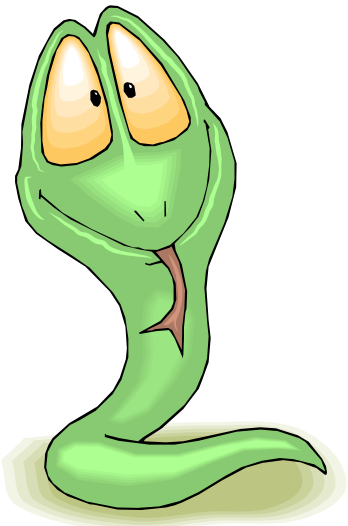
Intro to Python



Overview

- History
- Installing & Running Python
- Names & Assignment
- Sequences types: Lists, Tuples, and Strings
- Mutability
- Dictionaries
- Functions
- Logical expressions
- Flow of control
- For loops
- More on functions
- Assignment and containers
- Strings

Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning
- Considered a scripting language, but is much more
- Scalable, object oriented and functional from the beginning
- Used by Google from the beginning
- Increasingly popular

Python's Benevolent Dictator For Life

"Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered."

- Guido van Rossum

http://docs.python.org/

Download

Download these documents

Docs by version

Python 3.10 (in development)
Python 3.9 (stable)
Python 3.8 (stable)
Python 3.7 (security-fixes)
Python 3.6 (security-fixes)
Python 3.5 (EOL)
Python 2.7 (EOL)
All versions

Other resources

PEP Index
Beginner's Guide
Book List
Audio/Visual Talks
Python Developer's Guide

Python 3.9.1 documentation

Welcome! This is the documentation for Python 3.9.1.

Parts of the documentation:

What's new in Python 3.9?
or all "What's new" documents since 2.0

Tutorial
start here

Library Reference
keep this under your pillow

Language Reference
describes syntax and language elements

Python Setup and Usage
how to use Python on different platforms

Python HOWTOs
in-depth documents on specific topics

Installing Python Modules
installing from the Python Package Index & other sources

Distributing Python Modules
publishing modules for installation by others

Extending and Embedding
tutorial for C/C++ programmers

Python/C API
reference for C/C++ programmers

FAQs
frequently asked questions (with answers!)

The Python tutorial is good!

Previous topic
Changelog

Next topic
1. Whetting Your Appetite

This Page
Report a Bug
Show Source

The Python Tutorial

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

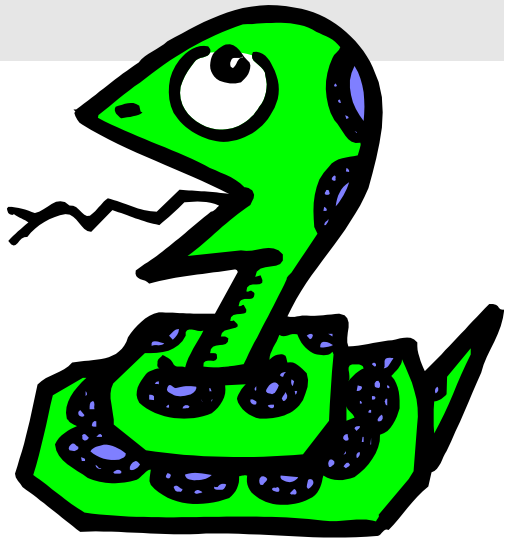
The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

For a description of standard objects and modules, see The Python Standard Library. The Python Language Reference gives a more formal definition of the language. To write extensions in C or C++, read Extending and Embedding the Python Interpreter and Python/C API Reference Manual. There are also several books covering Python in depth.

Running Python



The Python Interpreter

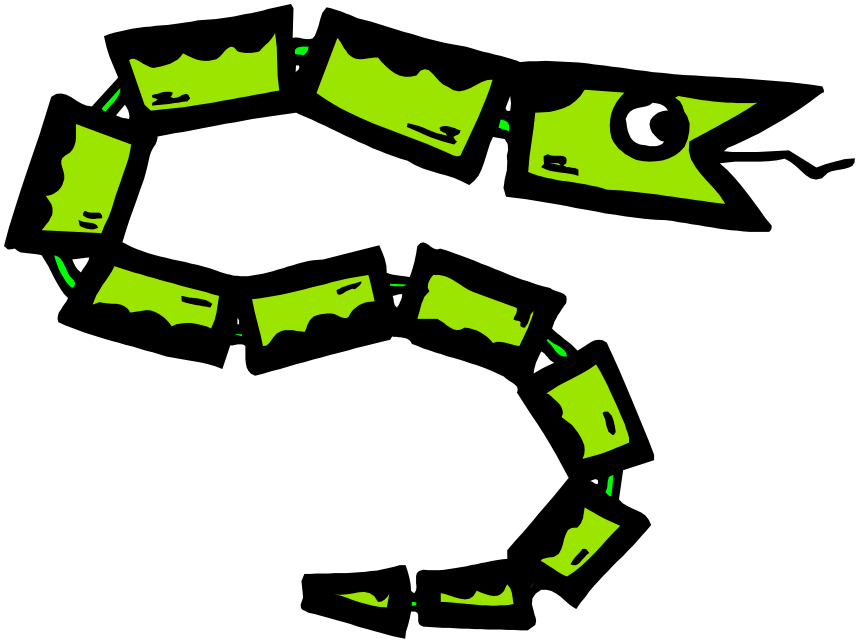
- Typical Python implementations offer both an interpreter and compiler
- Interactive interface to Python with a read-eval-print loop

```
[finin@linux2 ~]$ python
Python 2.4.3 (#1, Jan 14 2008, 18:32:40)
[GCC 4.1.2 20070626 (Red Hat 4.1.2-14)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def square(x):
...     return x * x
...
>>> map(square, [1, 2, 3, 4])
[1, 4, 9, 16]
>>>
```

Installing

- Python is pre-installed on most Unix systems, including Linux and MAC OS X
- The pre-installed version may not be the most recent one
- Download from http://python.org/download/
- Python comes with a large library of standard modules
- There are several options for an IDE
  - IDLE – works well with Windows
  - Emacs with python-mode or your favorite text editor
  - Eclipse with Pydev (http://pydev.sourceforge.net/)
- Our choice: Visual Studio Code and PyCharm
- Python 3.1 and up

The Basics



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 2.5, 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

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
```

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 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

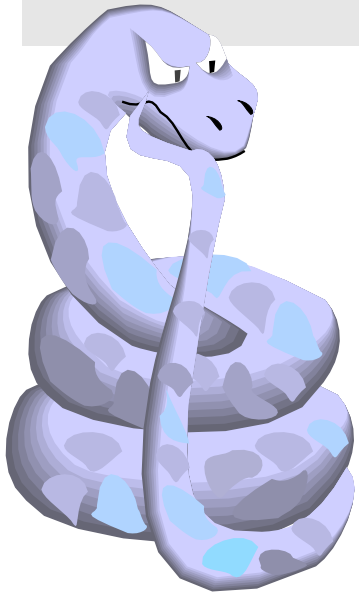
- The * operator produces a *new* tuple, list, or string that "repeats" the original content.

```
>>> (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'
```

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 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)
```

Dictionaries: A *Mapping* type

- Dictionaries store a *mapping* between a set of keys and a set of values
  - Keys can be any *immutable* type.
  - Values can be any type
  - A single dictionary can store values of different types
- You can define, modify, view, lookup or delete the key-value pairs in the dictionary
- Python's dictionaries are also known as *hash tables* and *associative arrays*

Creating & accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user']
'bozo'
>>> d['pswd']
1234
>>> d['bozo']
Traceback (innermost last):
  File '<interactive input>' line 1, in
  ?
KeyError: bozo
```

Updating Dictionaries

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

- Keys must be unique
- Assigning to an existing key replaces its value

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

- Dictionaries are unordered
  - New entries can appear anywhere in output
- Dictionaries work by *hashing*

Removing dictionary entries

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

>>> a=[1,2]
>>> del a[1]       # del works on lists, too
>>> a
[1]
```

Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()  # List of keys, VERY useful
['user', 'p', 'i']

>>> d.values() # List of values
['bozo', 1234, 34]

>>> d.items()  # List of item tuples
[('user','bozo'), ('p',1234), ('i',34)]
```