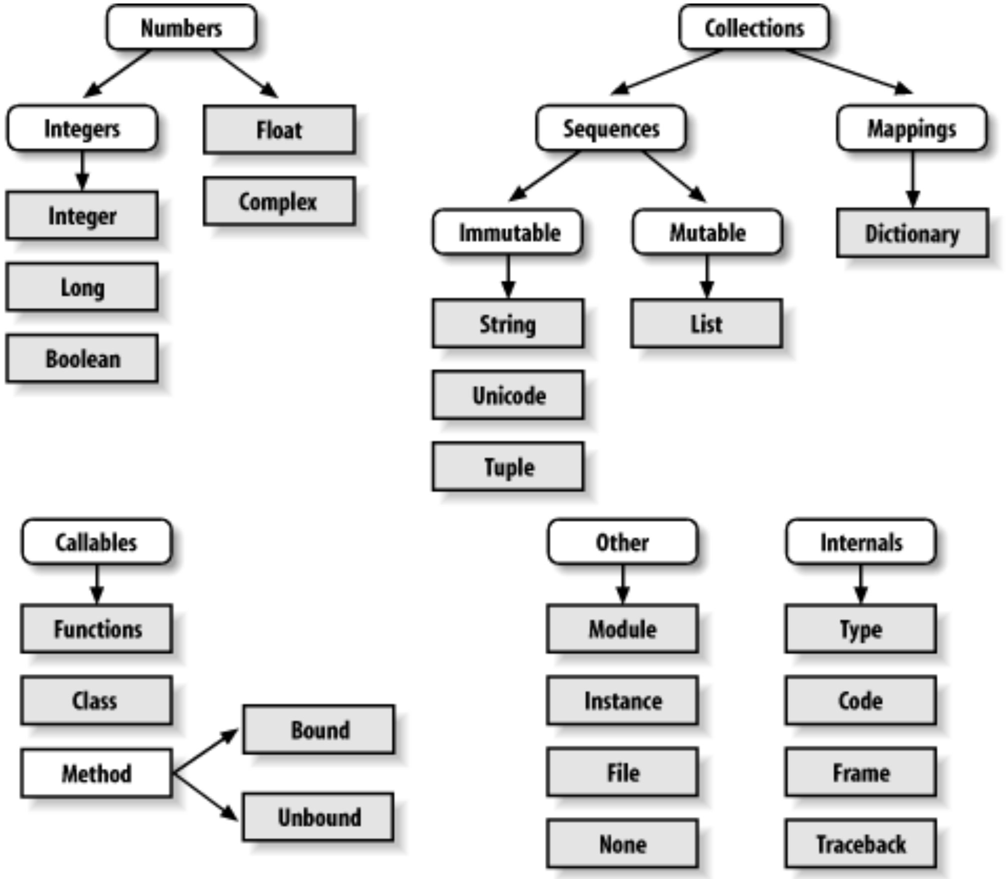
```
x = 34 - 23             # A comment.
y = “Hello”             # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”    # String concat.
print x
print y
```

Python’s data types

Everything is an object

- Python data is represented by objects or by relations between objects
- Every object has an identity, a type and a value
- Identity never changes once created Location or address in memory
- Type (e.g., integer, list) is unchangeable and determines the possible values it could have and operations that can be applied
- Value of some objects is fixed (e.g., an integer) and can change for others (e.g., list)

Python’s built-in type hierarchy



Scalar Data Types

Dynamically typed ⇔ Strongly typed

No variable declaration

```
a = 17
b = ‘Hello’
a, b = b, ‘World’
c = a + b
```

- Integers
- Floats
- Booleans
- Complex

Data Types - Integers

- **Integers**
- No limit
- Usual operators: + PLUS, - MINUS, * STAR and % PERCENTAGE
- Different operators:
  - / SLASH for floating division
  - // DOUBLE SLASH for truncating division
  - ** DOUBLE STAR for exponentiation
- assignment operators
  - $x \odot= y \Rightarrow x = x \odot y$

Data Types - Floats

- **Floats**
- Same as C double!
- With the usual troubles

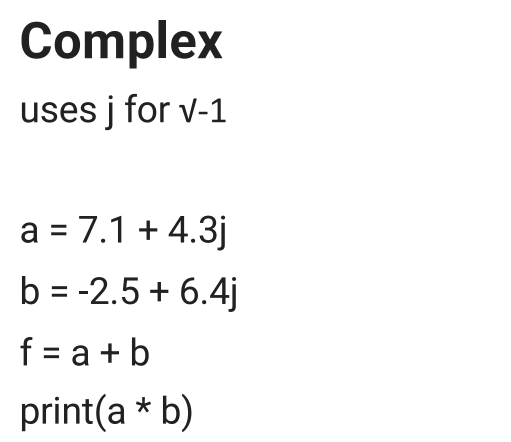
- x = 12.9
- y = 3.6
- print(x * y)
- 46.440000000000005

Booleans

True,False

Usual operators:

    ==, !=, <, <=, >, >=



Complex

uses j for √-1

```
a = 7.1 + 4.3j
b = -2.5 + 6.4j
f = a + b
print(a * b)
```

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
  facorial of n.”””
  assert(n>0)
  return 1 if n==1 else n*fact(n-1)
```

Assignment

- *Binding a variable* in Python means setting a *name* to hold a *reference* to some *object*
    - *Assignment creates references, not copies*
- Names in Python do not have an intrinsic type, objects have types
    - Python determines the type of the reference automatically based on what data is assigned to it
- You create a name the first time it appears on the left side of an assignment expression:

```
x = 3
```

- A reference is deleted via garbage collection after any names bound to it have passed out of scope
- Python uses *reference semantics* (more later)

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 recommend-ed 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

Assignment

- You can assign to multiple names at the same time

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

This makes it easy to swap values

```
>>> x, y = y, x
```

- Assignments can be chained

```
>>> a = b = x = 2
```

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
```

Sequence types: Tuples, Lists, and Strings

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 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' 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'
```

Mutability: Tuples vs. Lists

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.*

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
```

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()
  ()
  ```

Summary: Tuples vs. Lists

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