Tools Seminar

Week 5 - Python Basics

Hongzheng Chen

Dec 13, 2019

Why Python?

Dec 2019 Ranking

Nov 2019	Nov 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.246%	-0.50%
2	2		C	16.037%	+1.64%
3	4	▲	Python	9.842%	+2.16%
4	3	▼	C++	5.605%	-2.68%
5	6	▲	C#	4.316%	+0.36%
6	5	▼	Visual Basic .NET	4.229%	-2.26%
7	7		JavaScript	1.929%	-0.73%
8	8		PHP	1.720%	-0.66%
9	9		SQL	1.690%	-0.15%
10	12	▲	Swift	1.653%	+0.20%

Source: https://www.tiobe.com/tiobe-index/

Has become a necessary tool for partitioners in CS-related areas

Why Python is so hot?

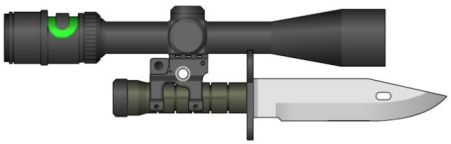
Extremely easy to use (programmability)!

So many applications/packages!

- Scripting/Glue Language:
  - Provide lots of interface (os, network, database, ...)
  - Connect different components
- Deep Learning: Tensorflow, Pytorch, MXNet
- Machine Learning: sklearn, scipy
- Scientific Computing: numpy, numba, matplotlib
- Data Mining: beautifulsoup
- Networking: socket, gRPC
- Website: Flask, Django
- Metalanguage: Python $\rightarrow$ LaTeX
- ...

Introductory languages for top CS schools

Why Python?

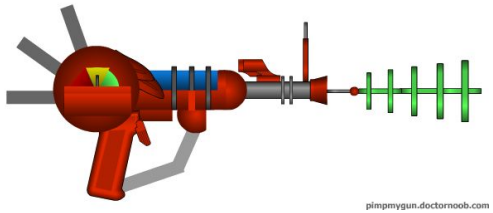


Assembly

C

C++

Java



Haskell

Python

Source: https://www.zhihu.com/question/25038841/answer/44396770

Why learn another programming language (PL)?

You have to find the best tool to finish one job.

Different languages may

- Be efficient in specific domains (why don't you write assembly?)
- Have specific features making it unique

Why learn another programming language (PL)?

You have to find the best tool to finish one job.

Different languages may

- Be efficient in specific domains (why don't you write assembly?)
- Have specific features making it unique

Thus, learning another PL helps to

- See key components across different programming languages
- See what paradigms certain languages prioritize
- Get deep into the software foundation & computer systems

Python official support

Official website: https://www.python.org/

- Python 2.7: ✗Jan 1, 2020
- Python 3.6: ✓most commonly used one
- Python 3.8: The newest version

All versions open-source and free

- Windows: Anaconda
- Unix: Initial support (minimum effort!)

* Package management: pip

How Python works?

- C/C++: Compilative languages (4 stages)
- Java/Python: **Interpretive** language (3 stages)

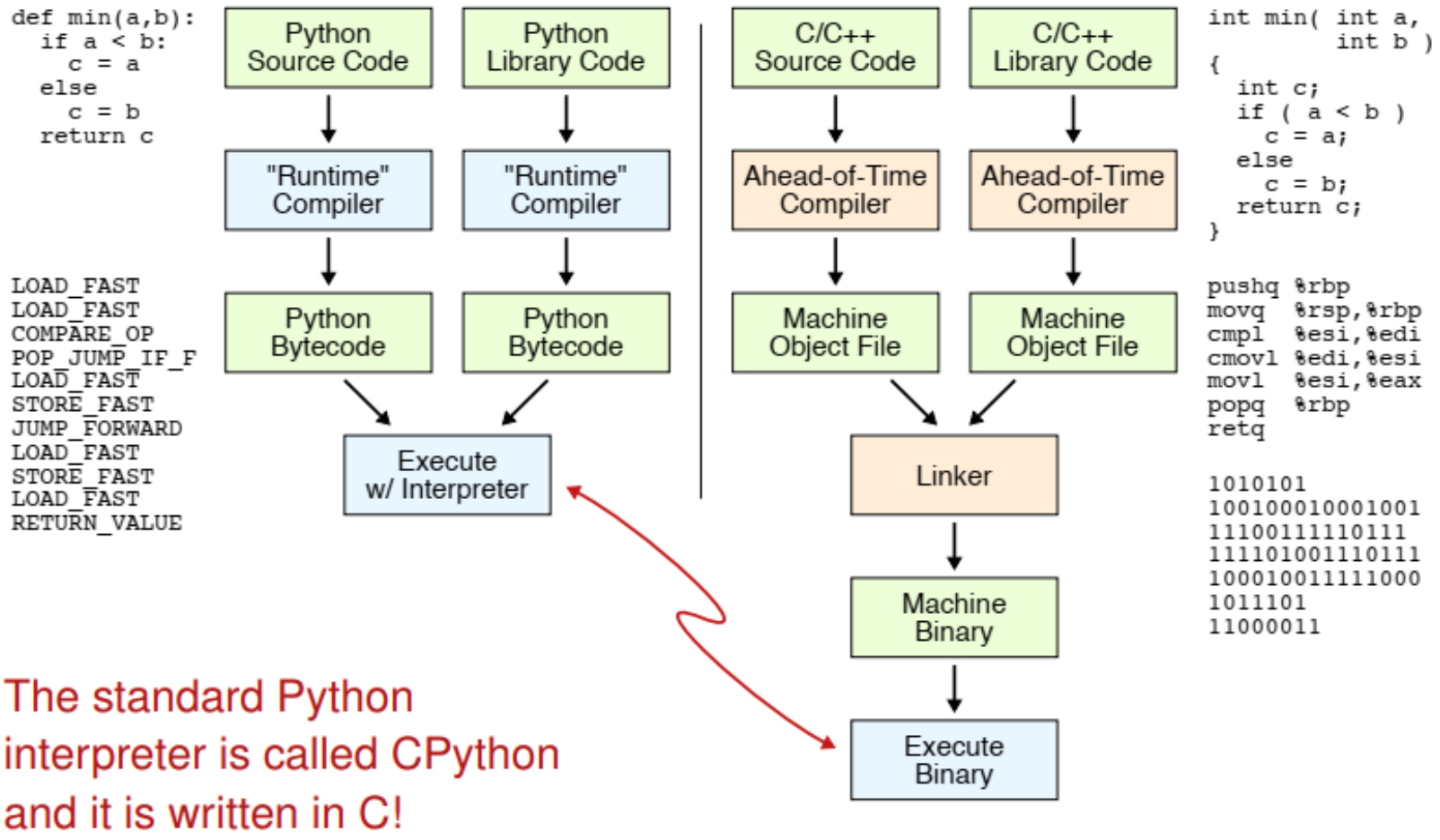
How Python works?

- C/C++: Compilative languages (4 stages)
- Java/Python: **Interpretive** language (3 stages)

Source code		**Byte code**		Runtime
`m.py`	$\rightarrow$	`m.pyc`	$\rightarrow$	Python Virtual Machine (PVM)

Byte code is independent of machines

Dynamically Interpreted vs. Statically Compiled



Source: Christopher Batten, *Cornell ECE 2400: Computer Systems Programming*, Fall 2019

2

Python Basics

So what does "interpret" means?

Just type "`python`" in your Unix cmd

- You need not write a file first and compile it
- But type something, it will **interactively** return you results!

Object, Everything is Object!!!

- No `int`, `uint`, `long`, etc.

Object, Everything is Object!!!

- No `int`, `uint`, `long`, etc.
- Python is dynamic type!

Object, Everything is Object!!!

- No `int`, `uint`, `long`, etc.
- Python is dynamic type!
- Everything in python is an object!

Object, Everything is Object!!!

- No `int`, `uint`, `long`, etc.
- Python is dynamic type!
- Everything in python is an object!
- Primitive object types

Number	`1234, 3.1415, 3+4j, Fraction`
String	`'spam', "bob's"` (similar to a list)
List	`[1,[2,three],4]` (mutable)
Tuple	`(1,'spam',4,'U')` (immutable!)
Dictionary	`{'food':'apple','taste':'yum'}`
Set	`set('a,b,c'), {'a','b','c'}`

Basic Operations

Arithmetic: `+`, `-`, `*`, `/`, `//`, `**`

- Primitive high-accuracy support
- Polymorphism (primitive operator reloading)
  - `[1,2,3] + [4,5]`
  - `[0] * 10`

Basic Operations

Arithmetic: `+`, `-`, `*`, `/`, `//`, `**`

- Primitive high-accuracy support
- Polymorphism (primitive operator reloading)
  - `[1,2,3] + [4,5]`
  - `[0] * 10`

`import math` / `from math import *`

- `floor(x)`
- `sqrt(x)`
- `exp(x)`
- `pow(x,y)`
- `factorial(x)`
- `log(x[,a])`
- `cos(x)`
- `pi`, `e`, `inf`, `nan` (Not a Number)

Lists

- Index starts from 0
- `len()`, `help()`
- Reverse indexing: `arr[-1]`
- Slicing:
  - `arr[a:b]`, $[a, b)$
  - `arr[:b]`, $[0, b)$
  - `arr[a:]`, $[a, len(arr))$
  - `arr[a:b:step]`
- `range(a,b,step)`: Python is such **lazy**!
- `list()`

Lists

Syntactic sugar

```
// A C/C++ Example
for ( int i = 0; i < y; i = i+1 ) {
  z = z + x;
}

{
  int i = 0;          // initialization statement
  while ( i < y ) { // conditional expression
    z = z + x;
    i = i + 1;        // increment statement
  }
}
```

* Example from ECE 2400

Lists

- `L.append`
- `L.insert`
- `L.pop`
- `L.reverse`
- `L.index`
- `L.remove`
- `L.copy` (shallow) / `copy.deepcopy(L)` (deep)
- `L.sort(reverse=False)`

String

- `strcmp` $\Longleftrightarrow$ `S1 == S2`
- `substr` $\Longleftrightarrow$ slicing
- `strstr` $\Longleftrightarrow$ `S.find`
- `ord` $\Longleftrightarrow$ `S.ord`
- `tolower` $\Longleftrightarrow$ `S.lower`
- `S.split`
- `S.replace`
- `S.join`

String formatting

- `print(a,b,c,sep=" ",end="\n")`
- `print(*[a,b,c])`, unpack the arguments
- `"The number is {}".format(a)`
- `"{0} + {1} = {0} + {1}".format(a,b)`
- `"{:.2f}.format(3.1415926)"`
- `"{:0>2d}".format(5)`

For more, please refer to
http://blog.xiayf.cn/2013/01/26/python-string-format/

List Comprehension

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

List Comprehension

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

- `[2*x for x in range(N) if x**2 > 3]`

List Comprehension

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

- `[2*x for x in range(N) if x**2 > 3]`
- `(2*x for x in itertools.count() if x**2 > 3), next()`

List Comprehension

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

- `[2*x for x in range(N) if x**2 > 3]`
- `(2*x for x in itertools.count() if x**2 > 3), next()`
- You can filter out prime numbers in such a concise way!

```
>>> [x for x in range(2,N)
...    if all(x % y != 0 for y in range(2,x))]
```

List Comprehension

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

- `[2*x for x in range(N) if x**2 > 3]`
- `(2*x for x in itertools.count() if x**2 > 3), next()`
- You can filter out prime numbers in such a concise way!

```
>>> [x for x in range(2,N)
...    if all(x % y != 0 for y in range(2,x))]
```

* This is some kind of functional programming (FP), e.g. Haskell. Many FP concepts in Python are borrowed from Haskell.

Pattern matching

```
>>> seq = [1,2,3,4]
>>> a, b, c, d = seq
1 2 3 4
>>> _, b, c, _ = seq # anonymous variables
2 3
>>> a, b = b, a # swap
3 2
>>> head, *tail = seq # unpack
1 [2, 3, 4]
>>> *init, last = seq
[1, 2, 3], 4
```

* Comments use #

Dictionary

Key-Value storing

```
D = {'food': 'apple', 'quantity': 4, 'color':['red', 'yellow']}
```

- `D['food']`
- `D.get(sth,0)`

Set

A fast way to remove replica

- `set([1,2,1,2,3,4,3,2])`
- `&` (intersection), `|` (union), `-` (difference)

Python's paradigm

- Python is a multi-paradigm programming language (IP, FP, OOP, etc.)
- Use **indentation** to distinguish commands (strictly)
- You can add ;, but it is no need
- **Control** is the feature of Imperative Programming (IP)

Condition

- `and`, `or`, `not`, **in** (even have **not in**!)
- `True`, `False`
- `if ... elif ... else ...`
  - No need for `()`
  - **No switch!**
- `<condition> ? <var1> : <var2>` $\Longleftrightarrow$
  `<var1> if <condition> else <var2>`
- `a <= b < c`

Condition

```
if x:
    if y:
        statement1
else:
    statement2
```

* Distinguish from C/C++

Loop

```
# for loop
for item in container: # compare to C++11 for-range for(auto x:
    ↪ lst)
    if search_something(item):
        # found it
        process(item)
        break
else:
    not_found_in_container()

# while loop
while value < threshold:
    value = update(value)
else: # value >= threshold
    handle_threshold_reached()
```

- break, continue, pass, but no goto!
- * List comprehension may be faster than naive loop

Generator / Iterator

Lazy evaluation:

```
>>> L = [1,2,3]
>>> I = iter(L) # obtain an iterator object
>>> I.next()
1
>>> I.next()
2
```

Generator / Iterator

- `range(N)`
- `enumerate(L)`
- `zip(L1,L2)`

* Or use `yield` to write your own generator

Functions

```
# No types are needed! (Extremely dynamic)
def foo(x,y,flag=False): # default parameter
    return (x + y if flag else x - y)

foo(1,0)
foo(1,1,True)
foo(1,2,flag=True) # keyword parameter
foo("a","b") # polymorphism
```

- Can be in any place, even in `if-else`
- Only need to be defined before calling

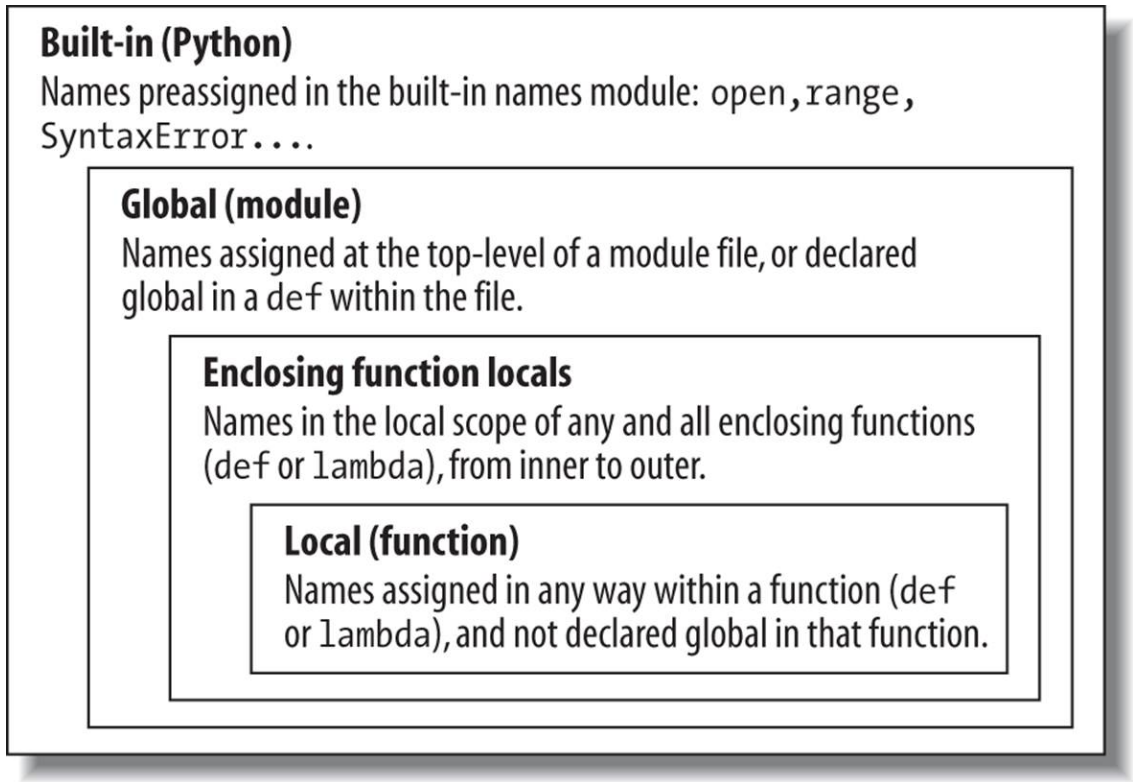
Variable arguments

- *pargs: tuple
- **kargs: dict

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)
...
>>> f(1,2,3,x=1,y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

Name Scope

LEGB scope lookup rule



Source: https://apprize.info/python/learning_1/18.html

Use `global` and `nonlocal` to change scopes

Scope

```
# Example 1
for i in range(10):
    # do something
print(i) # valid

# Example 2
def foo(x):
    print(y) # valid

y = 10
foo(y)
```

* Be careful of the global variable

Anonymous Functions

Lambda expression

```
lambda arg1, arg2, ..., argN: expression using arguments
```

```
sq = lambda x : x ** 2
sq(10)

sorted([(3,1),(4,4),(1,3),(2,2)],key=lambda x: x[1])
```

* C++11 []{...}

Functional programming

Higher-order functions

- `map(int,input().split())`
- `filter((lambda x: x > 0), L)`
- `reduce((lambda x, y: x * y), L)`

* You can even bind a function to a variable and call it

File stream

- `input()`: return a string
- `open(file,"r",encoding="utf-8")`, `close(file)`
- `with open(file_name) as infile`
  - Also an iterator
- `f.read()`, `f.write()`

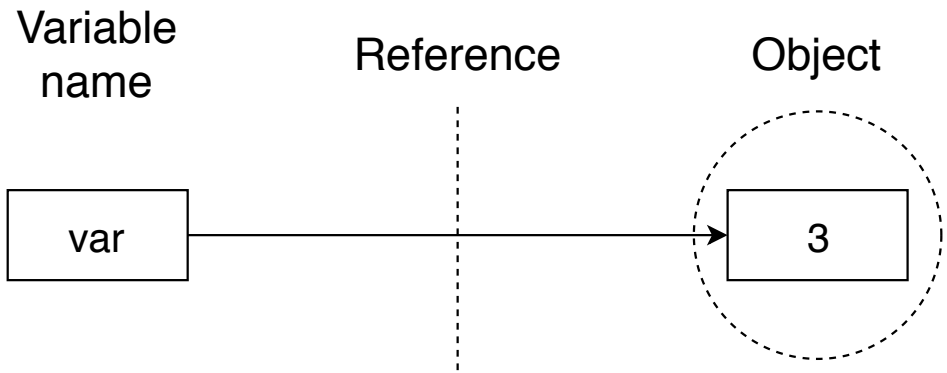
3

Dynamic Types

When creating a variable . . .

```
>>> var = 3
```

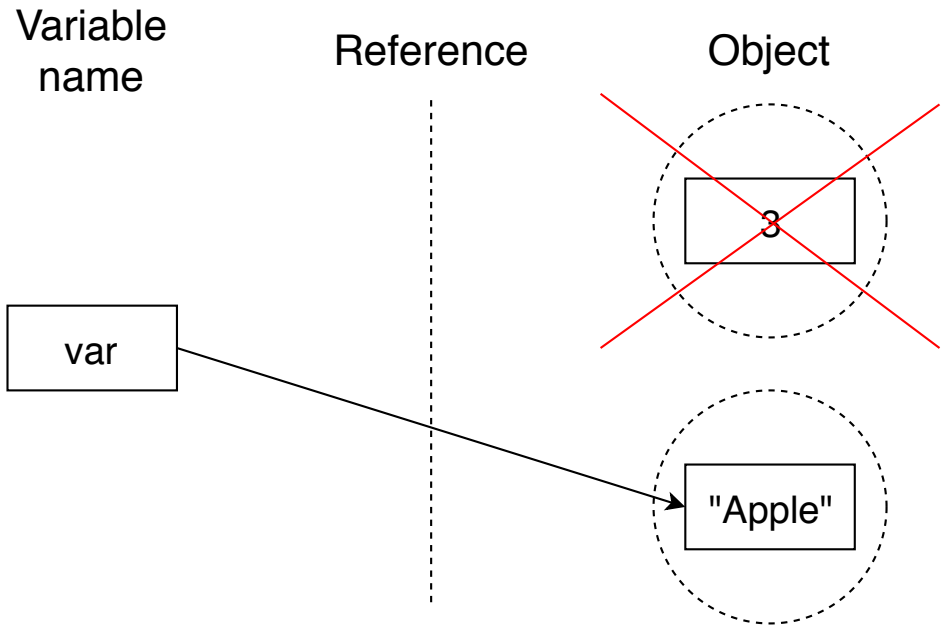
- Create a object to represent 3
- Create a variable `var` (if it has not been created)
- Create reference between variable & object 3



* Indeed, var gets a pointer pointing to the memory of the created object

Change reference

```
>>> var = 3
>>> var = "Apple"
```

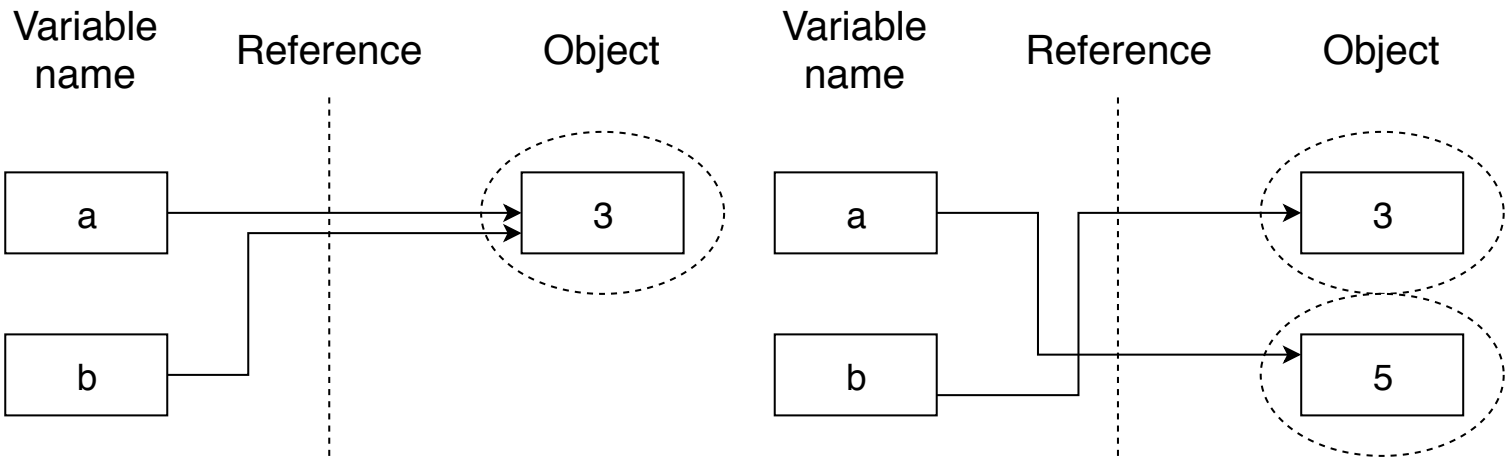


- Type belongs to objects instead of variables
- Garbage collection (gc), based on counter, very important in nowadays high-level languages

Shared reference

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

Int add is not inplace! (new object will be created)



* **Everything is an object!**

Shared reference

List operation is inplace!

```
>>> a = [1,2,3]
>>> b = a # a[:] # a.copy()
>>> a[0] = 5

>>> a
[5, 2, 3]
>>> b
[5, 2, 3]
```

Demo: Python tutor

Shared reference

Compare these two

```
>>> L = [1,2]
>>> L = L + [3] # concatenate: slower
>>> L
[1, 2, 3]
>>> L.append(4) # faster, but in-place
[1, 2, 3, 4]
```

Shared reference

Difference of == and is

```
>>> L = [1,2,3]
>>> M = L # M = [1,2,3]
>>> L == M # same value?
True
>>> L is M # same object?
True
```

Function arguments

- C: pass by value
- C++: pass by value, pass by reference

Function arguments

- C: pass by value
- C++: pass by value, pass by reference
- Python: pass by **object reference**

Function arguments

Pass by **object reference**

- Immutable objects (int,string) → create a new local object
- Mutable object (list,set) → operate on that object

```
# Immutable types
def foo(bar):
    bar = 'new value'
    print (bar)
    # >> 'new value'

answer_list = 'old value'
foo(answer_list)
print(answer_list)
# >> 'old value'
```

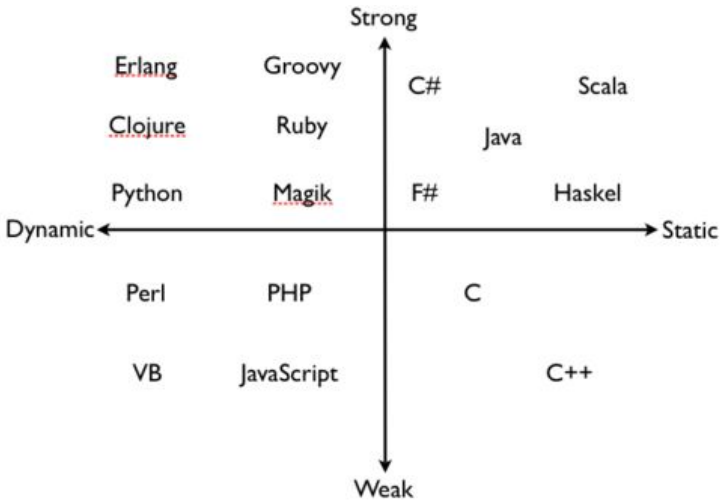
```
# Mutable types
def foo(bar):
    bar.append(42)
    print(bar)
    # >> [42]

answer_list = []
foo(answer_list)
print(answer_list)
# >> [42]
```

More about dynamic types - Reflection

Enable the program to modify/check the code of itself

- `type`
- `eval/exec`

Type systems

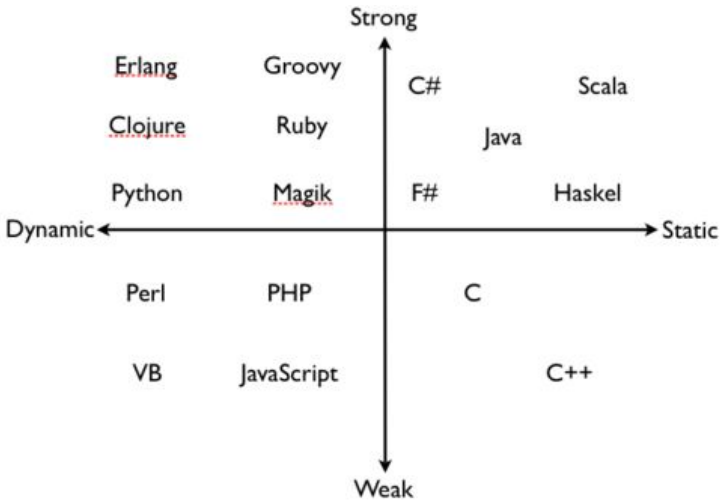


- Dynamic type: Objects have a type, but it is determined at runtime
- Static type: The type of variables must be known (and usually declared) at the point at which it is used (compile time)

```
int a;
a[10] = 1; // invalid
```

* Ref: https://www.zhihu.com/question/19918532

Type systems



- Strong type: You can't perform operations inappropriate to the type of the object
- Weak type: A compiler / interpreter will sometimes change the type of a variable

```
char b = 'a' + 1; // invalid
```

* Ref: https://www.zhihu.com/question/19918532

Dynamic Types

- Extremely flexible
- Biggest disadvantage: slow
  - Need to support various cases, e.g. +
  - Put many issues (like type checking) which can be done in compile time to runtime
  - Can be seen from the assembly source code, ref

Run codes in non-interactive mode

`python code.py`

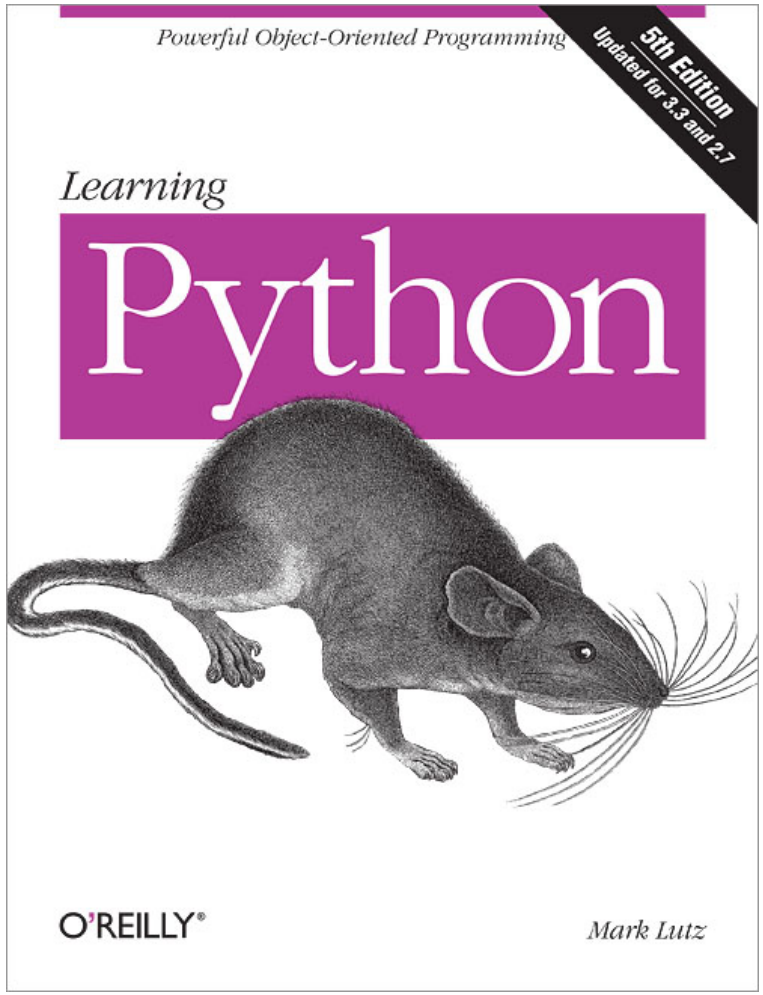
- No errors will be encountered unless you run the program
- Be careful of
  - Variable scope $\rightarrow$ avoid using the same name
  - Immutable / Mutable objects

4

Resources

Books

Learning Python, 5th Edition
Powerful Object-Oriented Programming
By Mark Lutz



* More materials can be found in `Python-Materials.pdf` and net disk

* Extremely detailed!

Courses

Structure and Interpretation of Computer Programs (SICP)

- UCB 61A
- NJU 22000130, Xingyu Feng

Documents

- Official documents: The Python Tutorial
- `help()`

PEP

Python Enhancement Proposals (PEP)

- PEP 8: *Style Guide for Python code*
- PEP 7: *Style Guide for C code*
- PEP 20: *The Zen of Python*, `import this`

Key idea of PEP 8

Be **concise, simple, and readable**

- One line code can do the work, then do not write 100 lines.
- Use list comprehension and functional facilities

5

Summary

Summary

- Introduction
- Basic types: Number, String, List, Tuple, Dictionary, Set
- Basic operations: List comprehension, pattern matching
- Control flow
- Functions: lambda expression, functional programming
- IO
- Dynamic types

* STL, Modules, OOP, etc. will be covered in the future seminar