

# Python

프로그래밍 언어마다 특징도 다양하고 장단점도 다양하다.

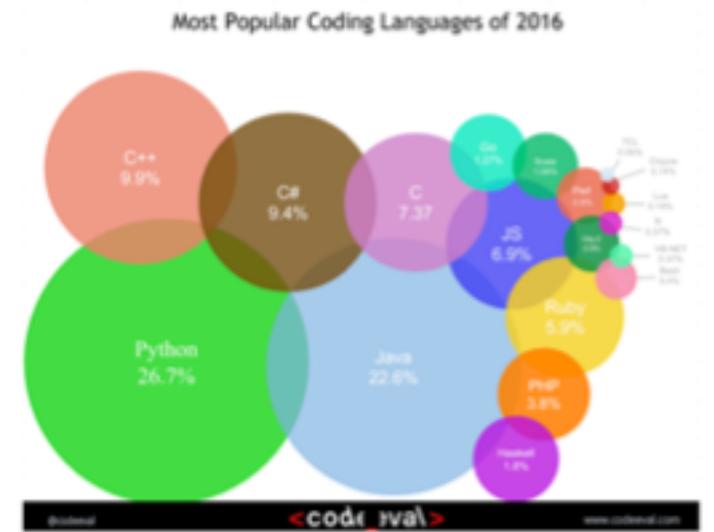


<http://ppss.kr/archives/28991>

<http://kstatic.inven.co.kr/upload/2017/08/03/bbs/i14621578868.jpg>

<http://redmist.tistory.com/21>

<http://redmist.tistory.com/30>



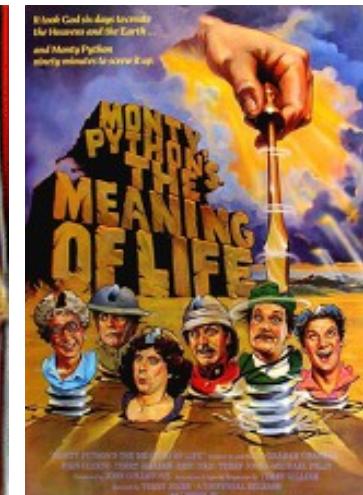
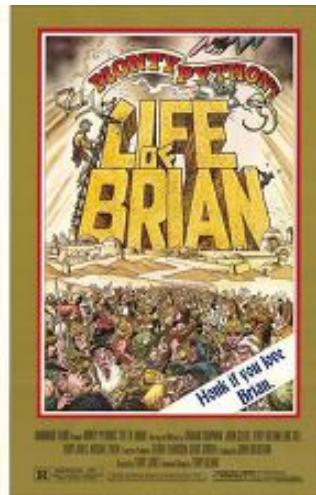
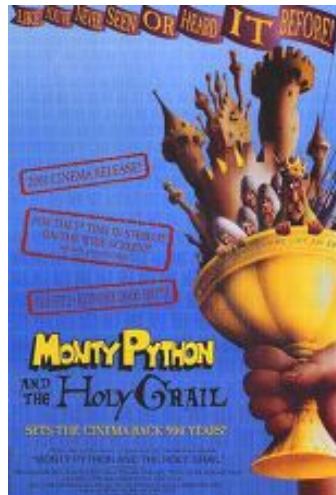
Mar 2017	Mar 2016	Change	Programming Language	Rating	Change
1	1		Java	10.38%	-4.14%
2	2		C	7.74%	-4.88%
3	3		C++	5.38%	-1.34%
4	4		CH	4.68%	-0.18%
5	5		Python	3.81%	-0.34%
6	7	+2	Visual Basic .NET	3.17%	-0.87%
7	6	-1	PHP	3.08%	-0.24%
8	8		JavaScript	2.88%	-0.23%
9	11	+3	Delphi/Object Pascal	2.54%	-0.04%
10	14	+4	Swift	2.28%	-0.68%
11	9	-2	Perl	2.28%	-0.07%
12	10	-2	Ruby	2.26%	-0.02%
13	12	-2	Assembly language	2.25%	-0.39%
14	16	+2	VB	2.18%	-0.73%
15	13	-2	Visual Basic	2.08%	-0.33%
16	15	-2	Objective-C	1.98%	-0.34%
17	48	+31	Go	1.98%	+1.78%
18	18		MATLAB	1.88%	-0.08%
19	19		PL/SQL	1.88%	-0.08%
20	26	+6	Scala	1.47%	-0.71%

TIOBE Index





# Humorous



<https://www.youtube.com/watch?v=iiu0IYQIPoE>

<https://gvanrossum.github.io/>

***Life is too short, You need***

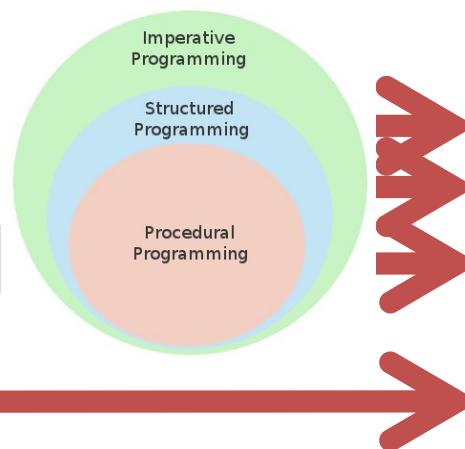
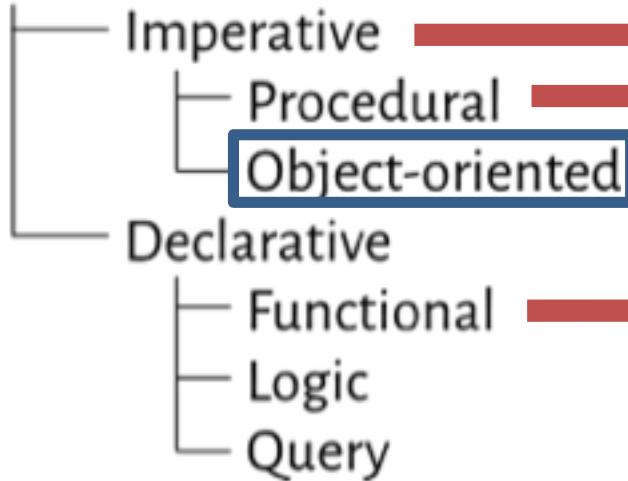


# 생산성

**Effective  
Efficient**

# Multi Paradigm

Programming languages



**Everything is an object**

The same problem can be solved and expressed in different ways

# Glue Language



## ■ CPython (c.f : cython)

- De facto
- Python Software Foundation 관리

## ■ 대체, 플랫폼 특정 구현체

- PyPy, Stackless
- IronPython, PythonNet, Jython
- Skulpt, Brython
- MicroPython

<https://wiki.python.org/moin/AdvocacyWritingTasks/GlueLanguage>  
<https://www.python.org/doc/essays/oma-darpa-mcc-position/>

# Library & Tool

## ■ 다양한 종류의 수많은 standard library 기본 탑재

플랫폼에 상관없음

## ■ 다양한 종류의 수많은 open-source libraries

the Python Package Index: <https://pypi.python.org>

pip

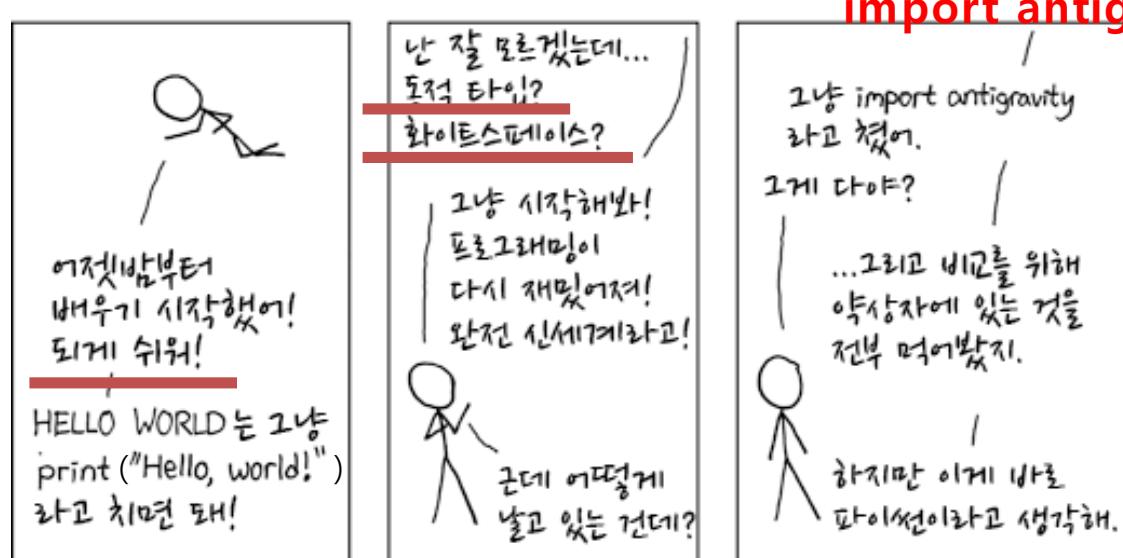
the de facto

default package manager

# General Purpose



C.f)  
Domain-specific



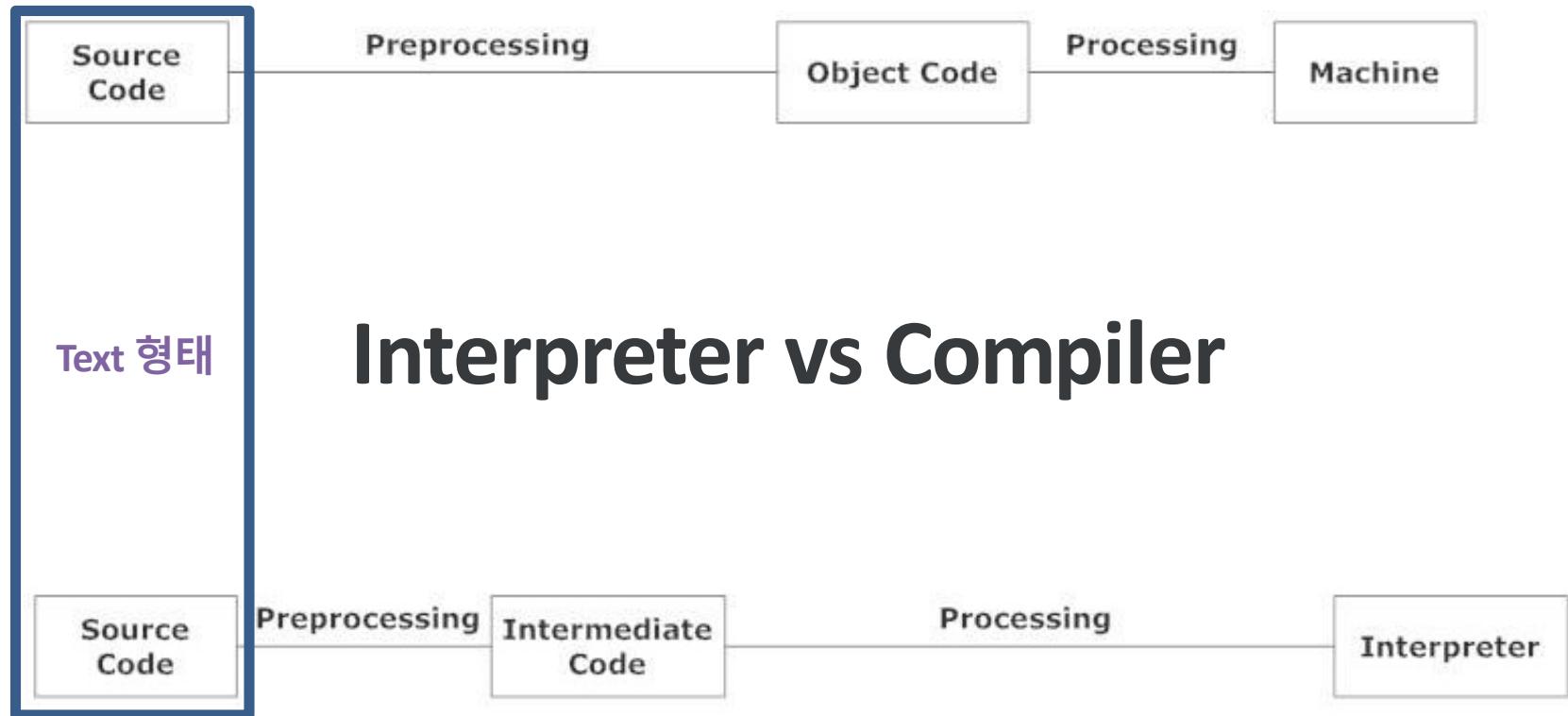
[https://en.wikipedia.org/wiki/List\\_of\\_Python\\_software](https://en.wikipedia.org/wiki/List_of_Python_software)  
<https://github.com/vinta/awesome-python>

c.f) 모바일 지원에 대한 약점

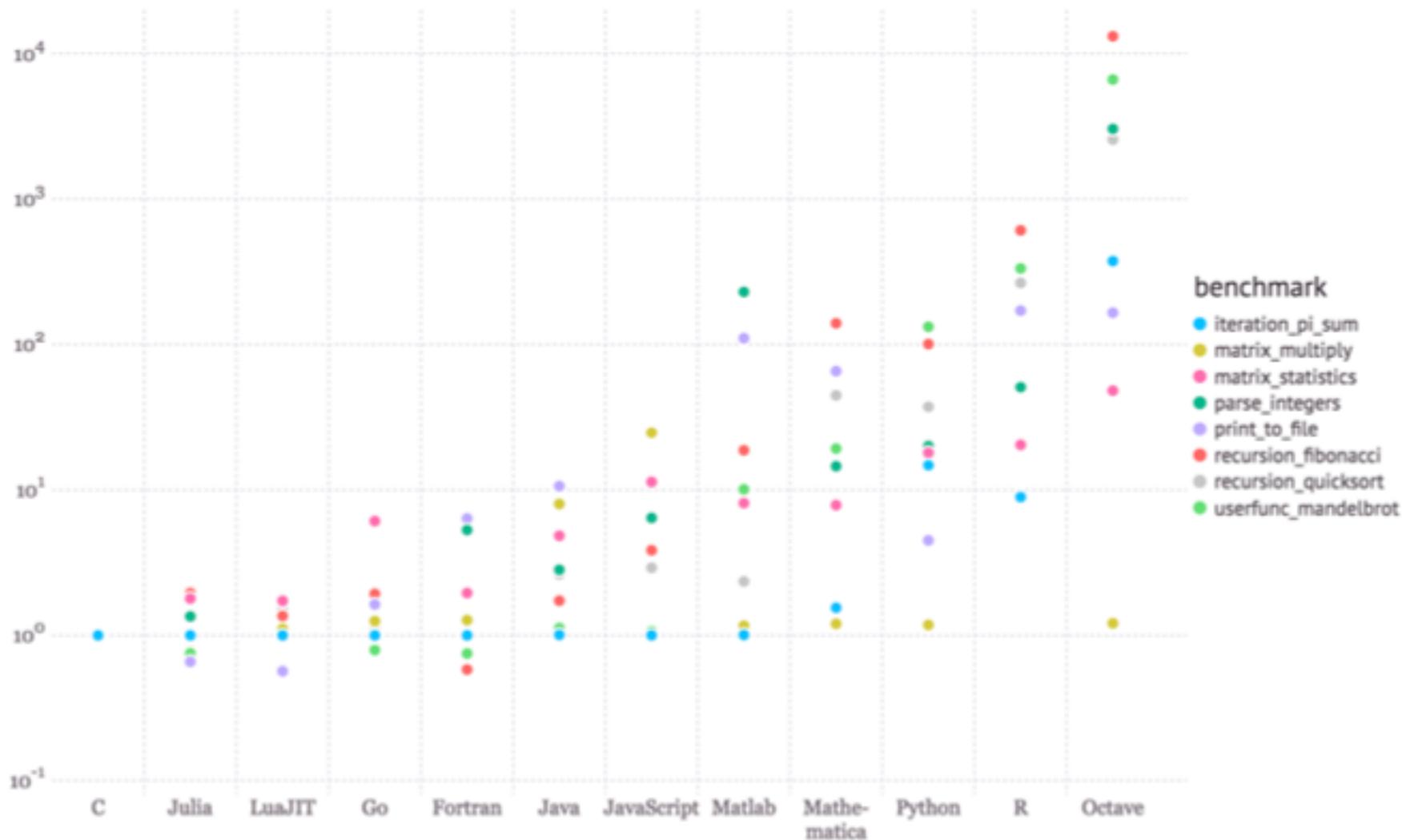
# Dynamic Language

<https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language>

# Interpreted Language script



파이 없는 dynamic 인터프리터 언어



# Python 개발 환경



2020년까지 bugfix만, 지원 종료 예정

<https://www.python.org/dev/peps/pep-0373/>

# REPL vs IDE vs Text Editor

don't need  
to use  
the print() function

Interactive Mode

## REPL



vs vs

## IDE



## Text Editor



vi, emacs...

## ■ Online

- Python.org Online Console: [www.python.org/shell](http://www.python.org/shell)
- Python Fiddle: [pythonfiddle.com](http://pythonfiddle.com)
- Repl.it: [repl.it](http://repl.it)
- Trinket: [trinket.io](http://trinket.io)
- Python Anywhere: [www.pythonanywhere.com](http://www.pythonanywhere.com)
- codeskulptor : <http://www.codeskulptor.org/viz/index.html>, <http://py3.codeskulptor.org/>
- <http://www.pythontutor.com/>

## ■ iOS (iPhone / iPad)

- [Pythonista app](#)

## ■ Android (Phones & Tablets)

- [Pydroid 3](#)

## ■ Distribution = a software bundle

- Python interpreter (Python standard library) + package managers



Anaconda



ActivePython



Enthought Canopy



python(x,y)

## ■ 가상

- virtualenv / venv
- VirtualenvWrapper
- pipenv

## ■ Docker

- <https://djangostars.com/blog/what-is-docker-and-how-to-use-it-with-python/>
- <https://www.pycon.kr/2015/program/71>

## ■ 윈도우 라이브러리

- <https://www.lfd.uci.edu/~gohlke/pythonlibs/>

### ■ REPL 실행

- \$ python3

### ■ Execute the file.py file:

- \$ python3 file.py

### ■ Execute the file.py file, and drop into REPL with namespace of file.py:

- \$ python3 -i file.py

### ■ Execute the json/tool.py module:

- \$ python3 -m json.tool

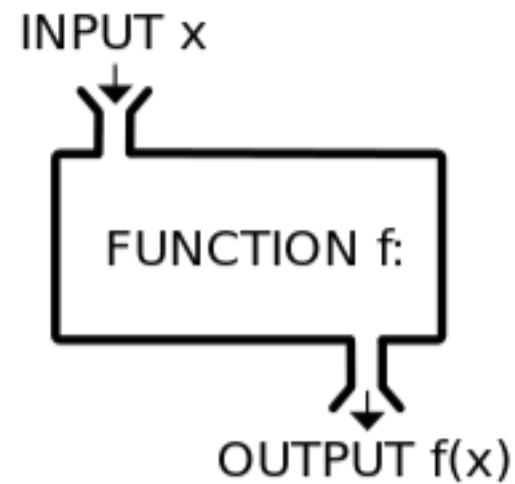
### ■ Execute "print('hi')"

- \$ python3 -c "print('hi')"

# **프로그래밍 문법의 구성 요소**

# 프로그래밍

문법(키워드, 식/문)을 이용해서 값(타입, 리터널)을 입력받고,  
계산/변환하고, 출력하는 흐름을 만드는 일



## The Zen of Python (PEP 20)

# import this

### 프로그래밍 언어의 문법

- 생각을 표현해내는 도구인 동시에, 생각이 구체화되는 틀
- 언어가 지향하고자 하는 철학에 따라 고안

문법에 대한 올바른 이해는 프로그래밍을 위한 필수적인 과정

### **BDFL**(Benevolent Dictator For Life!)

[https://en.wikipedia.org/wiki/Benevolent\\_dictator\\_for\\_life](https://en.wikipedia.org/wiki/Benevolent_dictator_for_life)

False	elif	lambda
None	else	nonlocal
True	except	not
and	finally	or
as	for	pass
assert	from	raise
break	global	return
class	if	try
continue	import	while
def	in	with
del	is	yield

import keyword

identifier(식별자)로 사용하지 못함

# Expression vs Statement

## 표현식 vs 구문

# Declaration vs Assignment

선언 vs 할당(대입)

## ■ 표현식(expression) = 평가식

- 표현식은 간단한 수식으로 취급되고 있으나, 언어의 구조의 근간을 이루는 매우 중요한 개념
- 어떤 값 혹은 값들과 연산자를 함께 사용해서 수식을 표현한 것
- 이후 “평가”되면서 하나의 특정한 결과값으로 축약
- 수
  - $1 + 1$  과 같은 수식
    - $1 + 1$  이라는 표현식은 평가될 때 계산되어 2라는 값으로 축약되는 식
  - 0과 같이 값 리터럴로 값을 표현해놓은 것
- 문자열
  - 그 자체가 값으로 표현식이며,
  - 문자열과 관련되는 연산
    - "hello" + ", world"
- 함수
  - lambda
- 표현식은 궁극적으로 “평가”되며, 평가된다는 것은 표현식은 결국 하나의 값으로 수렴한다는 의미

## ■ 구문(statement)

- 예약어(reserved word)와 표현식을 결합한 패턴
- 컴퓨터가 수행해야 하는 하나의 단일 작업(instruction)을 명시.
  - 활당, 대입 (assigning statement) : python에서는 보통 '바인딩(binding)'이라는 표현을 씀, 어떤 값에 이름을 붙이는 작업.
  - 선언, 정의(Declaration) : 재사용이 가능한 독립적인 단위를 정의. 별도의 선언 문법과 그 내용을 기술하는 블럭 혹은 블럭들로 구성. Ex) python에서는 기함수나 클래스를 정의
    - 블럭
      - » 여러 구문이 순서대로 나열된 덩어리
      - » 블럭은 여러 줄의 구문으로 구성되며, 블럭 내에서 구문은 위에서 아래로 쓰여진 순서대로 실행. 블럭은 분기문에서 조건에 따라 수행되어야 할 작업이나 반복문에서 반복적으로 수행해야 하는 일련의 작업을 나타낼 때 사용하며, 클래스나 함수를 정의할 때에도 쓰임.
  - 분기문 : 조건에 따라 수행할 작업을 나눌 때 사용. Ex) if 문
  - 반복문 : 특정한 작업을 반복수행할때 사용. Ex) for 문 및 while 문

# Type

## ■ 숫자형

- 일반적으로 정수나 float 값은 실수라는 수 체계에서 볼 때는 같은 값이지만 컴퓨터에서는 이진수를 사용해서 이를 표현하고 관리하기 위해서 서로 다른 기술적인 체계를 따름.
- 흔히 숫자라고하는 타입은 정수와 실수로 나뉘며, 보통의 프로그래밍 언어에서는 이들을 구분
- 그외에 프로그래밍 언어에서 사용되는 가장 기초적인 데이터 타입에는 다음과 같은 것들이 있음.
  - 정수 : 0, 1, -1 과 같이 소수점 이하 자리가 없는 수. 수학에서의 정수 개념과 동일. (int )
  - 실수 : 0.1, 0.5 와 같이 소수점 아래로 숫자가 있는 수. (float)
- 더하기, 빼기등의 산술 연산을 적용하는 값이며 가장 기본적인 수 개념들을 표현

## ■ 문자, 문자열

- 숫자 "1", "a", "A" 와 같이 하나의 낱자를 문자라 하며, 이러한 문자들이 1개 이상있는 단어/문장와 같은 텍스트
- 파이썬에서는 str 타입으로 분류한다. 특히 파이썬은 낱자와 문자열 사이에 구분이 없이 모두 str 타입을 적용한다.

## ■ 불리언

- 참/거짓을 뜻하는 대수값. 보통 컴퓨터는 0을 거짓, 0이 아닌 것을 참으로 구분
- True 와 False 의 두 멤버만 존재 (bool)
- Python에서는 숫자형의 일부

## ■ Container

- 기본적인 데이터 타입을 조합하여, 여러 개의 값을 하나의 단위로 묶어서 다루는 데이터 타입
- 논리적으로 이들은 데이터 타입인 동시에 데이터의 구조(흔히 말하는 자료 구조)의 한 종류. 보통 다른 데이터들을 원소로 하는 집합처럼 생각되는 타입들
- Sequence
  - 리스트 : 순서가 있는 원소들의 묶음
  - 튜플 : 순서가 있는 원소들의 묶음. 리스트와 혼동하기 쉬운데 단순히 하나 이상의 값을 묶어서 하나로 취급하는 용도로 사용된다.
  - range
- Lookup
  - 사전 : 그룹내의 고유한 이름인 키와 그 키에 대응하는 값으로 이루어지는 키값 쌍(key-value pair)들의 집합이다.
  - 집합/셋(set) : 순서가 없는 고유한 원소들의 집합.

## ■ None

- 존재하지 않음을 표현하기 위해서 “아무것도 아닌 것”을 나타내는 값
- 어떤 값이 없는 상태를 가리킬만한 표현이 마땅히 없기 때문에 “아무것도 없다”는 것으로 약속해놓은 어떤 값을 하나 만들어 놓은 것
- None이라고 대문자로 시작하도록 쓰며, 실제 출력해보아도 아무것도 출력되지 않음.
- 값이 없지만 False나 0과는 다르기 때문에 어떤 값으로 초기화하기 어려운 경우에 쓰기도 함

## ■ Dynamically typed

- All values and expressions have a type, but there is no declared type system
- Everything, including functions and built-in primitive values, is an object
- Python's built-in types are all classes, with operators and methods
  - E.g., integers, strings, lists and dictionaries
- Use `type(value)` to query a value's type and `dir(value)` to query its attributes
- Most built-in types have a literal form
  - E.g., 97 is an int, 'swallow' is a str, (4, 2) is a tuple pair, [] is an empty list, {'parrot'} is a set with a single value, {} is an empty dict

# Literal vs Object-oriented

값, 기호로 고정된 값을 대표  
간단하게 값 생성

## ■ 수를 표현하는 기본 리터럴

- 정수의 경우, 그냥 숫자들로 숫자값. 0, 100, 123 과 같은 표현
- 실수의 경우, 중간에 소수점이 들어간다. 0.1, 4.2, 3.13123 와 같은 식
- 0. 으로 시작하는 실수값에서는 흔히 앞에 시작하는 0을 뺄 수 있다. .5 는 0.5를 줄여쓴 표현
- 부호를 나타내는 - , +를 앞에 붙일 수 있다. (-1, +2.3 등)

## ■ 기본적으로 그냥 숫자만 사용하는 경우, 이는 10진법 값으로 해석.

- 10진법외에도 이진법, 8진법, 16진법이 존재한다.
  - 이진법 숫자는 0b로 시작
  - 8진법 숫자는 0o로 시작
  - 16진법숫자는 0x로 시작한(대소문자를 구분하지 않음)

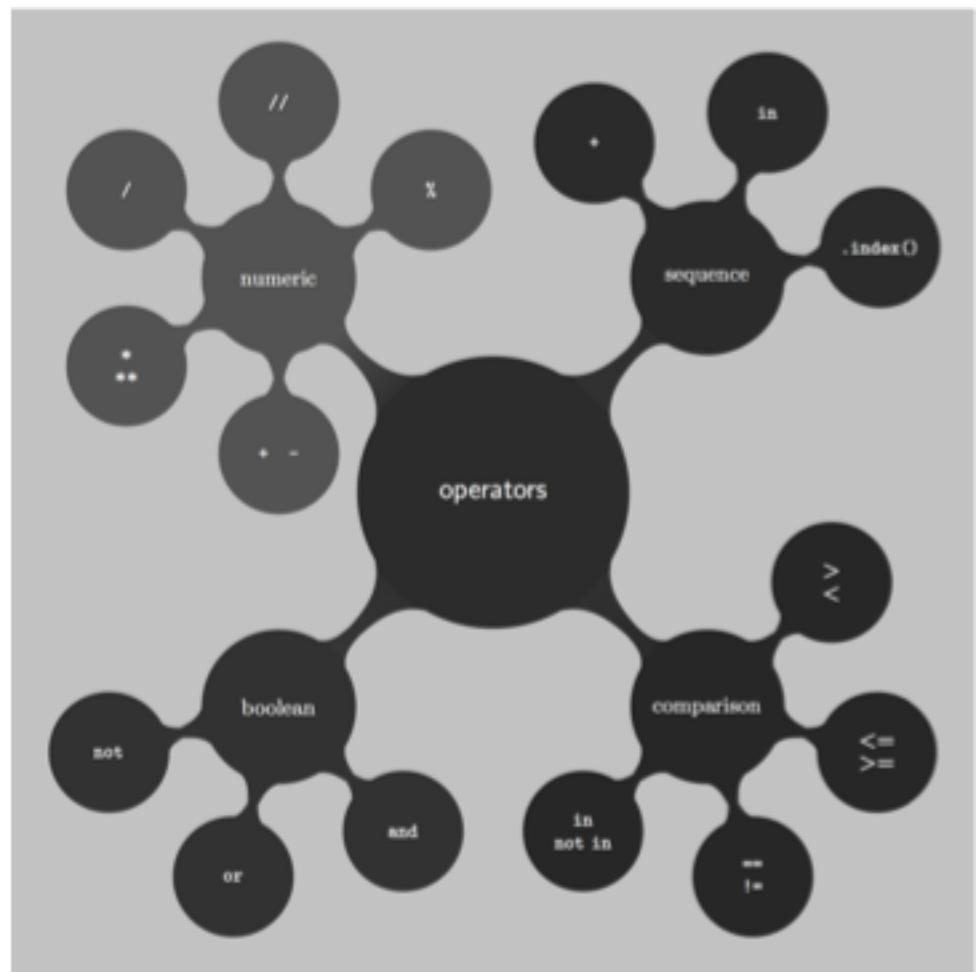
## ■ 숫자 리터럴 중간에 \_ 를 쓰는 것은 무시.

- 원하는 아무자리에나 가능

## ■ 참/ 거짓을 의미하는 부울대수값.

- 이들은 그 자체가 키워드로 True/False를 사용하여 표현

# Operation



## ■ 산술연산

- 계산기

## ■ 비교연산

- 동등 및 대소를 비교. 참고로 '대소'비교는 '전후'비교가 사실은 정확한 표현
- 비교 연산은 숫자값 뿐만 아니라 문자열에 대해서도 적용할 수 있음.

## ■ 비트연산

## ■ 멤버십 연산

- 특정한 집합에 어떤 멤버가 속해있는지를 판단하는 것으로 비교연산에 기반을 둠
- is, is not : 값의 크기가 아닌 값 자체의 정체성(identity)이 완전히 동일한지를 검사
- in, not in : 멤버십 연산. 어떠한 집합 내에 원소가 포함되는지를 검사 ('a' in 'apple')

## ■ 논리연산

- 비교 연산의 결과는 보통 참/거짓. 이러한 불리언값은 다음의 연산을 적용. 참고로 불리언외의 타입의 값도 논리연산을 적용

Operator	Example	Meaning	Result
+ (unary)	+ a	Unary Positive	a In other words, it doesn't really do anything. It mostly exists for the sake of completeness, to complement Unary Negation.
+ (binary)	a + b	Addition	Sum of a and b
- (unary)	-a	Unary Negation	Value equal to a but opposite in sign
- (binary)	a - b	Subtraction	b subtracted from a
*	a * b	Multiplication	Product of a and b
/	a / b	Division	Quotient when a is divided by b. The result always has type float.
%	a % b	Modulus	Remainder when a is divided by b
//	a // b	Floor Division (Integer Division)	Quotient when a is divided by b, rounded to the next smallest whole number
**	a ** b	Exponentiation	a raised to the power of b

Operator	Example	Meaning	Result
<code>==</code>	<code>a == b</code>	Equal to	True if the value of a is equal to the value of b False otherwise
<code>!=</code> <code>&lt; &gt;</code>	<code>a != b</code>	Not equal to	True if a is not equal to b False otherwise
<code>&lt;</code>	<code>a &lt; b</code>	Less than	True if a is less than b False otherwise
<code>&lt;=</code>	<code>a &lt;= b</code>	Less than or equal to	True if a is less than or equal to b False otherwise
<code>&gt;</code>	<code>a &gt; b</code>	Greater than	True if a is greater than b False otherwise
<code>&gt;=</code>	<code>a &gt;= b</code>	Greater than or equal to	True if a is greater than or equal to b False otherwise

Truthy	Falsey
True	False
Most objects	None
1	0
3.2	0.0
[1, 2]	[] (empty list)
{'a': 1, 'b': 2}	{ } (empty dict)
'string'	"" (empty string)
'False'	
'0'	

Operator	Example	Meaning
not	not x	True if x is False False if x is True (Logically reverses the sense of x)
or	x or y	True if either x or y is True False otherwise
and	x and y	True if both x and y are True False otherwise

	Operator	Description
lowest precedence	or	Boolean OR
	and	Boolean AND
	not	Boolean NOT
	in, not in	membership
	==, !=, <, <=, >, >=, is, is not	comparisons, identity
		bitwise OR
	^	bitwise XOR
	&	bitwise AND
	<<, >>	bit shifts
	+, -	addition, subtraction
highest precedence	*, /, //, %	multiplication, division, floor division, modulo
	+x, -x, ~x	unary positive, unary negation, bitwise negation
	**	exponentiation

## ■ 일반적 의미

- 아직 알려지지 않거나 어느 정도까지만 알려져 있는 양이나 정보에 대한 상징적인 **이름**
- Cf) 대수학 : 수식에 따라서 변하는 값
- Cf) 상수 : 변하지 않는 값

## ■ 프로그래밍에서의 변수

- 변수를 이용하면 값을 기억해 두고 필요할 때 읽을 수 있음.
- 중간 계산값을 저장하거나 누적할 수 있음.

## ■ Python

- 값을 저장하는 메모리 상의 공간을 가르키는 **이름**
- Python은 모든 것이 객체이므로 변수보다는 **식별자**로 언급하는게 맞지만 변수로 보통 사용
  - **변수, 상수, 함수, 사용자 정의 타입** 등에서 다른 것들과 구분하기 위해서 사용되는 변수의 이름, **상수의 이름, 함수의 이름, 사용자 정의 타입의 이름** 등 '이름'을 일반화 해서 지칭하는 용어
- **Pvthon에는 이름있는 상수 개념 없음**
- **선언 및 할당이 동시에 이루어져야 함**

# Python 문법

## ■ Python 문법

- 규칙(키워드 등)의 수가 적음

➤ Python 3 : 33 keywords (True, False)  
 ➤ Python 2 : 31  
 ➤ C++ : 62  
 ➤ Java : 53  
 ➤ Visual Basic : >120

- 대부분의 규칙이 일관된 맵락을 가지고 있음

이해하고 배우기 쉬움

**Case sensitive 대소문자 구분**

**line oriented (줄 기반)**

**Space sensitive / Indentation 중요**

- Indentation used to define **block structure**

- Implicit continuation across unclosed bracketing ((...), [...], {...})

- Forced continuation after `\` at end of line

### ■ #

- Line-based, i.e., but independent of indentation (but advisable to do so)

### ■ #!

- Shell-script friendly, i.e., **#!** as first line

- It is possible to combine multiple statements on a single line
- The semi-colon is a statement separator
- This is, however, strongly **discouraged**

```
if eggs: print('Egg'); print('Spam')
```

Possible

```
if eggs:  
    print('Egg')  
    print('Spam')
```

Preferred

- Python has no **named constants**.
- Python treats every name as a variable. <- identifier
- A variable holds an **object reference**
- A variable is a name for an object within a scope
- None is a reference to nothing
- A variable is created in the current scope on first assignment to its name
- It's a runtime error if an unbound variable is referenced

## ■ Variables are introduced in the smallest enclosing scope

- Parameter names are local to their corresponding construct (i.e., module, function, lambda or comprehension)
- To assign to a global from within a function, declare it `global` in the function
- Control-flow constructs, such as `for`, do not define scopes
- `del` removes a name from a scope

binding

identifier → **a** = **23** ← value  
= name



Assignment operator

```
parrot = 'Dead'
```

Simple assignment

```
x = y = z = 0
```

Assignment of single value to multiple targets

```
lhs, rhs = lhs, rhs
```

Assignment of multiple values to multiple targets

```
counter += 1
```

Augmented assignment

```
world = 'Hello'
```

Global assignment from within a function

```
def farewell():
```

```
    global world
```

```
    world = 'Goodbye'
```

- augmented assignments are statements not expressions
- Cannot be chained and can only have a single target
- no ++ or --

$+=$     $/=$     $<<=$

$-=$     $//=$     $>>=$

$*=$     $%=$     $\&=$

$**=$                $|=$

$^=$

---

### Arithmetic

---

$+$

$-$

$*$

$/$

$\%$

$//$

$**$

---

### Bitwise

---

$\&$

$|$   
 $\wedge$

$>>$

$<<$

- **case-sensitive names**
  - E.g., functions, classes and variables
- **Some identifier classes have reserved meanings**

Class	Example	Meaning
<code>_*</code>	<code>_load_config</code>	Not imported by wildcard module imports — a convention to indicate privacy in general
<code>__ __</code>	<code>__init__</code>	System-defined names, such as special method names and system variables
<code>__*</code>	<code>__cached_value</code>	Class-private names, so typically used to name private attributes

## ■ Camel Case

- Second and subsequent words are capitalized, to make word boundaries easier to see. (Presumably, it struck someone at some point that the capital letters strewn throughout the variable name vaguely resemble camel humps.)
  - Example: `numberOfCollegeGraduates`

## ■ Pascal Case

- Identical to Camel Case, except the first word is also capitalized.
  - Example: `NumberOfCollegeGraduates`

## ■ Snake Case

- Words are separated by underscores.
  - Example: `number_of_college_graduates`

## ■ style guide

- Offers guidance on accepted convention, and when to follow and when to break with convention
- mostly on layout and naming

## ■ But there is also programming guidance

## ■ PEP 8 conformance can be checked automatically

- E.g., <http://pep8online.com/>, pep8, flake8

<https://www.python.org/dev/peps/pep-0008/#naming-conventions>

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

Use 4 spaces per indentation level.

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

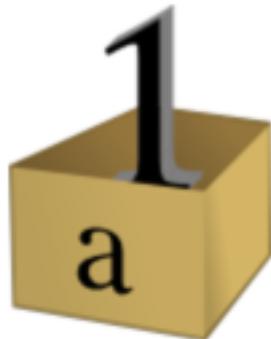
Limit all lines to a maximum of 79 characters.

Use is not operator rather than not ... is.

Use string methods instead of the string module.  
Class ..... use the CapWords convention.

# **Memory**

C



```
int a = 1;
```



```
a = 2;
```



```
int b = a;
```

---

## Python



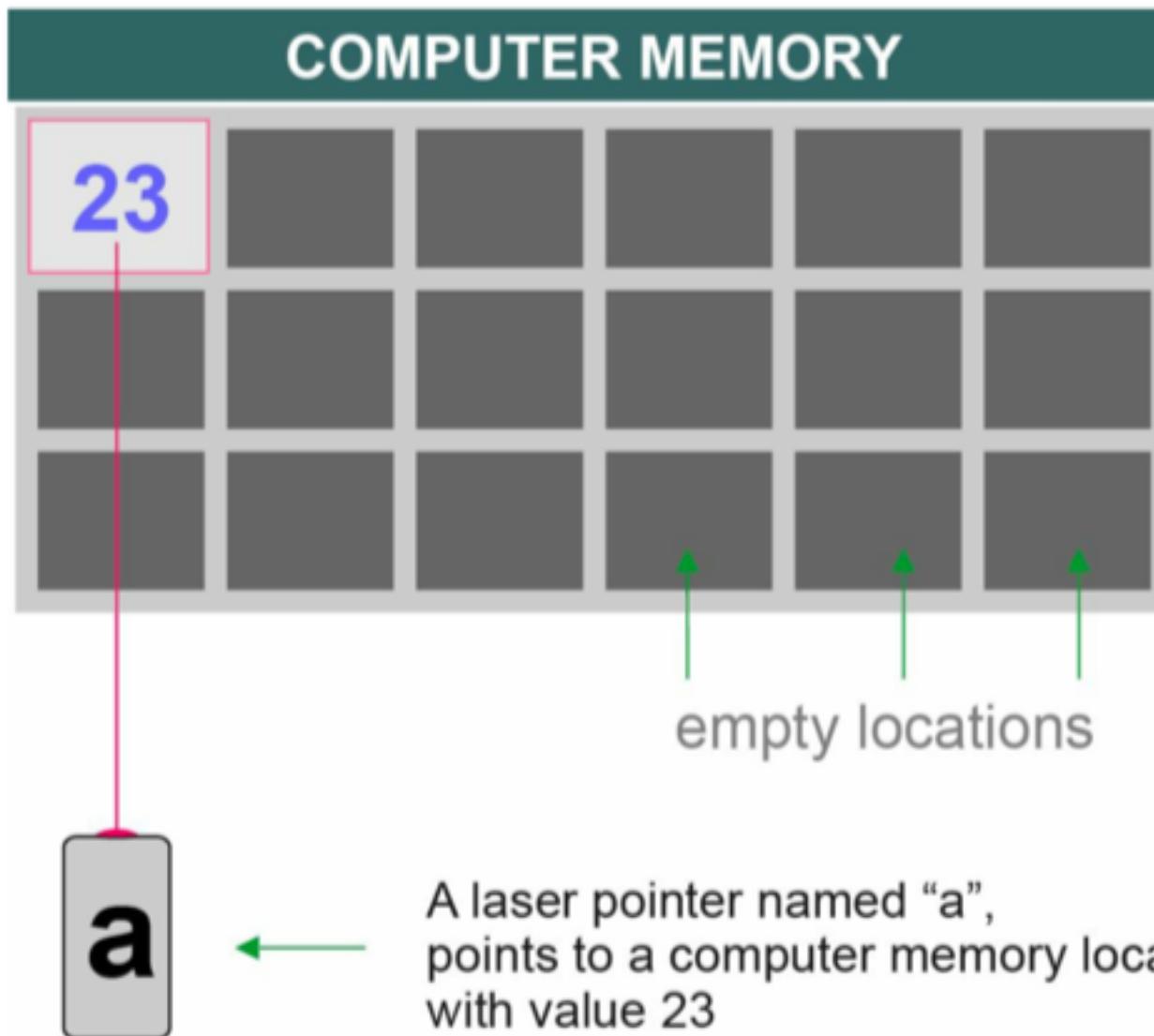
```
a = 1
```

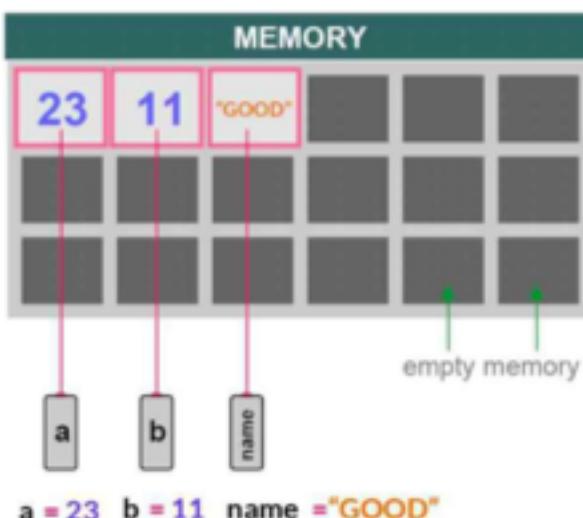


```
a = 2
```

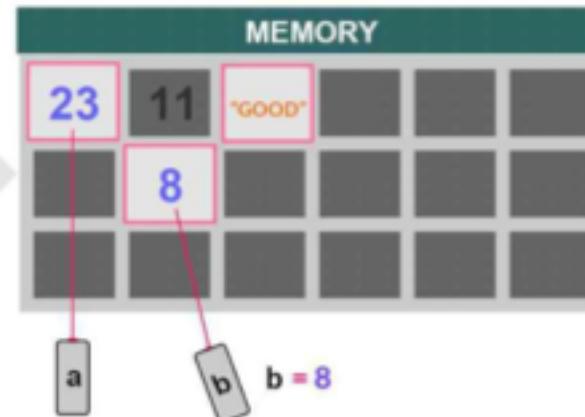


```
b = a
```

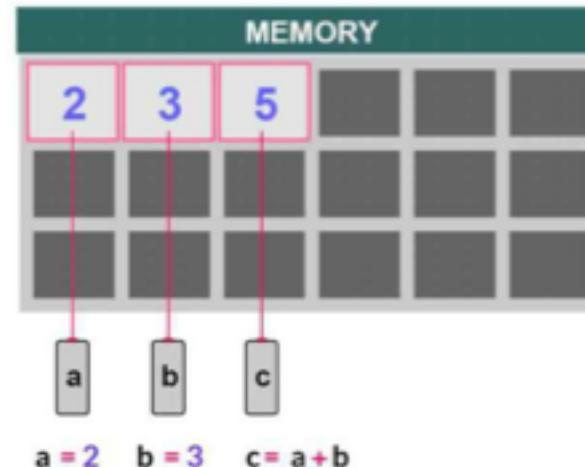
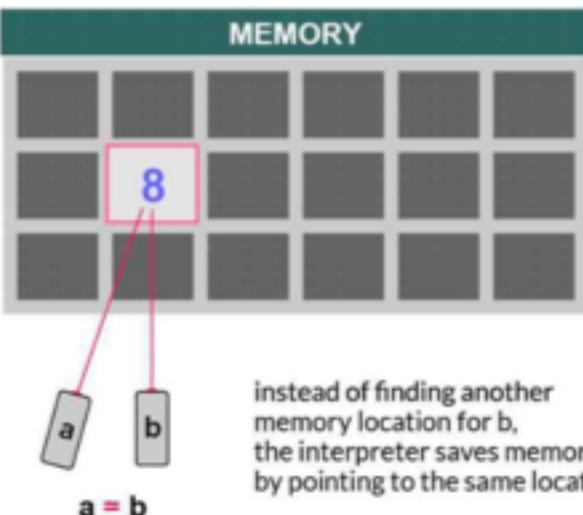




lets change the value of b from 11 to 8



Because numbers are immutable, "b" changes location to the new value.  
When there is no reference to a memory location the value fades away and the location is free to use again.  
This process is known as garbage collection



# Mutability

Cannot be modified in place

## Variable Creation

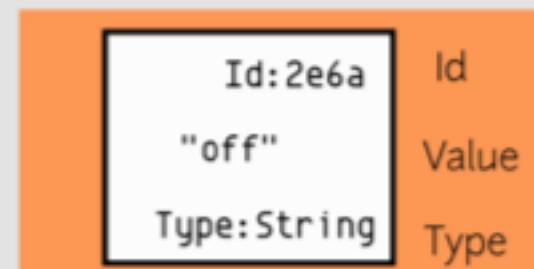
Code

`status = "off"`

What Computer Does

Variables

Objects



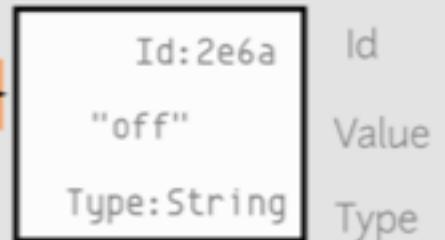
**Step 1: Python creates an object**

`status = "off"`

Variables

Objects

`status` →



**Step 2: A variable is created**

## Rebinding Variables

Code

a = 400

What Computer Does

Variables

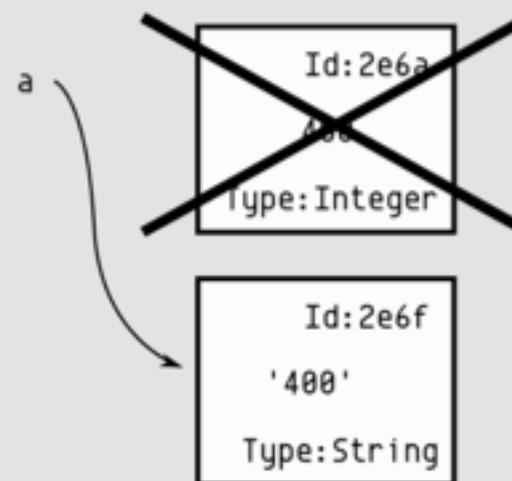
Objects



a = '400'

Variables

Objects

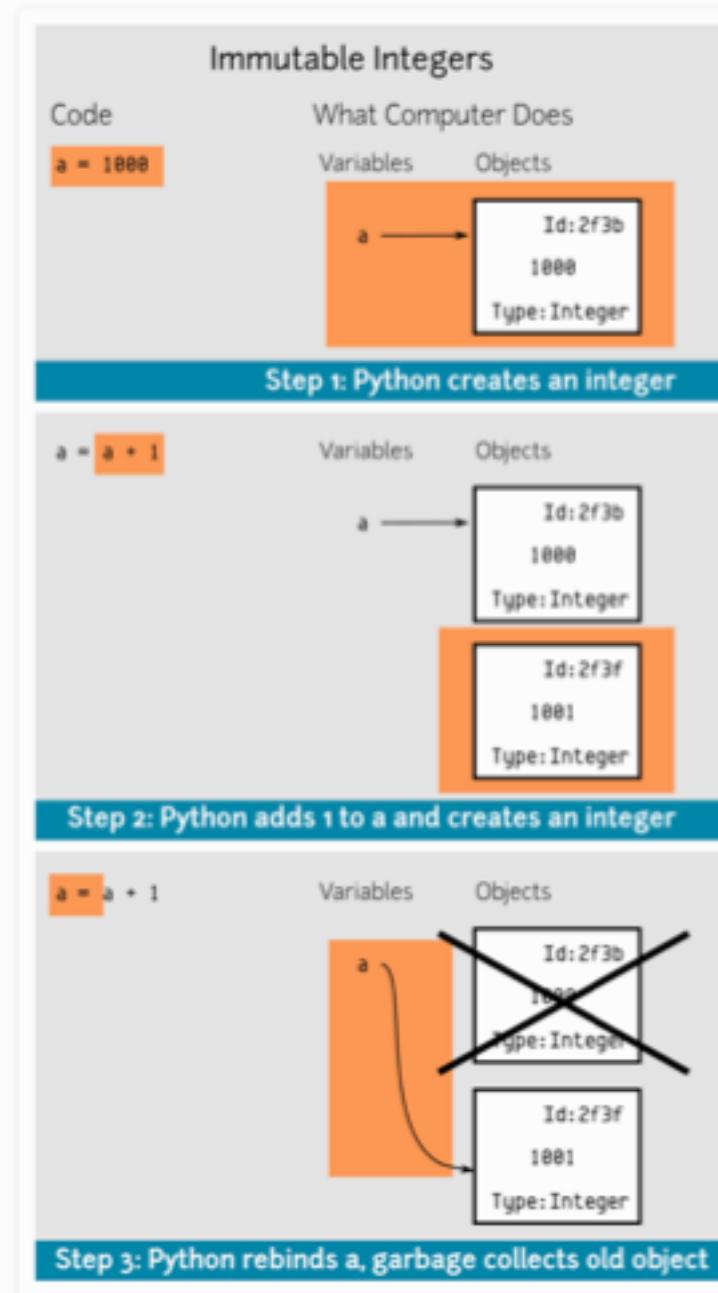


```
import sys  
>>> names = []  
>>> sys.getrefcount(names)
```

Old Object is Garbage Collected

# Built-in Types

# 1. Numeric



Type	예시
int	14, -14
int (Hex)	0xe
int (Octal)	0o16
int (Binary)	0b1110
float	14.0
float	1.4e1
complex	14+0j
bool	True, False
Underscore (readability)	1_000

- The **int** type offers arbitrary precision
  - Not limited to machine word size
  - import sys
  - sys.maxsize
- It supports the usual operators
  - But / results in a float,  
so use // if integer  
division is needed

- import sys
- sys.float\_info

```
>>> x = 1.1 + 2.2
>>> x == 3.3
False
```

3e8  
-273.15  
.5  
float('NaN')  
float('-Infinity')

■ Almost all platforms represent Python float values as 64-bit “double-precision” values, according to the IEEE 754 standard. In that case, the maximum value a floating-point number can have is approximately  $1.8 \times 10^{308}$ . Python will indicate a number greater than that by the string inf:

- 1.79e308
- 1.79e+308
- 1.8e308
- inf
- [https://en.wikipedia.org/wiki/IEEE\\_754\\_revision](https://en.wikipedia.org/wiki/IEEE_754_revision)

■ The closest a nonzero number can be to zero is approximately  $5.0 \times 10^{-324}$ . Anything closer to zero than that is effectively zero:

- 5e-324
- 5e-324
- 1e-325
- 0.0

- Floating point numbers are represented internally as binary (base-2) fractions. Most decimal fractions cannot be represented exactly as binary fractions, so in most cases the internal representation of a floating-point number is an approximation of the actual value. In practice, the difference between the actual value and the represented value is very small and should not usually cause significant problems.
- <https://docs.python.org/3.6/tutorial/floatingpoint.html>

## ■ Sequential Reduction : Left to Right

○ 1-2-3-4

→ -1-3-4

→ -4-4

→ -8

○ 1-(2-3)-4

→ 1--1-4

→ 0-4

→ -4

○ 9>1e-4 and {0:"Cavern", 5:'Tunnel'}[0].upper() in 'cave'

➤ True and {0:"Cavern", 5:'Tunnel'}[0].upper() in 'cave'

➤ True and "Cavern".upper() in 'cave'

➤ True and "CAVERN" in 'cave'

➤ True and False

➤ False

## ■ General Rules

- Left-to-right evaluation, but look-ahead first, checking if a higher priority operation comes next; parentheses force evaluation order; evaluation work can be "queued up" due to operator priority

## Binary arithmetic operators

- + Addition
- Subtraction
- \*
- \*\* Multiplication
- \*\* Power
- / Division
- // Integer division
- % Modulo

## Unary arithmetic operators

- + Unary plus
- Unary minus

## Bitwise operators

- ~ Bitwise complement
- << Bitwise left shift
- >> Bitwise right shift
- & Bitwise and
- | Bitwise or
- ^ Bitwise xor

Operation	Provided By	Result
abs(num)	<code>_abs_</code>	Absolute value of num
num + num2	<code>_add_</code>	Addition
bool(num)	<code>_bool_</code>	Boolean conversion
num == num2	<code>_eq_</code>	Equality
float(num)	<code>_float_</code>	Float conversion
num // num2	<code>_floordiv_</code>	Integer division
num >= num2	<code>_ge_</code>	Greater or equal
num > num2	<code>_gt_</code>	Greater than
int(num)	<code>_int_</code>	Integer conversion
num <= num2	<code>_le_</code>	Less or equal
num < num2	<code>_lt_</code>	Less than
num % num2	<code>_mod_</code>	Modulus
num * num2	<code>_mul_</code>	Multiplication
num != num2	<code>_ne_</code>	Not equal
-num	<code>_neg_</code>	Negative
+num	<code>_pos_</code>	Positive
num ** num2	<code>_pow_</code>	Power
round(num)	<code>_round_</code>	Round
num._sizeof_()	<code>_sizeof_</code>	Bytes for internal representation
str(num)	<code>_str_</code>	String conversion
num - num2	<code>_sub_</code>	Subtraction
num / num2	<code>_truediv_</code>	Float division
math.trunc(num)	<code>_trunc_</code>	Truncation

Operation	Provided By	Result
num & num2	<code>_and_</code>	Bitwise and
<code>math.ceil(num)</code>	<code>_ceil_</code>	Ceiling
<code>math.floor(num)</code>	<code>_floor_</code>	Floor
<code>~num</code>	<code>_invert_</code>	Bitwise inverse
<code>num &lt;&lt; num2</code>	<code>_lshift_</code>	Left shift
<code>num   num2</code>	<code>_or_</code>	Bitwise or
<code>num &gt;&gt; num2</code>	<code>_rshift_</code>	Right shift
<code>num ^ num2</code>	<code>_xor_</code>	Bitwise xor
<code>num.bit_length()</code>	<code>bit_length</code>	Number of bits necessary

Operation	Result
f.as_integer_ratio()	Returns num, denom tuple
f.is_integer()	Boolean if whole number

<b>abs(value)</b>	Absolute value
<b>bin(integer)</b>	String of number in binary
<b>divmod(dividend, divisor)</b>	Integer division and remainder
<b>float(string)</b>	Floating-point number from string
<b>hex(integer)</b>	String of number in hexadecimal
<b>int(string)</b>	Integer from decimal number
<b>int(string, base)</b>	Integer from number in base
<b>oct(integer)</b>	String of number in octal
<b>pow(value, exponent)</b>	value $\star\star$ exponent
<b>round(value)</b>	Round to integer
<b>round(value, places)</b>	Round to places decimal places
<b>str(value)</b>	String form of number (decimal)
<b>sum(values)</b>	Sum sequence (e.g., list) of values
<b>sum(values, initial)</b>	Sum sequence from initial start

# Sequence

## ■ by holding items

- Container sequences

- list, tuple, and collections.deque can **hold items of different types**.
  - hold references to the objects they contain, which may be of any type

- Flat sequences

- str, bytes, bytearray, memoryview, and array.array **hold items of one type**.
  - physically store the value of each item within its own memory space, and not as distinct objects.
  - more compact, but they are limited to holding primitive values like characters, bytes, and numbers.

## ■ by mutability:

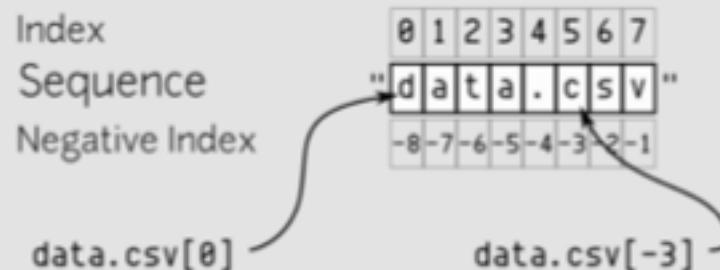
- Mutable sequences

- list, bytearray, array.array, collections.deque, and memoryview

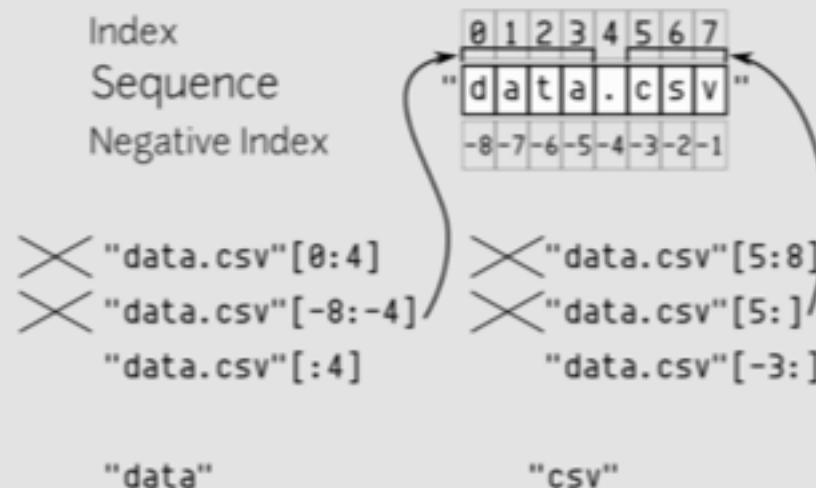
- Immutable sequences

- tuple, str, and bytes

## Index Examples



## Slicing Examples



# **2. String**

## ■ 문자열

- 문자열은 글자 혹은 글자가 모여서 만드는 단어나 문장
- 크게는 이런 단어, 문장이 모여서 여러 줄의 단락이나 글 전체가 하나의 문자열

## ■ 리터럴

- 큰 따옴표와 작은 따옴표를 구분하지 않지만 양쪽 따옴표가 맞아야 함. (문자열을 둘러싸는 따옴표와 다른 따옴표는 문자열 내의 일반 글자로 해석)
  - "apple" , 'apple' 은 모두 문자열 리터럴로 apple이라는 단어를 표현
- 두 개의 문자열 리터럴이 공백이나 줄바꿈으로 분리되어 있는 경우에 이것은 하나의 문자열 리터럴로 해석
  - "apple," "banana"는 "apple,banana"라고 쓴 표현과 동일
- 따옴표 세 개를 연이어서 쓰는 경우에는 문자열 내에서 줄바꿈이 허용.
  - 흔히 함수나 모듈의 간단한 문서화 텍스트를 표현할 때 쓰임.
  - """He said "I didn't go to 'SCHOOL' yesterday.""" > He said "I didn't go to 'SCHOOL' yesterday".
  - 여러 줄에 대한 내용을 쓸 때. """HOMEWORK: 1. print "hello, world" 2. print even number between 2 and 12 3. calculate sum of prime numbers up to 100,000 """
- 이스케이프를 허용하지 않는 raw string 리터럴 : r"
- 다른 값을 삽입하는 format string 리터럴 f;;
  - 바이트 배열을 정의하는 byte string 리터럴 b"

## ■ immutable

- Cannot be modified in place; if you want a different string, use a different string

## ■ The str type holds Unicode characters

- There is no type for single characters

## ■ The bytes type is used for strings of bytes, i.e., ASCII or Latin-1

- The bytearray type is used for modifiable byte sequences

## ■ 문자열 연산

- 문자열 + 문자열 : 두 문자열을 연결
- 문자열 in 문자열 : 문자열 내의 특정 글자가 있는지 검사
  - 'a' in 'apple', 'c' not in 'banana'
- 문자열 \* 정수 : 문자열을 정수값만큼 반복하여 문자열을 만듦
  - 'abc' \* 3 --> 'abcababc'
- 문자열에 실수를 곱하거나 문자열에 정수를 더하는 연산 ?
- ValueError 예러

## Immutable Strings

### Code

```
name = 'matt'
```

### What Computer Does

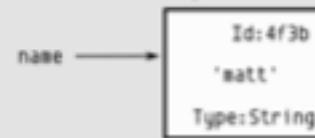
Variables      Objects



### Step 1: Python creates a string

```
correct = name.capitalize()
```

Variables      Objects



### Step 2: Python creates a new capitalized string

```
correct = name.capitalize()
```

Variables      Objects



### Step 3: Python creates a new variable

Type	Example
String	"hello\\tthere"
String	'hello'
String	'''He said, "hello"'''
Raw string	r'hello\\tthere'
Byte string	b'hello'

```
print(...)
```

'Single-quoted string'  
"Double-quoted string"  
'String with\nnewline'

'Unbroken\nstring'

r'\n is an escape code'  
'Pasted' '' 'string'

('Pasted'  
''  
'')

'string')

"""String with  
newline"""

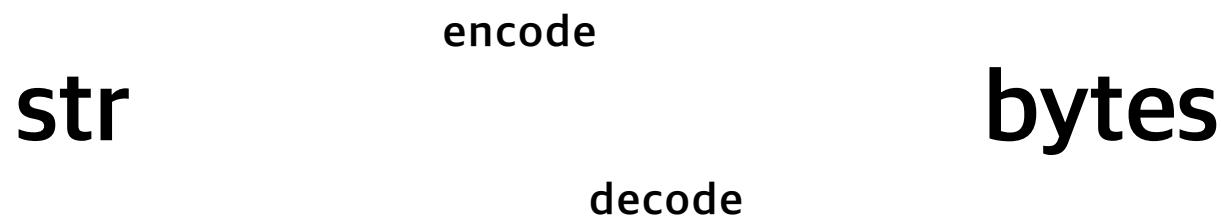
gives...

Single-quoted string  
Double-quoted string  
String with  
newline  
Unbroken string

\n is an escape code  
Pasted string  
Pasted string

String with  
newline

- bytes and str are immutable sequence types used for holding strings
  - str is Unicode and bytes holds, well, bytes
  - By default, encoding and decoding between str and bytes is based on UTF-8



Built-in query functions

**chr(codepoint)**  
**len(string)**  
**ord(character)**  
**str(value)**

Substring containment

**substring in string**  
**substring not in string**

String concatenation

**first + second**  
**string \* repeat**  
**repeat \* string**

String comparison (lexicographical ordering)

**lhs == rhs**  
**lhs != rhs**  
**lhs < rhs**  
**lhs <= rhs**  
**lhs > rhs**  
**lhs >= rhs**

- Strings can be indexed, which results in a string of length 1
- As strings are immutable, a character can be subscripted but not assigned to
- Index 0 is the initial character
- Indexing past the end raises an exception
- Negative indexing goes through the string in reverse
- I.e., -1 indexes the last character

### ■ It is possible to take a slice of a string by specifying one or more of...

- A start position, which defaults to 0
- A non-inclusive end position (which may be past the end, in which case the slice is up to the end), which defaults to len
- A step, which defaults to 1

### ■ With a step of 1, a slice is equivalent to a simple substring

string[index]	General form of subscript operation
string[0]	First character
string[len(string) – 1]	Last character
string[-1]	Last character
string[-len(string)]	First character
string[first:last]	Substring from first up to last
string[index:]	Substring from index to end
string[:index]	Substring from start up to index
string[:]	Whole string
string[first:last:step]	Slice from first to last in steps of step
string[::-step]	Slice of whole string in steps of step

string.join(strings)	Join each element of strings with string
string.split()	Split into a list based on spacing
string.split(separator)	Split into a list based on separator
string.replace(old, new)	Replace all occurrences of old with new
string.strip()	Strip leading and trailing spaces
string.lower()	All to lower case
string.upper()	All to upper case
string.startswith(text)	Whether starts with text substring
string.endswith(text)	Whether ends with text substring
string.count(text)	Number of occurrences of text
string.find(text)	Index of first occurrence of text, or -1
string.index(text)	Like find, but ValueError raised on fail
string.isalpha()	True if all characters are alphabetic
string.isdecimal()	True if all characters are decimal digits
string.islower()	True if all characters are lower case
string.isupper()	True if all characters are lower case

## ■ 내삽 (interpolation)

### ○ 내삽은 문자열에 대한 특별한 연산

- 예를 들어 "Tom has 3 bananas and 4 apples." 라는 문자열이 있다고 할 때, 이것을 리터럴로 정의하는 것은 그 내용이 소스코드에 고정되는, 이른 바 하드 코딩(hard coding)이다. 문자열은 한 번 생성된 이후로 변경되지 않는 불변의 고정값이므로 Tom이 가지고 있는 사과나 바나나의 개수가 바뀌었을 때, 그 내용을 적절히 변경해 줄 수가 없다.
- 내삽(interpolation)은 문자열내에 동적으로 변할 수 있는 값을 삽입하여 상황에 따라 다른 문자열을 만드는 방법이다. 문자열 내삽의 기본원리는 문자열 내에 다른 값으로 바뀔 치환자를 준비해두고, 필요한 시점에 치환자를 실제 값의 내용으로 바꿔 문자열을 생성하는 것이다. 내삽의 방법에는 다음의 세 가지 방법이 있다.
  - 전통적인 포맷 치환자를 사용하는 방법 : 문자열 % (값, ...)의 형식을 이용해서 문자열 내로 변수값을 밀어넣는 방법
  - 문자열의 .format() 메소드를 사용하는 방법 : 치환자의 구분없이 사용할 수 있으며, 각 값을 포맷팅할 수 있는 장점이 있다.
  - 문자열 포맷 리터럴을 사용하는 방법 : 포맷 메소드를 사용하지 않고 리터럴만으로 2.의 방법을 사용할 수 있다 (파이썬 3.6 이상)

## ■ 전통적인 포맷 치환 방법

- 문자열 내에 어떤 값을 집어 넣는 것에 대한 필요는 아주 오래전부터 있어왔고, 이러한 치환자의 종류와 형식은 파이썬이 만들어지기 이전부터 일종의 표준으로 정의되어 자리잡고 있었다. 문자열 치환자는 퍼센트 문자 뒤에 포맷형식을 붙여서 치환자를 정의한다. 치환자를 포함하는 문자열과 각 치환자에 해당하는 값의 튜플을 (튜플을 아직 배우지는 못했지만...) % 기호로 연결하여 표현한다.
- 주요 치환자에는 다음과 같은 종류가 있다.
- %d : 정수값을 나타낸다. d 앞에는 자리수와 채움문자를 넣을 수 있다. 예를 들어 %04d 라고 쓰면 앞의 0은 채움문자이고 뒤는 포맷의 폭이다. 즉 %04d 는 0으로 시작하는 네자리 정수를 의미한다. 13이라는 값을 포맷팅 할 때, %d 에 치환하면 "13"이 되지만, %04d 에 치환되는 경우에는 "0013"으로 치환된다.
- %f : 실수값을 나타낸다. f 앞에는 .3 과 같이 소수점 몇 째자리까지 표시할 것인지를 결정하는 확장정보를 넣을 수 있다. "%.3f" 라는 템플릿은 1.5를 "1.500"으로 표시해준다.
- 그 외에 정수값은 숫자 리터럴과 같이 %b, %o, %x를 이용해서 각각 이진법, 8진법, 16진법으로 표시할 수 있다.
- %s : 문자열을 의미한다.
- %r : "representation"으로 타입을 구분하지 않는 값의 표현형을 말한다. 표시되고자 하는 값의 타입이 분명하지 않을 때 사용한다. 위에서 소개된 %d, %f, %s 에 대해서 올바르지 않은 타입의 값을 치환하려 하면 TypeError가 발생한다. 이럴 때 사용할 수 있다.

## ■ format 메소드를 사용하는 방법

- 파이썬의 기본 내장함수 중에서도 format() 함수가 있는데, 이 함수도 문자열의 format 메소드와 동일한 동작을 한다. format(문자열, 치환값)`의 형식으로 사용하며 그외 내용은 이 절에서 설명하는 것과 동일하다.
- 전통적인 치환자구분이 타입을 가린다는 제약이 있고, 포맷팅의 방법이 제한되어 있다는 부분때문에 최근에 대세를 이루는 방식이다. 이 방식에서는 %d 와 같은 표현 대신에, {} 를 사용한다. {} 자체가 하나의 값을 의미하며, 번호를 부여해서 한 번 받은 값을 여러번 사용할 수 있다. 이 포맷팅 방식을 미니포맷이라고 불리는데 대략 다음과 같은 문법을 가지고 있다.
- { 치환자 } 치환자 ::= [필드이름] [!변환형] [: 포맷정의] 포맷정의 ::= [[채우기]정렬][부호][#][0][폭][그룹옵션][.소수점자리][타입]
- 뭔가 포맷방식이 엄청난데, 채우기와 폭은 전통적인 포맷 치환자에서도 지원하던 것인데, 왼쪽 정렬이나 중앙정렬도 설정할 수 있다. 숫자값인 경우에는 특별히 d, f, x, b, o 등의 표현타입도 정의해줄 수 있다.

### ■ Strings support format, a method for formatting multiple arguments

- Default-ordered, positional and named parameter substitution are supported

```
'First {}, now {}!'.format('that', 'this')
```

```
'First {0}, now {1}!'.format('that', 'this')
```

```
'First {1}, now {0}!'.format('that', 'this')
```

```
'First {0}, now {0}!'.format('that', 'this')
```

```
'First {x}, now {y}!'.format(x='that', y='this')
```

Use {{ and }} to embed { and } in a string without format substitution

### ■ Strings also support a printf-like approach to formatting

- The mini-language is based on C's printf, plus some additional features
- Its use is often discouraged as it is error prone for large formatting tasks

```
'First %s, now %s!' % ('that', 'this')
```

```
'First %(x)s, now %(y)s!' % {'x': 'that', 'y': 'this'}
```

```
'First %d, now %d!' % (1, 2)
```

<b>Escape Sequence</b>	<b>Output</b>
\n newline	Ignore trailing newline in triple quoted string
\\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	ASCII Backspace
\n	Newline
\r	ASCII carriage return
\t	Tab
\u12af	Unicode 16 bit
\U12af89bc	Unicode 32 bit
N{BLACK STAR}	Unicode name
\o84	Octal character
\xFF	Hex character

## ■ The first statement of a function, class or module can optionally be a string

- This docstring can be accessed via `__doc__` on the object and is used by IDEs and the help function
- Conventionally, one or more complete sentences enclosed in triple quotes

```
def echo(strings):  
    """Prints space-adjoined sequence of strings."""  
    print(' '.join(strings))
```

# **Built-in Containers**

# **Another Sequence**

- range, tuple and list are **sequence** types
  - Sequence types hold values in an **indexable** and **sliceable** order
- range, tuple and frozenset are **immutable** containers
- set and frozenset hold unique values
- dict is a mutable mapping type

- Built-in containers address most needs
- There are some common iteration patterns to follow (and avoid)
- Comprehensions address many common container iteration needs
- collections module offers variations on standard sequence and lookup types
- collections.abc supports container usage and definition

- Although everything is, at one level, a dict, dict is not always the best choice
  - Nor is list
- Choose containers based on usage patterns and mutability
  - E.g., tuple is immutable whereas list is not
  - Iteration is the most common container activity, so favour the most direct and efficient iteration style

<b>list</b>	list() [] [0, 0x33, 0xCC] ['Albert', 'Einstein', [1879, 3, 14]] [random() for _ in range(42)]
<b>tuple</b>	tuple() () (42,) ('red', 'green', 'blue') ((0, 0), (3, 4))
<b>range</b>	range(42) range(1, 100) range(100, 0, -1)

## ■ All container types — including range and str — are iterable

- Can appear on right-hand side of in (for or membership) or of a multiple assignment
- Except for text and range types, containers are heterogeneous

```
first, second, third = [1, 2, 3]
head, *tail = range(10)
*most, last = 'Hello'
```

Equality comparison

`lhs == rhs`  
`lhs != rhs`

Returns sorted list of container's values (also takes keyword arguments for key comparison — `key` — and reverse sorting — `reverse`)

Built-in queries and predicates

`len(container)`  
`min(container)`  
`max(container)`  
`any(container)`  
`all(container)`  
`sorted(container)`

Membership (key membership for mapping types)

`value in container`  
`value not in container`

- A tuple is an immutable sequence
- Supports (negative) indexing, slicing, concatenation and other operations
- A list is a mutable sequence
- It supports similar operations to a tuple, but in addition it can be modified
- A range is an immutable sequence
- It supports indexing, slicing and other operations

Search methods

`sequence.index(value)`  
`sequence.count(value)`

Raises exception if value not found



Indexing and slicing

`sequence[index]`  
`sequence[first:last]`  
`sequence[index:]`  
`sequence[:index]`  
`sequence[:]`  
`sequence[first:last:step]`  
`sequence[::-step]`

Lexicographical ordering (not for range)

`lhs < rhs`  
`lhs <= rhs`  
`lhs > rhs`  
`lhs >= rhs`

Concatenation (not for range)

`first + second`  
`sequence * repeat`  
`repeat * sequence`

- Tuples are the default structure for unpacking in multiple assignment
- The display form of tuple relies on parentheses
- Thus it requires a special syntax case to express a tuple of one item

```
x = 1,  
x = 1, 2  
x, y = 1, 2  
x = 1, (2, 3)
```

```
x = (1,)  
x = (1, 2)  
(x, y) = (1, 2)  
x = (1, (2, 3))
```

- As lists are mutable, assignment is supported for their elements
- Augmented assignment on subscripted elements
- Assignment to subscripted elements and assignment through slices
- List slices and elements support del
- Removes them from the list

Index- and slice-based operations

`list[index]`

`list[index] = value`

`del list[index]`

`list[first:last:step]`

`list[first:last:step] = other`

`del list[first:last:step]`

Whole-list operations

`list.clear()`

`list.reverse()`

`list.sort()`

Element modification methods

`list.append(value)`

`list.insert(index, value)`

`list.pop()`

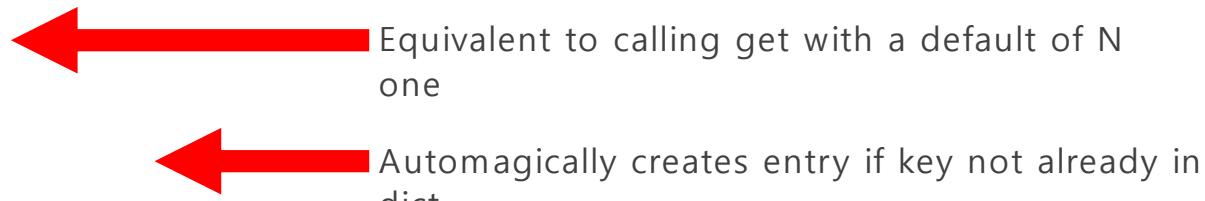
`list.pop(index)`

`list.remove(value)`

# **Dictionary**

- dict is a mapping type
  - i.e., it maps a key to a correspond value
  - Keys, values and mappings are viewable via keys, values and items methods
- Keys must be of immutable types
  - This includes int, float, str, tuple, range and frozenset, but excludes list, set and dict
  - Immutable types are hashable (hash can be called on them)

## Key-based operations

`key in dict``key not in dict``dict.get(key, default)``dict.get(key)``dict[key]``dict[key] = value``del dict[key]`

## Manipulation methods

## Views

`dict.keys()``dict.values()``dict.items()``dict.clear()`  
`dict.pop(key)`  
`dict.pop(key, default)`  
`dict.get(key)`  
`dict.get(key, default)`  
`dict.update(other)`

# **Set & frozenset**

- **set and frozenset both define containers of unique hashable values**
- **set is mutable and has a display form — note that set() is the empty set, not {}**
- **frozenset is immutable and can be constructed from a set or other iterable**

```
text = 'the cat sat on the mat'  
words = frozenset(text.split())  
print('different words used:', len(words))
```

Set relations (in addition to equality and set membership)

`lhs < rhs`  
`lhs <= rhs lhs.issubset(rhs)`  
`lhs > rhs`  
`lhs >= rhs lhs.issuperset(rhs)`  
`lhs.isdisjoint(rhs)`

Set combinations

`lhs | rhs lhs.union(rhs)`  
`lhs & rhs lhs.intersection(rhs)`  
`lhs - rhs lhs.difference(rhs)`  
`lhs ^ rhs lhs.symmetric_difference(rhs)`

## Manipulation methods

set.add(element)  
set.remove(element)  
set.discard(element)  
set.clear()  
set.pop()

Removes and return  
s arbitrary element



## Augmented assignment and updates

lhs |= rhs lhs.update(rhs)  
lhs &= rhs lhs.intersection\_update(rhs)  
lhs -= rhs lhs.difference\_update(rhs)  
lhs ^= rhs lhs.symmetric\_difference\_update(rhs)

## Manipulation methods

set.add(element)  
set.remove(element)  
set.discard(element)  
set.clear()  
set.pop()

Removes and return  
s arbitrary element



## Augmented assignment and updates

lhs |= rhs lhs.update(rhs)  
lhs &= rhs lhs.intersection\_update(rhs)  
lhs -= rhs lhs.difference\_update(rhs)  
lhs ^= rhs lhs.symmetric\_difference\_update(rhs)

# 형변환 (Type Conversion)

int('314')	314
str(314)	'314'
str([3, 1, 4])	'[3, 1, 4]'
list('314')	['3', '1', '4']
set([3, 1, 4, 1])	{1, 3, 4}

# **Control Flow**

**From the conditional to the exceptional**

# 조건문

Condition-based loop

p

**while** condition:

...

**else:**

...

Exception handling

**try:**

...

**except** exception:

...

**else:**

**finally:**

...

Multiway conditional

**if** condition:

...

**elif** condition:

...

**else:**

...

No-op

**pass**

Value-based loop

**for** variable **in** sequence:

...

**else:**

...

- Logic is not based on a strict Boolean type, but there is a bool type
- The and, or and if else operators are partially evaluating
- There is no switch/case

## Relational

**==** Equality  
**!=** Inequality  
**<** Less than  
**<=** Less than or equal to  
**>** Greater than  
**>=** Greater than or equal to

## Boolean

**and** Logical and  
**or** Logical or  
**not** Negation

## Identity

**is** Identical  
**is not** Non-identical

Membership and containmen  
t

**in** Membership  
**not in** Non-membership

## Conditional

**if else** Conditional (ternary) operator

False  
None  
0  
0.0  
"  
b"  
[]  
{}  
( )

True  
Non-null references and not otherwise  
false  
Non-zero numbers  
Non-empty strings

Non-empty containers

Other empty containers with  
non-literal empty forms are also  
considered false.



**not None == True** which means that **not (any function that returns None) == True**

- The logical operators (and & or)
- partially evaluating
- I.e., they do not evaluate their right-hand operand if they do not need to
- They return the last evaluated operand, without any conversion to bool

'Hello' and 'World'

'World'

'Hello' or 'World'

'Hello'

[] and [3, 1, 4]

[]

{42, 97} or {42}

{42, 97}

{0} and {1} or [2]

{1}

- without using an intermediate and
- The **short-circuiting evaluation** is the same, i.e., if the first comparison is false, no further evaluation occurs

Brief form...

$\text{minimum} < \text{value} < \text{maximum}$

" != selection in options

Equivalent to...

$\text{minimum} < \text{value}$  **and**  $\text{value} < \text{maximum}$

" != selection **and** selection in options

## ■ There is no header–suite delineation, as for statements

- The else is mandatory
- There is no elif

## ■ Ternary Operator

- Python has its own ternary operator, called a *conditional expression* (see PEP 308). These are handy as they can be used in comprehension constructs and lambda functions:
- >>> last = 'Lennon' if band == 'Beatles' else 'Jones' Note that this has similar behavior to an if statement, but it is an expression, and not a statement. Python distinguishes these two. An easy way to determine between the two, is to remember that an expression follows a return statement. Anything you can return is an expression.

```
user = input('Who are you? ')
user = user.title() if user else 'Guest'
```

예외처리



assert

raise

try

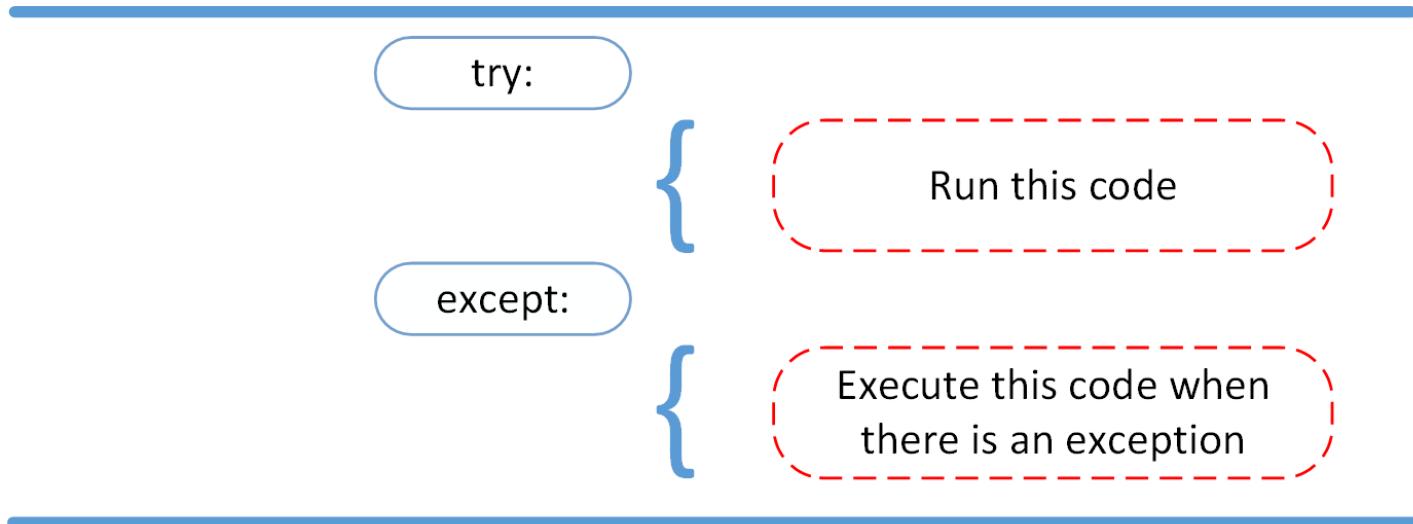
except

else

finally

- normally used to signal error or other undesirable events
- But this is not always the case, e.g., StopIteration is used by iterators to signal the end of iteration
- objects and their type is defined in a class hierarchy
- The root of the hierarchy is BaseException
- But yours should derive from Exception

- An exception is a signal (with data) to discontinue a control path
- A raised exception propagates until handled by an except in a caller



```
prompt = ('What is the airspeed  
velocity '
```

```
        'of an unladen swallow? ')
```

```
try:  
    response = input(prompt)  
except:  
    response = "
```

- Exceptions can be handled by type
- except statements are tried in turn until there is a base or direct match

The most specific exception type: ZeroDivisionError derives from ArithmeticError

try:

...

except ZeroDivisionError:

...

except ArithmeticError:

...

except:

...

The most general match handles any remaining exceptions, but is not recommended

- A handled exception can be bound to a variable
- The variable is bound only for the duration of the handler, and is not accessible after

```
try:  
    ...  
except ZeroDivisionError as byZero:  
    ...  
except ArithmeticError:  
    ...  
except Exception as other:  
    ...
```

byZero is only accessible here

- It is possible to define an except handler that matches different types
- A handler variable can also be assigned

```
try:  
    ...  
except (ZeroDivisionError, OverflowError):  
    ...  
except (EOFError, OSError) as external:  
    ...  
except Exception as other:  
    ...
```

- A `raise` statement specifies a class or an object as the exception to raise
- If it's a class name, an object instantiated from that class is created and raised

Use `raise` to force an exception:



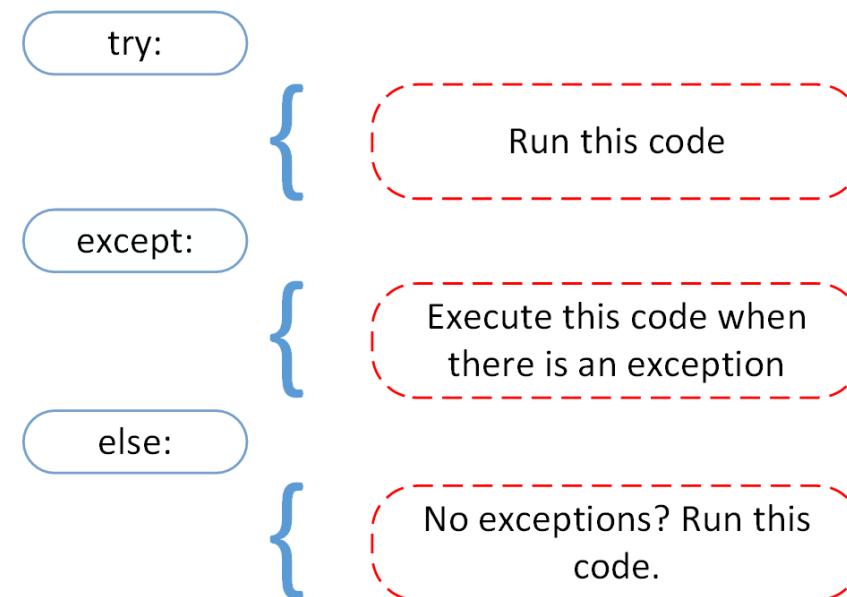
Re-raise current handled exception  
Raise a new exception chained from an existing exception named `caughtException`

`raise Exception`  
`raise Exception("I don't know that")`  
`raise`  
`raise Exception from caughtException`  
`raise Exception from None`

Raise a new exception ignoring any previous exception

■ A try and except can be associated with an else statement

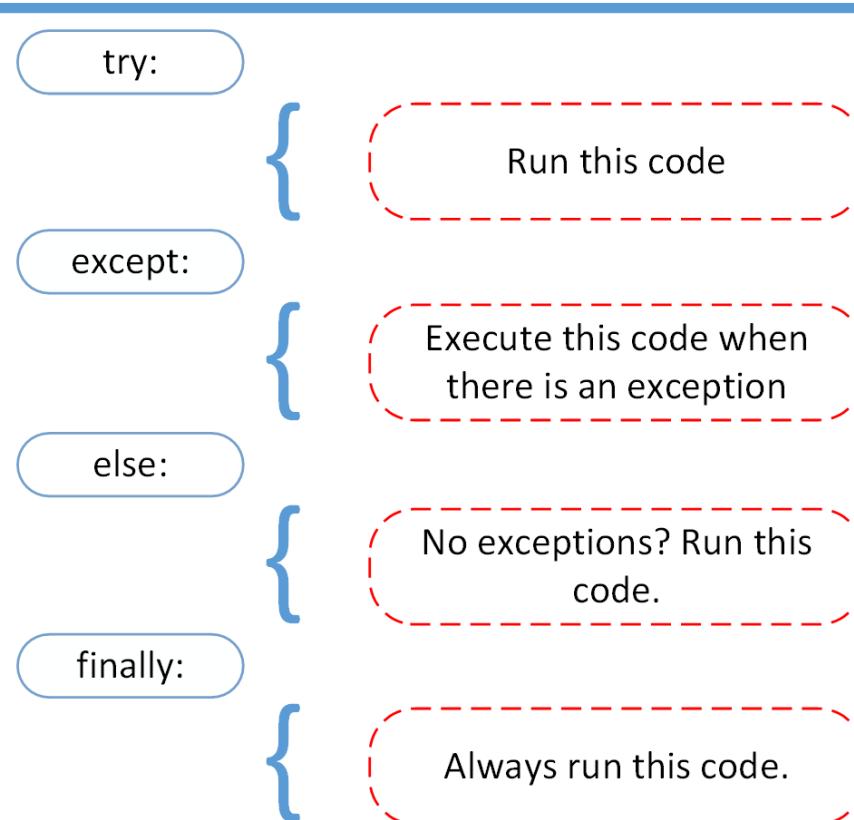
■ This is executed after the try if and only if no exception was raise



```
try:  
    hal.open(pod_bay.doors())  
except SorryDave:  
    airlock.open()  
else:  
    pod_bay.park(pod)
```

■ A **finally** statement is executed whether an exception is raised or not

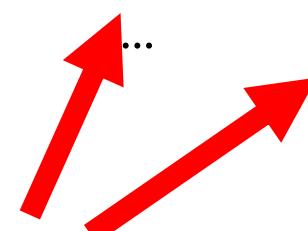
■ Useful for factoring out common code from **try** and **except** statements, such as clean-up code



```
log = open('log.txt')
try:
    print(log.read())
finally:
    log.close()
```

`try:``...  
except:`

One or more except  
handlers, but no m  
ore than a single fi  
nally or else

`try:``...  
finally:``try:``...  
except:``...  
finally:``try:``...  
except:``...  
else:``try:``...  
except:``...  
else:``...  
finally:`

else cannot appear unles  
s it follows an except

### ■ Don't catch an exception unless you know what to do with it

- And be suspicious if you do nothing with it

### ■ Always use `with` for resource handling, in particular file handling

- And don't bother checking whether a file is open after you open it

### ■ Use exceptions rather than error codes

## ■ assert statement

- check invariants in program code
  - Can also be used for ad hoc testing
- 

Assert that a condition is met:

assert:

{

Test if condition is True

---

Equivalent to...

assert value is not None  
if \_\_debug\_\_ and not (value is not None):  
    raise AssertionError

Equivalent to...

assert value is not None, 'Value missing'  
if \_\_debug\_\_ and not (value is not None):  
    raise AssertionError('Value missing')

# Functions

- Functions are **first-class objects**
- Python supports **lambda expressions** and **nested function definitions**
- Argument values can be defaulted
- Functions can take both positional arguments and keyword arguments
- Functions can be defined to take an arbitrary number of arguments

## ■ first class function

- it is "first class" in the type system: that is, that functions themselves are values. In languages where functions are not first class, functions are defined as a relationship between values, which makes them "second class".
- you can use a function as a value, which means (among other things) that you can pass functions into functions.
  - You couldn't do this in Java before Java 7 (reflection doesn't count), so that's an example of a programming language only having "second class" functions. In order to "pass a function", you had to define a type where that function could live (as a method), and then pass an instance of that type.

## ■ So, in Python, all functions are first class, because all functions can be used as a value.

## ■ higher order function

- a function that takes a function as an argument. Not all functions in Python are higher-order, because not all functions take another function as an argument. But even the functions that are not higher-order are first class. (It's a square/rectangle thing.)

- Can be called both with positional and with keyword arguments
- Can be defined with default arguments
- Can return multiple values

```
def roots(value):  
    from math import sqrt  
    result = sqrt(value)  
    return result, -result
```

```
first, second = roots(4)
```

```
def is_at_origin(x, y):  
    return x == y == 0
```

```
is_at_origin(0, 1)
```

### ■ Functions can be passed as arguments and can be used in assignment

- This supports many callback techniques and functional programming idioms

### ■ Conversely, any object that is callable can be treated as a function

### ■ callable is a built-in predicate function

### ■ Functions always return a value

### ■ None if nothing is returned explicitly

### ■ High-Order Function

- 함수를 파라미터로 전달 받는 함수

- 함수를 리턴하는 함수

### ■ First-class function

- 함수 function을 first-class citizen으로 취급

- 함수 자체를 인자 (argument) 로써 다른 함수에 전달하거나 다른 함수의 결과값으로 리턴 할 수도 있고, 함수를 변수에 할당하거나 데이터 구조안에 저장할 수 있는 함수를 뜻합니다.

```
def does_nothing():
    pass

assert callable(does_nothing)
assert does_nothing() is None
```

```
unsorted = ['Python', 'parrot']
print(sorted(unsorted, key=str.lower))
```



Keyword argument

```
def is_even(number):
    return number % 2 == 0

def print_if(values, predicate):
    for value in values:
        if predicate(value):
            print(value)

print_if([2, 9, 9, 7, 9, 2, 4, 5, 8], is_even)
```

■ **def a():**

○ return 0

■ **def b():**

○ print(0)

■ **a() + 1**

■ **b() + 1**

- A lambda is simply an expression that can be passed around for execution
- It can take zero or more arguments
- As it is an expression not a statement, this can limit the applicability
- Lambdas are anonymous, but can be assigned to variables
- But defining a one-line named function is preferred over global lambda assignment

```
def print_if(values, predicate):
    for value in values:
        if predicate(value):
            print(value)
```

```
print_if(
    [2, 9, 9, 7, 9, 2, 4, 5, 8],
    lambda number: number % 2 == 0)
```



An ad hoc function that takes a single argument — in this case, `number` — and evaluates to a single expression — in this case, `number % 2 == 0`, i.e., whether the argument is even or not.

- Create a dict mapping airport codes to airport names
- Select 3 to 6 airports
- Use sorted to sort by...
- Airport code
- Airport name
- Longitude and/or latitude — either extend the mapping from just names or introduce an additional dict with positions

■ Function definitions can be nested

■ Each invocation is bound to its surrounding scope, i.e., it's a closure

```
def logged_execution(action, output):
    def log(message):
        print(message, action.__name__, file=output)
    log('About to execute')
    try:
        action()
        log('Successfully executed')
    except:
        log('Failed to execute')
        raise
```

```
world = 'Hello'  
def outer_function():  
    def nested_function():  
        nonlocal world  
        world = 'Ho'  
    world = 'Hi'  
    print(world)  
    nested_function()  
    print(world)
```



Refers to any world that will be assigned in within outer\_function, but not the global world

```
outer_function()  
print(world)
```

Hi  
Ho  
Hello

- The defaults will be substituted for corresponding missing arguments
- Non-defaulted arguments cannot follow defaulted arguments in the definition

```
def line_length(x=0, y=0, z=0):  
    return (x**2 + y**2 + z**2)**0.5
```

```
line_length()  
line_length(42)  
line_length(3, 4)  
line_length(1, 4, 8)
```

- Defaults evaluated once, on definition, and held within the function object
- Avoid using mutable objects as defaults, because any changes will persist between function calls
- Avoid referring to other parameters in the parameter list — this will either not work at all or will appear to work, but using a name from an outer scope

- A function can be defined to take a variable argument list
- Variadic arguments passed in as a tuple
- Variadic parameter declared using \* after any mandatory positional arguments
- This syntax also works in assignments

```
def mean(value, *values):  
    return sum(values, value) / (1 + len(values))
```

- To apply values from an iterable, e.g., a tuple, as arguments, unpack using \*
- The iterable is expanded to become the argument list at the point of call

```
def line_length(x, y, z):  
    return (x**2 + y**2 + z**2)**0.5
```

```
point = (2, 3, 6)  
length = line_length(*point)
```

- Reduce the need for chained builder calls and parameter objects
- Keep in mind that the argument names form part of the function's public interface
- Arguments following a variadic parameter are necessarily keyword arguments

```
def date(year, month, day):  
    return year, month, day
```

```
sputnik_1 = date(1957, 10, 4)  
sputnik_1 = date(day=4, month=10, year=1957)
```

- A function can be defined to receive arbitrary keyword arguments
- These follow the specification of any other parameters, including variadic
- Use `**` to both specify and unpack
- Keyword arguments are passed in a dict — a keyword becomes a key
- Except any keyword arguments that already correspond to formal parameters

```
def present(*listing, **header):
    for tag, info in header.items():
        print(tag + ': ' + info)
    for item in listing:
        print(item)

present(
    'Mercury', 'Venus', 'Earth', 'Mars',
    type='Terrestrial', star='Sol')
```

```
type: Terrestrial
star: Sol
Mercury
Venus
Earth
Mars
```

- A function's parameters and its result can be annotated with expressions
- No semantic effect, but are associated with the function object as metadata, typically for documentation purposes

```
def is_leap_year(year : 'Gregorian') -> bool:  
    return year % 4 == 0 and year % 100 != 0 or year % 400 == 0
```

`is_leap_year.__annotations__`

`{'return': <class 'bool'>, 'year': 'Gregorian'}`

- A function definition may be wrapped in decorator expressions
- A decorator is a function that transforms the function it decorates

```
class List:  
    @staticmethod  
    def nil():  
        return []  
  
    @staticmethod  
    def cons(head, tail):  
        return [head] + tail
```

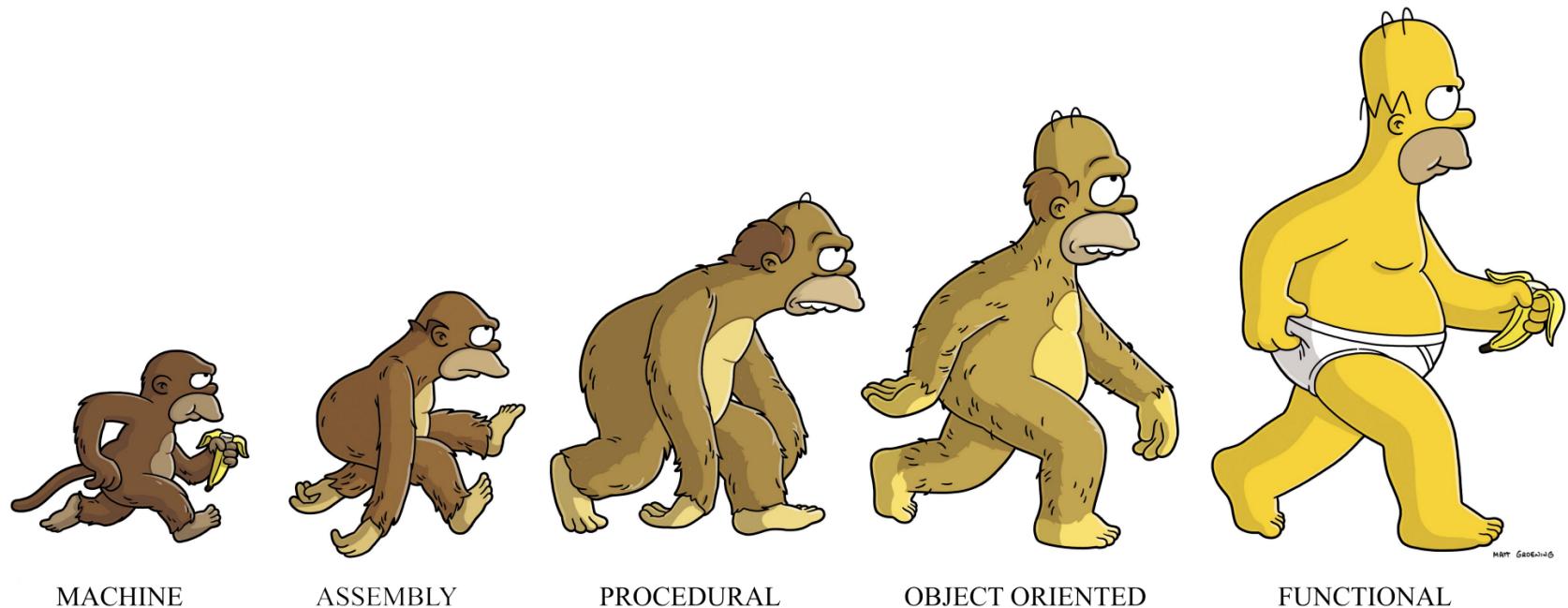
nil = List.nil()  
[]

one = List.cons(1, nil)  
[1]

two = List.cons(2, one)  
[2, 1]

# Functional Thinking

- ✓ Functional programming techniques, tips & tricks



## ■ 함수형 언어

- 부작용 없는 프로그래밍을 지원하고 장려하는 언어
- 가능한한 부작용을 제거하고 그렇지 않은 곳에는 철저히 제어 할 수 있도록 적극적으로 도와주는 언어
- 더 적극적이고 더 격렬하게 부작용에 적대적인 언어. 부작용은 복잡성이고, 복잡성은 버그이며, 버그는 악마이다. 함수형 언어는 여러분들도 부작용에 적대적이 되도록 도와줄 것이다. 여러분과 함께 그들(부작용, 복잡성, 버그)을 깨부시고 굴복시킬 것이다.
- 부작용을 완전히 피할 수는 없다. 대부분의 프로그램은 반환 값을 얻기 위해서가 아니라 어떤 동작을 하기 위해 실행하기 때문이다. 하지만 프로그램 내부에서는 엄격하게 통제하고자 한다. 우리는 가능한 모든 곳에서 부작용(과 부원인)을 제거하고, 또 제거할 수 없는 경우에는 철저하게 통제할 것이다.
- 코드 조각이 필요로 하는 것과 유발하게 될 결과를 숨기지 말자. 코드 조각이 제대로 실행하기 위해 뭔가를 필요로 한다면 그렇게 말하자. 뭔가 유용한 일을 한다면 출력 형태로 선언하자. 이렇게 한다면 우리의 코드는 더 명확해 질 것이다. 복잡성이 표면에 드러나고 우리는 그것을 분해하여 처리할 수 있을 것이다.

## ■ Functions are the principal units of composition

## ■ Functions are pure, i.e., no side effects

- 숨겨진 출력은 “부작용(side-effect)”으로, 숨겨진 입력은 “부원인(side-cause)”
- 모든 입력이 입력으로 선언되고 (숨겨진 것이 없어야 한다) 마찬가지로 모든 출력이 출력으로 선언된 함수를 ‘순수(pure)’하다고 부른다.

## ■ Immutability, i.e., query and create state rather than modify it

## ■ 파이썬

- 자바에서의 기본적인 부작용 패턴을 잠깐 살펴보자.

- ```
public String getName() {  
    return this.name;  
}
```

- 이 함수를 어떻게 순수하게 만들 수 있을까? `this`가 숨겨진 입력이므로 인자로 끌어올리기만 하면 된다.

- ```
public String getName(Person this) {  
    return this.name;  
}
```

- 이제 `getName`은 순수 함수이다. 파이썬은 기본적으로 이 두 번째 패턴을 채택하고 있음을 주목하자. 파이썬에서, 모든 객체 메소드는 `this`를 첫번째 인자로 가진다. 다만 그것을 `self`라고 부를 뿐이다.

- ```
def getName(self):  
    self.name
```

- 분명히 명시적인 것이 묵시적인 것보다 낫다.

- Immutability...
- Prefer to return new state rather than modifying arguments and attributes
- Resist the temptation to match every query or get method with a modifier or set
- Expressiveness...
- Consider where loops and intermediate variables can be replaced with comprehensions and existing functions

- Python is reference-based language, so objects are shared by default
- Easily accessible data or state-modifying methods can give aliasing surprises
- Note that true and full object immutability is not possible in Python
- Default arguments should be of immutable type, e.g., use tuple not list
- Changes persist between function calls

- A common feature of functional programming is fewer explicit loops
- Recursion, but note that Python does not support tail recursion optimisation
- Comprehensions — very declarative, very Pythonic
- Higher-order functions, e.g., map applies a function over an iterable sequence
- Existing algorithmic functions, e.g., min, str.split, str.join

## ■ Recursion may be a consequence of data structure traversal or algorithm

- E.g., iterating over a tree structure
- E.g., quicksort

## ■ But it can be used as an alternative to looping in many simple situations

## ■ But beware of efficiency concerns and Python's limits (see `sys.getrecursionlimit`)

```
def factorial(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

Two variables being modified explicitly

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

```
def factorial(n):
    if n > 0:
        return n * factorial(n - 1)
    else:
        return 1
```

No variables modified

There is a tendency in functional programming to favour expressions...

```
def factorial(n):
    if n > 0:
        return n * factorial(n - 1)
    else:
        return 1
```

```
def factorial(n):
    return n * factorial(n - 1) if n > 0 else 1
```

```
factorial = lambda n: n * factorial(n - 1) if n > 0 else 1
```



But using a lambda bound to a variable instead of a single-statement function is not considered Pythonic and means factorial lacks some metadata of a function, e.g., a good `__name__`.

- A comprehension is used to define a sequence of values declaratively
- Sequence of values defined by intension as an expression, rather than procedurally in terms of loops and modifiable state
- They have a select...from...where structure
- Many common container-based looping patterns are captured in the form of container comprehensions

```
def is_leap_year(year):  
    return year % 4 == 0 and year % 100 != 0 or year % 400 == 0
```

```
leap_years = []  
for year in range(2000, 2030):  
    if is_leap_year(year):  
        leap_years.append(year)
```

[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028]

Imperative list initialisation

```
leap_years = [year for year in range(2000, 2030) if is_leap_year(year)]
```

[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028]

List comprehension

- A higher-order function...
- Takes one or more functions as arguments
- Returns one or more function as its result
- Takes and returns functions
- Higher-order functions often used to abstract common iteration operations
- Hides the mechanics of repetition
- Comprehensions are often an alternative to such higher-order functions

- map applies a function over an iterable to produce a new iterable
- Can sometimes be replaced with a comprehension or generator expression that has no predicate
- Often shorter if no lambdas are involved

```
map(len, 'The cat sat on the mat'.split())
```

```
(len(word) for word in 'The cat sat on the mat'.split())
```

- filter includes only values that satisfy a given predicate in its generated result
- Can sometimes be replaced with a comprehension or generator expression that has a predicate
- Often shorter if no lambdas are involved

```
filter(lambda score: score > 50, scores)
```

```
(score for score in scores if score > 50)
```

- `functools.reduce` implements what is known as a **fold left operation**
- Reduces a sequence of values to a single value, left to right, with the accumulated value on the left and the other on the right

```
def factorial(n):  
    return reduce(lambda l, r: l*r, range(1, n+1), 1)
```

```
def factorial(n):  
    return reduce(operator.mul, range(1, n+1), 1)
```



`int.__mul__` would be a less general alternative

operator exports named functions corresponding to operators

Comparison operations

|                       |                       |
|-----------------------|-----------------------|
| <b>eq(a, b)</b>       | $a == b$              |
| <b>ne(a, b)</b>       | $a != b$              |
| <b>lt(a, b)</b>       | $a < b$               |
| <b>le(a, b)</b>       | $a <= b$              |
| <b>gt(a, b)</b>       | $a > b$               |
| <b>ge(a, b)</b>       | $a >= b$              |
| <b>is_(a, b)</b>      | $a \text{ is } b$     |
| <b>is_not(a, b)</b>   | $a \text{ is not } b$ |
| <b>contains(a, b)</b> | $a \text{ in } b$     |

Binary arithmetic operations

|                       |           |
|-----------------------|-----------|
| <b>add(a, b)</b>      | $a + b$   |
| <b>sub(a, b)</b>      | $a - b$   |
| <b>mul(a, b)</b>      | $a * b$   |
| <b>truediv(a, b)</b>  | $a / b$   |
| <b>floordiv(a, b)</b> | $a // b$  |
| <b>mod(a, b)</b>      | $a \% b$  |
| <b>pow(a, b)</b>      | $a^{**}b$ |

Unary operations

|                  |          |
|------------------|----------|
| <b>pos(a)</b>    | $+a$     |
| <b>neg(a)</b>    | $-a$     |
| <b>invert(a)</b> | $\sim a$ |

- Use reduce to implement your own version of map
- Can be written as a single-line method
- Return the result as a list

```
def my_map(func, iterable):
```

```
    ...
```

```
my_map(lambda n: n*2, [3, 1, 4, 1, 5, 9])
```

[6, 2, 8, 2, 10, 18]

In mathematics and computer science, **currying** is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument (partial application). It was introduced by Moses Schönfinkel and later developed by Haskell Curry.

<http://en.wikipedia.org/wiki/Currying>

- Nested functions and lambdas bind to their surrounding scope
- I.e., they are closures

```
def curry(function, first):  
    def curried(second):  
        return function(first, second)  
    return curried
```

```
def curry(function, first):  
    return lambda second: function(first, second)
```

```
hello = curry(print, 'Hello')  
hello('World')
```

Force non-positional use  
of subsequent parameter



```
def timed_function(function, *, report=print):
    from time import time
    def wrapper(*args):
        start = time()
        try:
            function(*args)
        finally:
            report(time() - start)
    return wrapper
```

```
wrapped = timed_function(long_running)
wrapped(arg_for_long_running)
```

- Values can be bound to a function's parameters using `functools.partial`
- Bound positional and keyword arguments are supplied on calling the resulting callable, other arguments are appended

```
quoted = partial(print, '>')
quoted()
                                print('>')

quoted('Hello, World!')
                                print('>', 'Hello, World!')

quoted('Hello, World!', end=' <\n')
                                print('>', 'Hello, World!', end=' <\n')
```

**min(iterable)**  
**min(iterable, default=value)**  
**min(iterable, key=function)**  
**max(iterable)**  
**max(iterable, default=value)**  
**max(iterable, key=function)**  
**sum(iterable)**  
**sum(iterable, start)**  
**any(iterable)**  
**all(iterable)**  
**sorted(iterable)**  
**sorted(iterable, key=function)**  
**sorted(iterable, reverse=True)**  
**zip(iterable, ...)**  
...

Default used if iterable is empty  
The function to transform the values before determining the lowest one



One or more iterables can be zipped together as tuples, i.e., effectively converting rows to columns, but zipping two iterables is most common

**chain**(iterable, ...)

**compress**(iterable, selection)

**dropwhile**(predicate, iterable)

**takewhile**(predicate, iterable)

**count()**

**count**(start)

**count**(start, step)

**islice**(iterable, stop)

**islice**(iterable, start, stop)

**islice**(iterable, start, stop, step)

**cycle**(iterable)

**repeat**(value)

**repeat**(value, times)

**zip\_longest**(iterable, ...)

**zip\_longest**(iterable, ..., fillvalue=fill)

...

# **Loop & Iterators & Generators**

## ■ for iterates over a sequence of values

- Over containers — e.g., string, list, tuple, set or dictionary — or other iterables
- Loop values are bound to one or more target variables

```
machete = ['IV', 'V', 'II', 'III', 'VI']
for episode in machete:
    print('Star Wars', episode)
```

```
for episode in 'IV', 'V', 'II', 'III', 'VI':
    print('Star Wars', episode)
```

## For Loops Create a Variable

Code

```
for letter in ['c', 'a', 't']:
    print(letter)
```

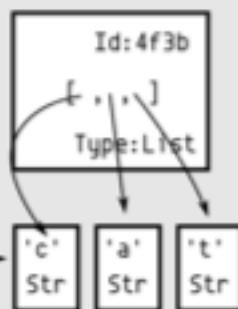
Output

c

What Computer Does

Variables      Objects

```
for letter in ['c', 'a', 't']:
    print(letter)
```



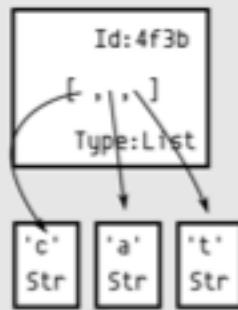
**Step 1: During start, letter points to c**

```
for letter in ['c', 'a', 't']:
    print(letter)
```

Output

c  
a

letter



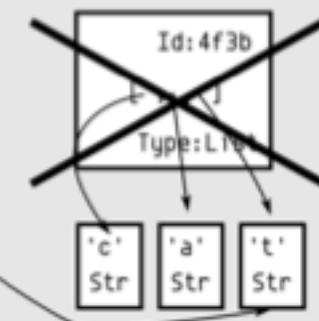
**Step 2: Then, letter points to a**

```
for letter in ['c', 'a', 't']:
    print(letter)
```

Output

c  
a  
t

letter



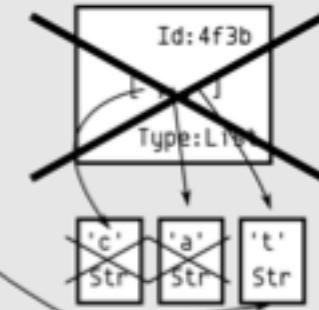
**Step 3: Finally, letter points to t, list removed**

```
for letter in ['c', 'a', 't']:
    print(letter)
```

Output

c  
a  
t

letter



**Step 4: Then, c and a are removed. Letter t remains**

**■ It is common to use range to define a number sequence to loop over**

- A range is a first-class, built-in object and doesn't allocate an actual list of numbers

```
for i in range(100):  
    if i % 2 == 0:  
        print(i)
```



Iterate from 0 up to (but not including) 100

```
for i in range(0, 100, 2):  
    print(i)
```



Iterate from 0 up to (but not including) 100 in steps of 2

```
airports = {  
    'AMS': 'Schiphol',  
    'LHR': 'Heathrow',  
    'OSL': 'Oslo',  
}
```

```
for code in airports:  
    print(code)
```

 Iterate over the keys

```
for code in airports.keys():  
    print(code)
```

 Iterate over the keys

```
for name in airports.values():  
    print(name)
```

 Iterate over the values

```
for code, name in airports.items():  
    print(code, name)
```

 Iterate over key-value pairs as tuples

## ■ Both while and for support optional else statements

- It is executed if when the loop completes, i.e., (mostly) equivalent to a statement following the loop

```
with open('log.txt') as log:  
    for line in log:  
        if line.startswith('DEBUG'):  
            print(line, end='')  
        else:  
            print('*** End of log ***')
```

## ■ Both while and for support...

- Early loop exit using break, which bypasses any loop else statement
- Early loop repeat using continue, which execute else if nothing further to loop

```
with open('log.txt') as log:  
    for line in log:  
        if line.startswith('FATAL'):  
            print(line, end='')  
            break
```

- Functional programming tools reduce many loops to simple expressions
- Iterators and generators support a lazy approach to handling series of values
- Iterators enjoy direct protocol and control structure support in Python
- Generators are functions that automatically create iterators
- Generator expressions support a simple way of generating generators
- Generators can abstract with logic

- Functions as principal units of composition
- A declarative rather than imperative style, emphasising data structure relations
- Functions executing without side-effects, i.e., functions are pure functions
- Immutability, so querying state and creating new state rather than updating it
- Lazy rather than eager access to values

```
def is_leap_year(year):
    return year % 4 == 0 and year % 100 != 0 or year % 400 == 0
years = range(1800, 2100)
```

```
leap_years = []
for year in years:
    if is_leap_year(year):
        leap_years.append(year)
```

Imperative list initialisation

```
leap_years = list(filter(is_leap_year, years))
```

Filtering

- Comprehensions are expressions that define value sequences declaratively
- Defined by intension as an expression, rather than procedurally
- Comprehension syntax is used to create lists, sets, dictionaries and generators
- Although the for and if keywords are used, they have different implications
- Can read for as from and if as where

```
squares = []
for i in range(13):
    squares.append(i**2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

Imperative list initialisation

```
squares = [i**2 for i in range(13)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

List comprehension

```
[i**2 for i in range(13)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

List comprehension

```
{abs(i) for i in range(-10, 10)}
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Set comprehension

```
{i: i**2 for i in range(8)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49}
```

Dictionary comprehension

```
list(range(0, 100, 2))
```

```
list(range(100))[::2]
```

```
list(range(100)[::2])
```

```
list(map(lambda i: i * 2, range(50)))
```

```
list(filter(lambda i: i % 2 == 0, range(100)))
```

```
[i * 2 for i in range(50)]
```

```
[i for i in range(100) if i % 2 == 0]
```

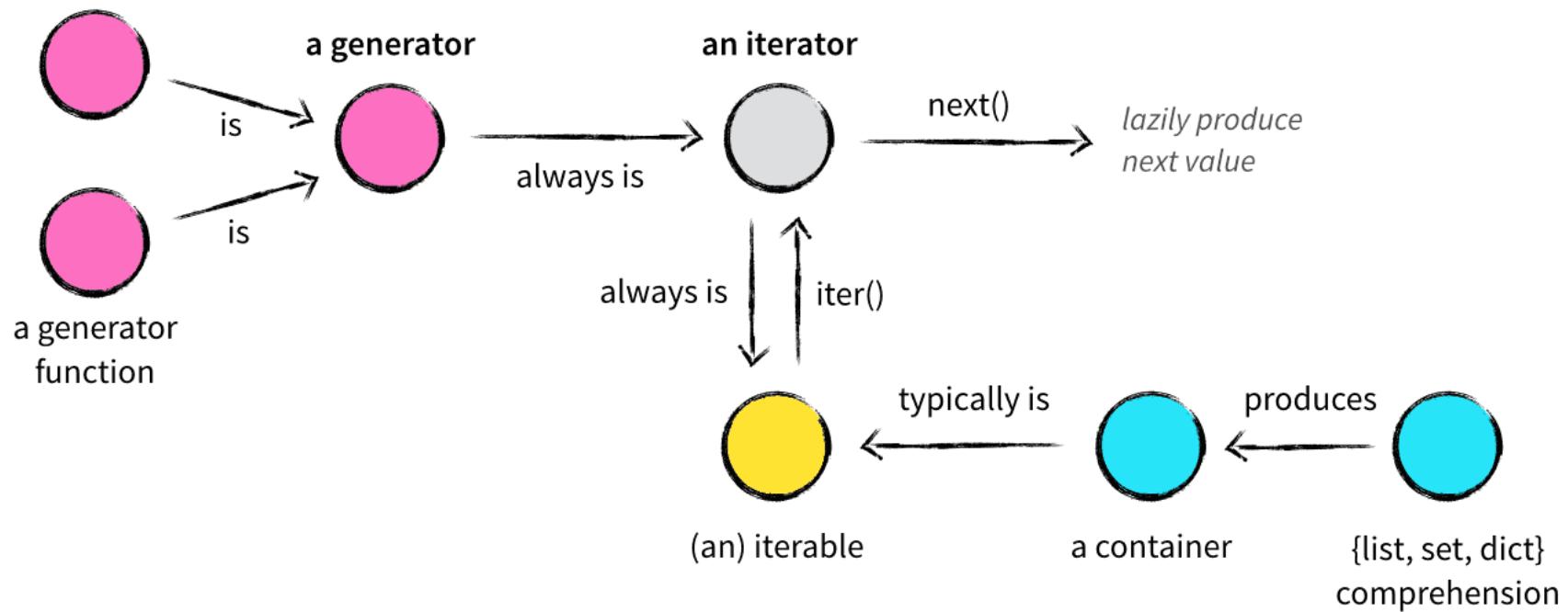
```
[i for i in range(100)][::2]
```

```
values = ['A'] + list(range(2, 11)) + ['J', 'Q', 'K']
suits = ['spades', 'hearts', 'diamonds', 'clubs']
[(value, suit) for suit in suits for value in values]
```



In what order do you expect the elements in the comprehension to appear?

a generator  
expression



- Be lazy by taking advantage of functionality already available
- E.g., the min, max, sum, all, any and sorted built-ins all apply to iterables
- Many common container-based looping patterns are captured in the form of container comprehensions
- Look at itertools and functools modules for more possibilities

- Be lazy by using abstractions that evaluate their values on demand
- You don't need to resolve everything into a list in order to use a series of values
- Iterators and generators let you create objects that yield values in sequence
- An iterator is an object that iterates
- For some cases you can adapt existing iteration functionality using the `iter` built-in

- An iterator is an object that iterates, following the iterator protocol
- An iterable object can be used with for
- Iterators and generators are lazy, returning values on demand
- You don't need to resolve everything into a list in order to use a series of values
- Yield values in sequence, so can linearise complex traversals, e.g., tree structures

- An iterator is an object that supports the `__next__` method for traversal
- Invoked via the `next` built-in function
- An iterable is an object that returns an iterator in support of `__iter__` method
- Invoked via the `iter` built-in function
- Iterators are iterable, so the `__iter__` method is an identity operation

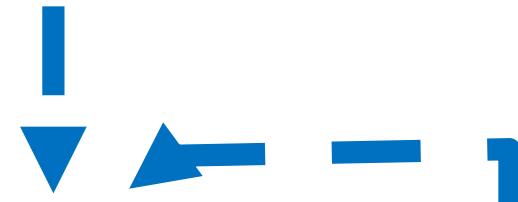
An object is iterable if it supports the `__iter__` special method, which is called by the `iter` function

```
class Iterable:
```

...

```
def __iter__(self):  
    return Iterator(...)
```

...



All iterators are iterable, so the `__iter__` method is an identity operation

```
class Iterator:
```

...

```
def __iter__(self):  
    return self
```



The `__next__` special method, called by `next`, advances the iterator

```
def __next__(self):
```

...

...

...

...

Iteration is terminated by raising `StopIteration`

```
StopIteration
```

- There are many ways to provide an iterator...
- Define a class that supports the iterator protocol directly
- Return an iterator from another object
- Compose an iterator with `iter`, using an action and a termination value
- Define a generator function
- Write a generator expression

- Use `iter` to create an iterator from a callable object and a sentinel value
- Or to create an iterator from an iterable

```
def pop_until(stack, end):
    return iter(stack.pop, end)

for popped in pop_until(history, None):
    print(popped)

def repl():
    for line in iter(lambda: input('> '), 'exit'):
        print(evaluate(line))
```

- Parallel assignment = tuple unpacking
- Parallel assignment in for loops
- Function argument unpacking = splat
- reduction functions = all, any, max, min, sum
- `.sort()` / `sorted()` 비교

- Iterators can be advanced manually using `next`
- Calls the `__next__` method
- Watch out for `StopIteration` at the end...

```
def repl():  
    try:  
        lines = iter(lambda: input('> '), 'exit')  
        while True:  
            line = next(lines)  
            print(evaluate(line))  
    except StopIteration:  
        pass
```

- A comprehension-based expression that results in an iterator object
- Does not result in a container of values
- Must be surrounded by parentheses unless it is the sole argument of a function
- May be returned as the result of a function

```
numbers = (random() for _ in range(42))  
sum(numbers)
```

```
sum(random() for _ in range(42))
```

- A generator is a comprehension that results in an iterator object
- It does not result in a container of values
- Must be surrounded by parentheses unless it is the sole argument of a function

```
(i * 2 for i in range(50))  
(i for i in range(100) if i % 2 == 0)
```

```
sum(i * i for i in range(10))
```

- You can write your own iterator classes or, in many cases, just use a function
- On calling, a generator function returns an iterator and behaves like a coroutine

```
def evens_up_to(limit):  
    for i in range(0, limit, 2):  
        yield i  
  
for i in evens_up_to(100):  
    print(i)
```

- A generator is an ordinary function that returns an iterator as its result
- The presence of a `yield` or `yield from` makes a function a generator, and can only be used within a function
- `yield` returns a single value
- `yield from` takes values from another iterator, advancing by one on each call
- `return` in a generator raises `StopIteration`, passing it any return value specified

- **enumerate**
- **filter**
- **map**
- **reversed**
- **zip**

## ■ enumerate to generate indexed pairs from any iterable

```
codes = ['AMS', 'LHR', 'OSL']
for index, code in enumerate(codes, 1):
    print(index, code)
```

1 AMS  
2 LHR  
3 OSL

- Elements from multiple iterables can be zipped into a single sequence
- Resulting iterator tuple-ises corresponding values together

```
codes = ['AMS', 'LHR', 'OSL']
names = ['Schiphol', 'Heathrow', 'Oslo']

for airport in zip(codes, names):
    print(airport)

airports = dict(zip(codes, names))
```

- map applies a function to iterable elements to produce a new iterable
- The given callable object needs to take as many arguments as there are iterables
- The mapping is carried out on demand and not at the point map is called

```
def histogram(data):
    return map(lambda size: size * '#', map(len, data))

text = "I'm sorry Dave, I'm afraid I can't do that."
print('\n'.join(histogram(text.split())))
```

- filter includes only values that satisfy a given predicate in its generated result
- If no predicate is provided — i.e., None —the Boolean of each value is assumed

```
numbers = [42, 0, -273.15, 0.0, 97, 23, -1]
```

```
positive = filter(lambda value: value > 0, numbers)
```

```
non_zero = filter(None, numbers)
```

```
list(positive)
```

[42, 97, 23]

```
list(non_zero)
```

[42, -273.15, 97, 23, -1]

- Prefer use of comprehensions over use of map and filter
- But note that a list comprehension is fully rather than lazily evaluated

```
numbers = [42, 0, -273.15, 0.0, 97, 23, -1]
positive = [value for value in numbers if value > 0]
non_zero = [value for value in numbers if value]
```

positive

[42, 97, 23]

non\_zero

[42, -273.15, 97, 23, -1]

```
for medal in medals():
    print(medal)
```

```
def medals():
    yield 'Gold'
    yield 'Silver'
    yield 'Bronze'
```

```
def medals():
    for result in 'Gold', 'Silver', 'Bronze':
        yield result
```

```
def medals():
    yield from ['Gold', 'Silver', 'Bronze']
```

## ■ infinite generators

- count(), cycle(), repeat()

## ■ generators that consume multiple iterables

- chain(), tee(), izip(), imap(), product(), compress()...

## ■ generators that filter or bundle items

- compress(), dropwhile(), groupby(), ifilter(), islice()

## ■ generators that rearrange items

- product(), permutations(), combinations()

```
currencies = {  
    'EUR': 'Euro',  
    'GBP': 'British pound',  
    'NOK': 'Norwegian krone',  
}
```

```
for code in currencies:  
    print(code, currencies[code])
```

```
ordinals = ['first', 'second', 'third']
```

```
for index in range(0, len(ordinals)):  
    print(ordinals[index])
```

```
for index in range(0, len(ordinals)):  
    print(index + 1, ordinals[index])
```

```
currencies = {  
    'EUR': 'Euro',  
    'GBP': 'British pound',  
    'NOK': 'Norwegian krone',  
}
```

```
for code, name in currencies.items():  
    print(code, name)
```

```
ordinals = ['first', 'second', 'third']
```

```
for ordinal in ordinals:  
    print(ordinal)
```

```
for index, ordinal in enumerate(ordinals, 1):  
    print(index, ordinal)
```

# Modular Programming

■ refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. Individual modules can then be cobbled together like building blocks to create a larger application.

## ■ Modular programming in Python

- A Python file corresponds to a module
  - A program is composed of one or more modules
  - A module defines a namespace, e.g., for function and class names
- A module's name defaults to the file name without the .py suffix
  - Module names should be short and in lower case

## ■ Simplicity

- Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.

## ■ Maintainability

- Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. (You may even be able to make changes to a module without having any knowledge of the application outside that module.) This makes it more viable for a team of many programmers to work collaboratively on a large application.

## ■ Reusability

- Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to recreate duplicate code.

## ■ Scoping

- Modules typically define a separate namespace, which helps avoid collisions between identifiers in different areas of a program. (One of the tenets in the Zen of Python is Namespaces are one honking great idea—let's do more of those!)

### ■ **extensive library of standard modules**

- Much of going beyond the core language is learning to navigate the library

## ■ Names in another module are not accessible unless imported

- A module is executed on first import
- Names beginning \_ considered private

Import

```
import sys  
import sys as system  
from sys import stdin  
from sys import stdin, stdout  
from sys import stdin as source  
from sys import *
```



Discouraged

Imported

```
sys.stdin, ...  
system.stdin, ...  
stdin  
stdin, stdout  
source  
stdin, ...
```

- A module's name is held in a module-level variable called `_name_`
- A module's contents can be listed using the built-in `dir` function

- `dir` can also be used to list attributes on any object, not just modules

```
import sys  
print(sys.__name__, 'contains', dir(sys))  
print(__name__, 'contains', dir())
```

### ■ A module's name is set to '\_\_main\_\_' when it is the root of control

- I.e., when run as a script
- A module's name is unchanged on import

...

```
def main():
```

...

```
if __name__ == '__main__':  
    main()
```



A common idiom is to define a main function to be called when a module is used as '\_\_main\_\_'

# Modules & Packages

The mechanics of organising source code

- A package hierarchically organises modules and other packages
- Where a module corresponds to a file, a regular package corresponds to a directory
- Modules and packages define global namespaces
- Extension modules allow extension of Python itself

**■ A module in Python corresponds to a file with a .py extension**

- import is used to access features in another module

**■ A module's `_name_` corresponds to its file name without the extension**

- When run as a script (e.g., using the -m option), the root module has '`__main__`' as its `_name_`

### ■ The Python interpreter can be extended using extension modules

- Python has a C API that allows programmatic access in C (or C++) and, therefore, any language callable from C

### ■ Python can also be embedded in an application as a scripting language

- E.g., allow an application's features to be scripted in Python

## ■ Packages are a way of structuring the module namespace

- A submodule bar within a package foo represents the module foo.bar

## ■ A regular package corresponds to a directory of modules (and packages)

- A regular package directory must have an `__init__.py` file (even if it's empty)
- The package name is the directory name

**■ A namespace is a mapping from names to objects, e.g., variables**

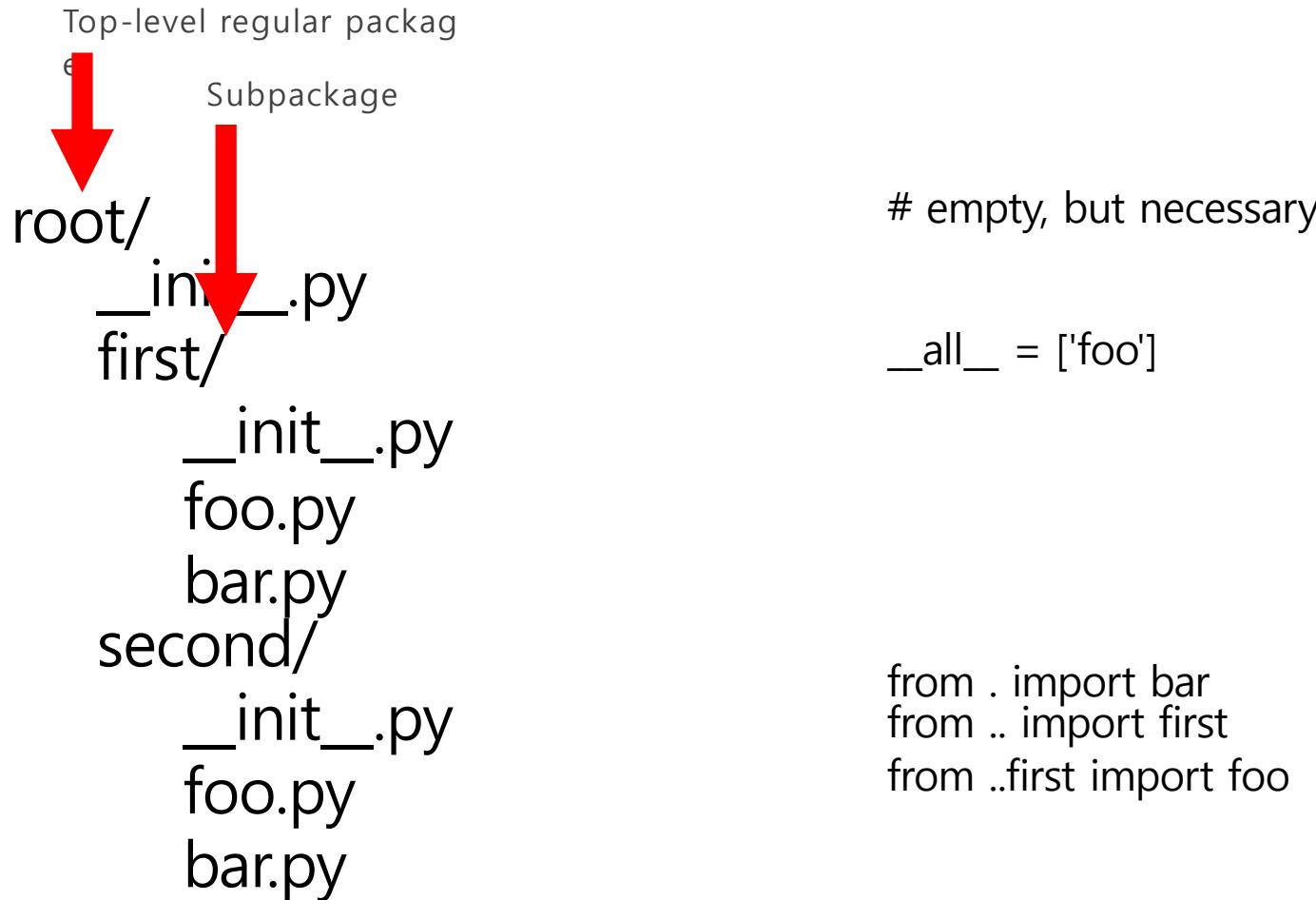
- A namespace is implemented as a dict
- Global, local, built-in and nested

**■ Each module gets its own global namespace, as does each package**

- import pulls names into a namespace
- Within a scope, names in a namespace can be accessed without qualification

### ■ A submodule can be imported with respect to its package

- E.g., `import package.submodule`
- Relative naming can be used to navigate within a package
- Main modules must use absolute imports
- Submodules seen by wildcard import can be specified by assigning a list of module names (as strings) to `__all__` in `__init__.py`



# **Classes & Objects**

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer," code is also represented by objects.)

Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory. The `is` operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

- Object types are defined by classes
- Attributes are not strictly private
- Methods can be bound to instance or class or be static
- Classes derive from one or more other classes, with object as default base
- Classes and methods may be abstract by convention or by decoration
- object oriented
  - Polymorphism is based on duck typing
  - Encapsulation style is façade-like rather than strict

## ■ A class's suite is executed on definition

- Instance methods in a class must allow for a self parameter, otherwise they cannot be called on object instances
- Attributes are held internally within a dict



By default all classes in



inherit from object

```
class Point:  
    def __init__(self, x=0, y=0):  
        self.x, self.y = x, y  
    def is_at_origin(self):  
        return self.x == self.y == 0
```

- The special `__init__` method is used to initialise an object instance
- It is called automatically, if present
- A class name is callable, and its call signature is based on `__init__`
- Introduce instance attributes by assigning to them in `__init__`
- Can be problematic in practice to introduce instance attributes elsewhere

## ■ Attribute and method names that begin `_` are considered private

- They are mangled with the class name, e.g., `_x` in `Point` becomes `_Point__x`
- Other names publicly available as written

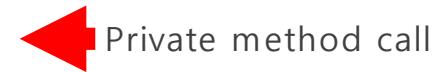
`_x` and `_y` are mangled to `_Point__x` and `_Point__y`

Within `Point`'s methods  
`_x` and `_y` are available  
by their unmangled nam  
es

```
class Point:  
    def __init__(self, x=0, y=0):  
        self._x, self._y = x, y  
    def is_at_origin(self):  
        return self._x == self._y == 0
```

- Attributes and methods of an object can be accessed using dot notation
- Access from outside is subject to name mangling if the features are private
- Access within a method is unmangled

```
class Spam:  
    def __bacon(self):  
        return Spam()  
    def egg(self):  
        return self.__bacon()
```



- A class attribute is defined as a variable within the class statement
- Can be accessed as a variable of the class using dot notation
- Can be accessed by methods via self
- Single occurrence shared by all instances

```
class Menu:  
    options = {'spam', 'egg', 'bacon', 'sausage'}  
    def order(self, *choice):  
        return set(choice) <= self.options
```

- Class methods are bound to the class, not just its instances
- They are decorated with `@classmethod`
- They take a class object, conventionally named `cls`, as their first argument
- They can be overridden in derived classes
- They are also accessible, via `self`, within instance methods

```
class Menu:  
    @classmethod  
    def options(cls):  
        return {'spam', 'egg', 'bacon', 'sausage'}  
  
    def order(self, *choice):  
        return set(choice) <= self.options()  
  
class NonSpammyMenu(Menu):  
    @classmethod  
    def options(cls):  
        return {'beans', 'egg', 'bacon', 'sausage'}
```

- Static methods are scoped within a class but have no context argument
- I.e., neither a `cls` nor a `self` argument
- They are effectively global functions in a class scope

```
class Menu:  
    @staticmethod  
    def options():  
        return {'spam', 'egg', 'bacon', 'sausage'}  
    def order(self, *choice):  
        return set(choice) <= self.options()
```

- Instance and class methods are functions bound to their first argument
- A bound function is an object
- Method definitions are ordinary function definitions
- Can be used as functions, accessed with dot notation from the class

```
class Spam:  
    def egg(self):  
        pass  
spam = Spam()  
spam.egg()  
Spam.egg(spam)
```

- A class can be derived from one or more other classes
- By default, all classes inherit from object
- Method names are searched depth first from left to right in the base class list

```
class Base:  
    pass  
class Derived(Base):  
    pass
```

```
class Base(object):  
    pass  
class Derived(Base):  
    pass
```



- Methods from the base classes are inherited into the derived class
- No difference in how they are called
- They can be overridden simply by defining a new method of the same name
- Inheritance is queryable feature of objects

`isinstance(value, Class)`

`issubclass(Class1, Class2)`

- Abstract classes are more a matter of convention than of language
- Python lacks the direct support found in statically typed languages
- There are two approaches...
- In place of declaring a method abstract, define it to raise `NotImplementedError`
- Use the `@abstractmethod` decorator from the `abc` module

Instances of Command can be instantiated, but calls to execute will fail

```
class Command:  
    def execute(self):  
        raise NotImplementedError
```

Instances of Command cannot be instantiated — but ABCMeta must be the metaclass for @abstractmethod to be used

```
from abc import ABCMeta  
from abc import abstractmethod
```

```
class Command(metaclass=ABCMeta):  
    @abstractmethod  
    def execute(self):  
        pass
```

# Special Methods & Attributes

Integrating new types with operators & built-in functions

- Special methods take the place of operators or overloading in Python
- Built-in functions also build on special methods and special attributes
- The set of special names Python is aware of is predefined
- Context managers allow new classes to take advantage of with statements

- It is not possible to overload operators in Python directly
- There is no syntax for this
- But classes can support operators and global built-ins using special methods
- They have reserved names, meanings and predefined relationships, and some cannot be redefined, e.g., is and id
- Often referred to as magic methods

Equality operations

**`_eq_(self, other)`**

`self == other`

**`_ne_(self, other)`**

`self != other`

If not defined for a class, `==` supports reference equality



Ordering operations

**`_lt_(self, other)`**

`self < other`

**`_gt_(self, other)`**

`self > other`

**`_le_(self, other)`**

`self <= other`

**`_ge_(self, other)`**

`self >= other`

If not defined for a class, `!=` is defined as `not self == other`



Note that the `__cmp__` special method is not supported in Python 3, so consider using the `functools.total_ordering` decorator to help define the rich comparisons



These methods return `NotImplemented` if not provided for a class, but whether `TypeError` is raised when an operator form is used depends on whether a reversed comparison is also defined, e.g., if `__lt__` is defined but `__gt__` is not, `a > b` will be executed as `b < a`

Unary operators and built-in functions

|                             |                        |
|-----------------------------|------------------------|
| <code>_neg_(self)</code>    | <code>-self</code>     |
| <code>_pos_(self)</code>    | <code>+self</code>     |
| <code>_invert_(self)</code> | <code>~self</code>     |
| <code>_abs_(self)</code>    | <code>abs(self)</code> |

Binary arithmetic operators

|                                      |                            |
|--------------------------------------|----------------------------|
| <code>_add_(self, other)</code>      | <code>self + other</code>  |
| <code>_sub_(self, other)</code>      | <code>self - other</code>  |
| <code>_mul_(self, other)</code>      | <code>self * other</code>  |
| <code>_truediv_(self, other)</code>  | <code>self / other</code>  |
| <code>_floordiv_(self, other)</code> | <code>self // other</code> |
| <code>_mod_(self, other)</code>      | <code>self % other</code>  |
| <code>_pow_(self, other)</code>      | <code>self**other</code>   |

Reflected forms also exist, e.g., `__radd__`, where method dispatch should be on the right-hand rather than left-hand operand



...

Note that if these special methods are not defined, but the corresponding binary operations are, these assignment forms are still valid, e.g., if `__add__` is defined, then `lhs += rhs` becomes `lhs = lhs.__add__(rhs)`

Augmented assignment operators



|                                         |                             |
|-----------------------------------------|-----------------------------|
| <code>__iadd__(self, other)</code>      | <code>self += other</code>  |
| <code>__isub__(self, other)</code>      | <code>self -= other</code>  |
| <code>__imul__(self, other)</code>      | <code>self *= other</code>  |
| <code>__itruediv__(self, other)</code>  | <code>self /= other</code>  |
| <code>__ifloordiv__(self, other)</code> | <code>self // other</code>  |
| <code>__imod__(self, other)</code>      | <code>self %= other</code>  |
| <code>__ipow__(self, other)</code>      | <code>self **= other</code> |

...



Each method should return the result of the in-place operation (usually `self`) as the result is used for the actual assignment, i.e., `lhs += rhs` is equivalent to `lhs = lhs.__iadd__(rhs)`

## Numeric type conversions

`_int_(self)``_float_(self)``_round_(self)``_complex_(self)``_bool_(self)``_hash_(self)`

And also `_round_(self, n)` to round to n digits after the decimal point (or before if negative)

Should only define if `__eq__` is also defined, such that objects comparing equal have the same hash code

## String type conversions

If `__str__` is not defined, `__repr__` is used by `str` if present

Official string representation, that should ideally be a valid Python expression that constructs the value

`_str_(self)``_format_(self, format)``_repr_(self)``_bytes_(self)`

## Content query

**\_\_len\_\_(self)**`len(self)`**\_\_contains\_\_(self, item)** item in self

If `__contains__` not defined,  
in uses linear search based  
on `__iter__` or indexing fro  
m 0 of `__getitem__`



## Iteration

Most commonly invoke  
d indirectly in the cont  
ext of a for loop

▶ **\_\_iter\_\_(self)** `iter(self)`  
**\_\_next\_\_(self)** `next(self)`

## Subscripting

**\_\_getitem\_\_(self, key)** `self[key]`**\_\_setitem\_\_(self, key, item)** `self[key] = item`**\_\_delitem\_\_(self, key)** `del self[key]`**\_\_missing\_\_(self, key)**

Hook method used by dict for its subcl  
ases

Support for function calls and attribute access

- `_call_(self, ...)`
- `_getattr_(self, name)`
- `_getattribute_(self, name)`
- `_setattr_(self, name, value)`
- `_delattr_(self, name)`
- `_dir_(self)`

Object lifecycle

- `_init_(self, ...)`
- `_del_(self)`

Context management

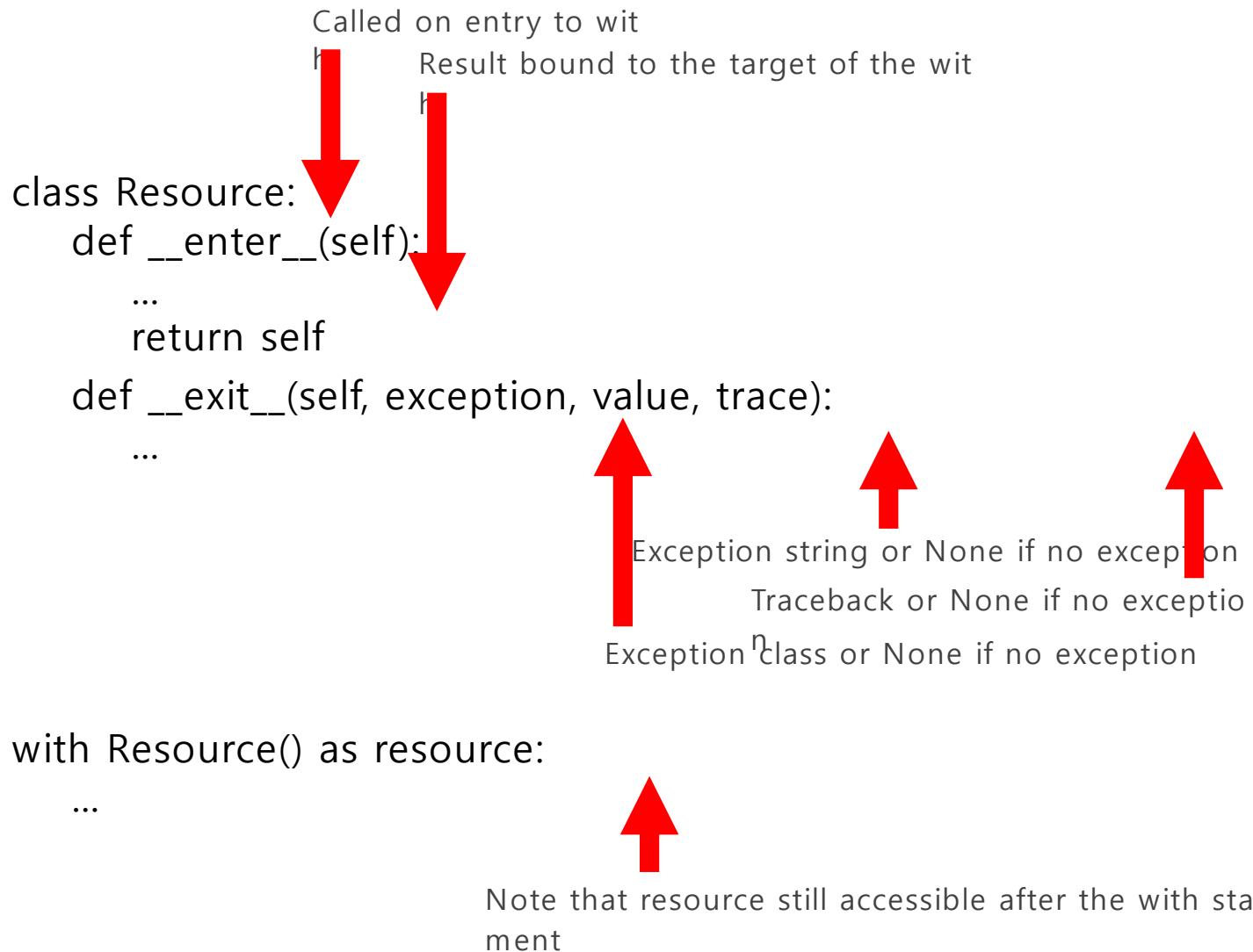
- `_enter_(self)`
- `_exit_(self, exception, value, traceback)`

■ **The with statement provides exception safety for paired actions**

- I.e., where an initial acquire/open/lock action must be paired with a final release/close/unlock action, regardless of exceptions

■ **A context manager is an object that can be used in this way in a with**

■ **It is defined by a simple protocol**



Write a context manager to time the execution of code within a with statement



with Timing():  
    task\_to\_time()

Code to time the execution of



```
from time import time  
  
class Timing:  
    def __enter__(self):  
        def __exit__(self, exception, value, trace):
```

Record current time using time



Print elapsed time, optionally indicating whether an exception was thrown



### ■ A number of special attributes can also be defined, including...

- `__doc__` is the documentation string and is used by the built-in help function
- `__name__` is used to hold the name for functions and classes
- `__dict__` refers to the dictionary of values held by any object
- `__module__` names the defining module



- Classes are mutable after definition, so methods can be added and removed
- Objects can have attributes added and removed after creation
- Decorators wrap functions on definition
- A metaclass is the class of a class, defining how the class behaves

- Everything in Python is expressed as objects, from functions to modules
- Functions are callable objects — with associated state and metadata — and are instantiated with a def statement
- And Python is a very dynamic language, not just dynamically typed
- Python's object model can be fully accessed at runtime

- Objects are normally created based on calling their class name
- But the class name does not have to be present, only the class object
- E.g., `type(parrot)()` creates a new object with the same type as parrot
- This eliminates the need for some of the factory patterns and convoluted tricks employed in other languages

- Keyword arguments make dicts easy to use as ad hoc data structures
- Keyword arguments are passed as dicts, with keywords becoming keys
- See also `collections.namedtuple`

```
sputnik_1 = dict(year=1957, month=10, day=4)
                  {'day': 4, 'month': 10, 'year': 1957}

origin = dict(x=0, y=0)
                  {'y': 0, 'x': 0}
```

- It is possible to add attributes to an object after it has been created
- Object attributes are dynamic and implemented as a dictionary
- An attribute can be removed using `del`

Ad hoc data structures with named attributes can be created easily

```
class Spam:  
    pass  
  
meal = Spam()  
meal.egg = True  
meal.bacon = False
```

A **monkey patch** is a way for a program to extend or modify supporting system software locally (affecting only the running instance of the program). This process has also been termed **duck punching**.

The definition of the term varies depending upon the community using it. In Ruby, Python, and many other dynamic programming languages, the term monkey patch only refers to dynamic modifications of a class or module at runtime.

- Methods and attributes can be attached to existing classes
- But be careful doing so!

```
class Menu:  
    pass  
  
menu = Menu()  
  
Menu.options = {'spam', 'egg', 'bacon', 'sausage'}  
def order(self, *choice):  
    return set(choice) <= self.options  
Menu.order = order  
  
assert menu.order('egg')  
assert not menu.order('toast')
```

- A number of functions help to explore the runtime object model
- `dir`: a sorted list of strings of names in the current or specified scope
- `locals`: dict of current scope's variables
- `globals`: dict of current scope's global variables
- `vars`: dict of an object's state, i.e., `_dict_`, or `locals` if no object specified

- Object attributes can be accessed using `hasattr`, `getattr` and `setattr`
- These global functions are preferred to manipulating an object's `_dict_` directly
- An object can provide special methods to intercept attribute access
- Query is via `__getattr__` (if not present) and `__getattribute__` (unconditional)
- Modify via `__setattr__` and `__delattr__`

- A function definition may be wrapped in decorator expressions
- A decorator is a callable object that results in a callable object given a callable object or specified arguments
- On definition the function is passed to the decorator and the decorator's result is used in place of the function
- This forms a chain of wrapping if more than one decorator is specified

```
def non_zero_args(wrapped):
    def wrapper(*args):
        if all(args):
            return wrapped(*args)
        else:
            raise ValueError
    return wrapper
```

A nested function that performs the validation, executing the wrapped function with arguments if successful  
(Note: for brevity, keyword arguments are not handled)

```
@non_zero_args
def date(year, month, day):
    return year, month, day
```

```
date(1957, 10, 4)
date(2000, 0, 0)
```

Decorator

Returns (1957, 10, 4)

Raises ValueError

```
from functools import wraps
def non_zero_args(wrapped):
    @wraps(wrapped)
    def wrapper(*args):
        if all(args):
            return wrapped(*args)
        else:
            raise ValueError
    return wrapper
```

```
@non_zero_args
def date(year, month, day):
    return year, month, day
```



Ensures the resulting wrapper has the same principal attributes as the wrapped function, e.g., `__name__` and `__doc__`

```
def check_args(checker):
    def decorator(wrapped):
        def wrapper(*args):
            if checker(args):
                return wrapped(*args)
            else:
                raise ValueError
        return wrapper
    return decorator
```

```
@check_args(all)
def date(year, month, day):
    return year, month, day
```

```
date(1957, 10, 4)
date(2000, 0, 0)
```

Nested function that curries the checker argument  
Nested nested function that performs the validation

Returns (1957, 10, 4)  
Raises ValueError

# Metaclasses

- A metaclass can be considered the class of class
  - A class defines how objects behaves; a meta class defines how classes behaves
- Metaclasses are most commonly used as class factories
  - Customise class creation and execution
  - A class is created from a triple of name, base classes and a dictionary of attributes

# abc.ABCMeta

Use of the ABCMeta metaclass to define abstract classes is one of the more common metaclass examples

Instantiation for Visitor is disallowed because of ABCMeta and the presence of at least one @abstractmethod

@abstractmethod only has effect if ABCMeta is the metaclass

```
from abc import ABCMeta  
from abc import abstractmethod
```

```
class Visitor(metaclass=ABCMeta):  
    @abstractmethod  
    def enter(self, visited):  
        pass  
    @abstractmethod  
    def exit(self, visited):  
        pass  
    @abstractmethod  
    def visit(self, value):  
        pass
```

# type

- type is the default metaclass in Python
  - And type is its own class, i.e., type(type) is a n identity operation
  - type is most often used as a query, but it ca n also be used to create a new type

```
def init(self, options):  
    self.__options = options  
def options(self):  
    return self.__options  
Menu = type('Menu', (),  
            {'__init__': init, 'options': options})  
menu = Menu({'spam', 'egg', 'bacon', 'sausage'})
```

The base def  
aults to obje  
ct

# Adding new code at runtime

- New code, from source, can be added to the Python runtime
  - The `compile` function returns a code object from a module, statement or expression
  - The `exec` function executes a code object
  - The `eval` function evaluates source code and returns the resulting value
- Here be dragons!

# Object Thinking

- ✓ Protocols, polymorphism, patterns & practical advice

# Facts at a glance

- Object usage defined more by protocols than by class relationships
- Substitutability is strong conformance
- Polymorphism is a fundamental consideration (but inheritance is not)
- Values can be expressed as objects following particular conventions
- Enumeration types have library support

# Not all objects are equal

- Be careful how you generalise your experience!
  - Patterns of practice are context sensitive
- Python's type system means that...
  - Some OO practices from other languages and design approaches travel well
  - Some do not translate easily or well — or even at all
  - And some practices are specific to Python

# Protocols and conformance

- An expected set of method calls can be considered to form a protocol
  - Protocols may be named informally, but they are not part of the language
- Conformance to a protocol does not depend on inheritance
  - Dynamic binding means polymorphism and subclassing are orthogonal
  - Think duck types not declared types

# Liskov Substitution Principle

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of an other type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object  $o_1$  of type  $S$  there is an object  $o_2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$ , then  $S$  is a subtype of  $T$ .

Barbara Liskov  
"Data Abstraction and Hierarchy"

# Contracts & consequences

- Substitutability is the strongest form of conformance to a protocol
  - A contract of behavioural equivalence
- Following the contract means...
  - An overridden method cannot result in a wider range or cover a narrower domain
  - Concrete classes should not be bases
  - A derived class should support the same initialisation protocol as its base

# Inheritance & composition

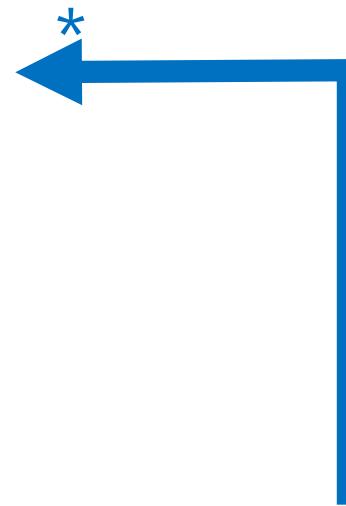
- A class can respect logical invariants
  - Assertions true of its methods and state
- A derived class should respect the invariants of its base classes
  - And may strengthen them
- Consider composition and forwarding
  - Especially if derivation would break invariants and lead to method spam from the base classes

# Object structure & traversal

- The following patterns combine and complement one another:
  - Composite: recursive—whole part structure
  - Visitor: dispatch based on argument type
  - Enumeration Method: iteration based on inversion of control flow
  - Lifecycle Callback: define actions for object lifecycle events as callbacks

# Composite pattern

```
class Node(metaclass=ABCMeta):
    @abstractmethod
    def children(self):
        pass
    def name(self):
        return self._name
    ...
```



```
class Primitive(Node):
    def __init__(self, name):
        self._name = name
    def children(self):
        return ()
    ...
```

```
class Group(Node):
    def __init__(self, name, children):
        self._name = name
        self._children = tuple(children)
    def children(self):
        return self._children
    ...
```

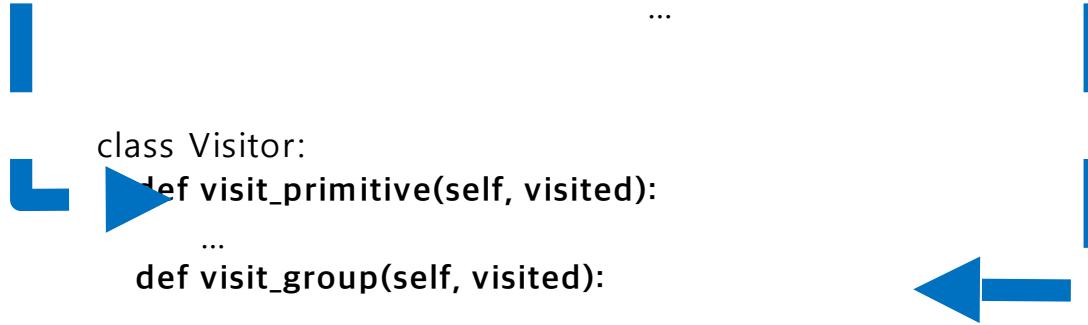
# Visitor pattern

```
class Node(metaclass=ABCMeta):
    ...
    @abstractmethod
    def accept(self, visitor):
        pass
    ...
```

```
class Primitive(Node):
    ...
    def accept(self, visitor):
        visitor.visit_primitive(self)
    ...
```

```
class Group(Node):
    ...
    def accept(self, visitor):
        visitor.visit_group(self)
```

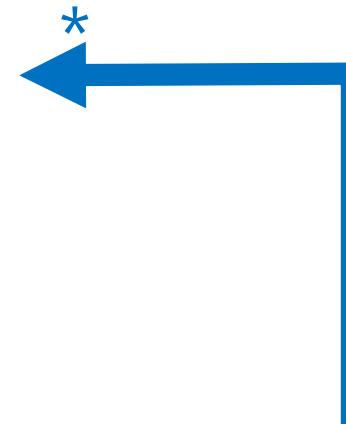
```
class Visitor:
    L  def visit_primitive(self, visited):
        ...
        def visit_group(self, visited):
            ...
```



```
class Node(metaclass=ABCMeta):
    @abstractmethod
    def map(self, callback):
        pass
    ...
```

```
class Primitive(Node):
    ...
    def map(self, callback):
        pass
    ...
```

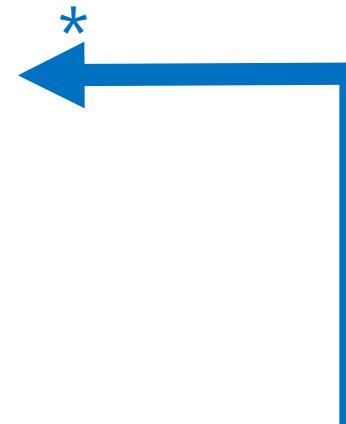
```
class Group(Node):
    def __init__(self, name, children):
        self._name = name
        self._children = tuple(children)
    def map(self, callback):
        for child in self._children:
            callback(child)
    ...
```



```
class Visitor:  
    def enter(self, group):  
        ...  
    def leave(self, group):  
        ...  
    def visit(self, primitive):  
        ...
```

```
class Printer:  
    def __init__(self):  
        self.__depth = 0  
  
    def enter(self, group):  
        print(self.__depth * ' ' + '<group>')  
        self.__depth += 1  
  
    def leave(self, group):  
        self.__depth -= 1  
        print(self.__depth * ' ' + '</group>')  
  
    def visit(self, primitive):  
        print(self.__depth * ' ' + '<primitive/>')
```

```
class Node(metaclass=ABCMeta):
    @abstractmethod
    def for_all(self, visitor):
        pass
    ...
```



```
class Primitive(Node):
    ...
    def for_all(self, visitor):
        visitor.visit(self)
    ...
```

```
class Group(Node):
    ...
    def for_all(self, visitor):
        visitor.enter(self)
        for child in self._children:
            child.for_all(visitor)
        visitor.leave(self)
    ...
```

- Many object forwarding patterns have a simple and general form in Python
- Proxy: offer transparent forwarding by one object to another, supporting the target object's protocol
- Null Object: represent absence of an object with an object that realises the object protocol with do-nothing or return-default implementations for its methods

```
class TimingProxy:  
    def __init__(self, target, report=print):  
        self.__target = target  
        self.__report = report  
    def __getattr__(self, name):  
        attr = getattr(self.__target, name)  
        def wrapper(*args, **kwargs):  
            start = time()  
            try:  
                return attr(*args, **kwargs)  
            finally:  
                end = time()  
                self.__report(name, end - start)  
        return wrapper if callable(attr) else attr
```

A generic Null Object implementation useful, for example, as for test dummy objects, particularly if no specific return values are expected or tested



```
class NullObject:  
    def __call__(self, *args, **kwargs):  
        return self  
    def __getattr__(self, name):  
        return self
```



Method calls on NullObject are chainable as NullObject is callable

- It is worth differentiating between objects that represent mechanisms...
- And may therefore have changing state
- And objects that represent values
- Their focus is information (e.g., quantities), rather than being strongly behavioural (e.g., tasks), or entity-like (e.g., users)
- They should be easily shared and consistent, hence immutable (i.e., like str)

- For a class of value objects...
- Provide for rich construction to ensure well-formed objects are easy to create
- Provide query but not modifier methods
- Consider @property for zero-parameter query methods so they look like attributes
- Define \_\_eq\_\_ and \_\_hash\_\_ methods
- Provide relational operators if values are ordered (see `functools.total_ordering`)

```
@total_ordering
class Money:
    def __init__(self, units, hundredths):
        self.__units = units
        self.__hundredths = hundredths

    @property
    def units(self):
        return self.__units

    @property
    def hundredths(self):
        return self.__hundredths

    def __eq__(self, other):
        return (self.units == other.units and
                self.hundredths == other.hundredths)

    def __lt__(self, other):
        return ((self.units, self.hundredths) <
                (other.units, other.hundredths))

    ...
```

```
@total_ordering
class Money:
    def __init__(self, units, hundredths):
        self.__total_hundredths = units * 100 + hundredths

    @property
    def units(self):
        return self.__total_hundredths // 100

    @property
    def hundredths(self):
        return self.__total_hundredths % 100

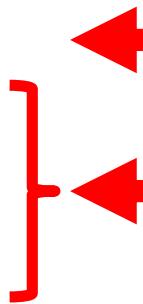
    def __eq__(self, other):
        return (self.units == other.units and
                self.hundredths == other.hundredths)

    def __lt__(self, other):
        return ((self.units, self.hundredths) <
                (other.units, other.hundredths))

    ...
```

- Python (as of 3.4) supports enum types
- They are similar — but also quite different — to enum types in other languages
- Support comes from the enum module
- An enumeration type must inherit from either Enum or IntEnum
- Enum is the base for pure enumerations
- IntEnum is the base class for integer-comparable enumerations

```
from enum import Enum  
class Suit(Enum):  
    spades = 1  
    hearts = 2  
    diamonds = 3  
    clubs = 4
```



Use IntEnum if easily comparable enumerations are needed

Enumeration names are accessible as strings are accessible via the name property and the integer via value

```
values = ['A'] + list(range(2, 11)) + ['J', 'Q', 'K']  
[(value, suit.name) for suit in Suit for value in values]
```



Enumeration types are iterable

- **Enum and IntEnum can be used to create enums on the fly**
- **They are both callable**

```
Colour = Enum('Colour', 'red green blue')
```

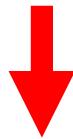
```
Colour = Enum('Colour', ('red', 'green', 'blue'))
```

```
Colour = Enum('Colour', {'red': 1, 'green': 2, 'blue': 3})
```

```
class Colour(Enum):  
    red = 1  
    green = 2  
    blue = 3
```

```
dict      dict()
{}
{'Bohr': True, 'Einstein': False}
{n: n**2 for n in range(0, 100)}
set      set()
{'1st', '2nd', '3rd'}
{n**2 for n in range(1, 100)}
frozenset frozenset()
frozenset({15, 30, 40})
frozenset(n**2 for n in range(0, 100))
```

List comprehension



Set comprehension

```
[n for n in range(2, 100)
 if n not in
    {m for l in range(2, 10)
     for m in range(l*2, 100, l)}]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

- Built-in container types meet many common and initial needs
- But sometimes a different data structure will cut down the amount of code written or the runtime resources used
- Built-in container types supplemented by the standard collections module
- Container types in collections are not built in, so they do not have display forms

- defaultdict offers on-demand creation for keys that are not already present
- Derives from dict
- Uses a given factory function, such as a class name, to create default values

```
def histogram(data):
    result = defaultdict(str)
    for item in data:
        result[word] += '#'
    return result
```



Inserts a value of str()  
for each key looked up that is not already present

- Counter tracks occurrences of a key
- It derives from dict but behaves a lot like a bag or a multiset
- Counts can be looked up for a key
- Each key 'occurrence' can be iterated

```
with open(filename) as input:  
    words = input.read().split()  
counted = Counter(votes)  
most_common = counted.most_common(1)  
most_to_least_common = counted.most_common()
```

- OrderedDict preserves the order in which keys were inserted
- Derives from dict (with some LSP issues...)
- One of the most useful applications is to create sorted dictionaries, i.e., initialise from result of sorted on a dict

```
def word_counts(words):
    counts = OrderedDict()
    for word in words:
        counts[word] = 1 + counts.setdefault(word, 0)
    return counts
```

- ChainMap provides a view over a number of mapping objects
- Lookups are in sequence along the chain
- Updates apply only to the first mapping —avoid side effects by supplying {}

```
defaults = {'drink': 'tea', 'snack': 'biscuit'}
preferences = {'drink': 'coffee'}
options = ChainMap(preferences, defaults)
```

```
for key, value in options.items():
    print(key, value, sep=': ')
```

drink: coffee  
snack: biscuit

- A deque is a double-ended queue
- Supports efficient append and pop and appendleft and popleft operations
- Can be bounded, with overflow causing a pop from the opposite end to the append

```
def tail(filename, lines=10):  
    with open(filename) as source:  
        return deque(source, lines)
```

- namedtuple allows the creation of a tuple type with named attributes
- Can still be indexed and iterated
- namedtuple is subclass of tuple

```
Date = namedtuple('Date', 'year month day')
```

```
Date = namedtuple('Date', ('year', 'month', 'day'))
```

```
sputnik_1 = Date(1957, 10, 4)
```

```
sputnik_1 = Date(year=1957, month=10, day=4)
```

```
year, month, day = sputnik_1
```

```
year = sputnik_1.year
```

```
month = sputnik_1[1]
```

- The `collections.abc` module provides abstract container classes
- Using `isinstance`, these can be used to check whether an object conforms to a protocol, e.g., `Container`, `Hashable`, `Iterable`, `MutableSequence`, `Set`
- These abstract classes can also be used as mixin base classes for new container classes

# Unit Testing

Programming unittest with GUTs

- Python has many testing frameworks available, including unittest
- unittest is derived from JUnit and has a rich vocabulary of assertions
- Good Unit Tests (GUTs) are structured around explanation and intention
- The outcome of a unit test depends solely on the code and the tests

- The unittest framework is a standard JUnit-inspired framework
- It is organised around test case classes, which contain test case methods
- Naming follows Java camelCase
- Runnable in other frameworks, e.g., pytest
- Python has no shortage of available testing frameworks!
- E.g., doctest, nose, pytest

Very many people say "TDD" when they really mean, "I have good unit tests" ("I have GUTs"?). Ron Jeffries tried for years to explain what this was, but we never got a catch-phrase for it, and now TDD is being watered down to mean GUTs.

Alistair Cockburn

<http://alistair.cockburn.us/The+modern+programming+professional+has+GUTs>

- Good unit tests (GUTs)...
- Are fully automated, i.e., write code to test code
- Offer good coverage of the code under test, including boundary cases and error-handling paths
- Are easy to read and to maintain
- Express the intent of the code under test — they do more than just check it

- Problematic test styles include...
- Monolithic tests: all test cases in a single function, e.g., test
- Ad hoc tests: test cases arbitrarily scattered across test functions, e.g., test1, test2, ...
- Procedural tests: test cases bundled into a test method that correspond to target method, e.g., test\_foo tests foo

Test case method names must start with test

Derivation from base class is necessary

```
from leap_year import is_leap_year  
import unittest
```

```
class LeapYearTests(unittest.TestCase):
```

```
    def test_that_years_not_divisible_by_4_are_not_leap_years(self):  
        self.assertFalse(is_leap_year(2015))
```

```
    def test_that_years_divisible_by_4_but_not_by_100_are_leap_years(self):  
        self.assertTrue(is_leap_year(2016))
```

```
    def test_that_years_divisible_by_100_but_not_by_400_are_not_leap_years(self):  
        self.assertFalse(is_leap_year(1900))
```

```
    def test_that_years_divisible_by_400_are_leap_years(self):  
        self.assertTrue(is_leap_year(2000))
```

```
if __name__ == '__main__':  
    unittest.main()
```

```
assertEqual(lhs, rhs)
assertNotEqual(lhs, rhs)
assertTrue(result)
assertFalse(result)
assertIs(lhs, rhs)
assertIsNot(lhs, rhs)
assertIsNone(result)
assert IsNotNone(result)
assertIn(lhs, rhs)
assertNotIn(lhs, rhs)
assertIsInstance(lhs, rhs)
assertNotIsInstance(lhs, rhs)
```

...

All assertions also take an optional msg argument

```
assertLess(lhs, rhs)
assertLessEqual(lhs, rhs)
assertGreater(lhs, rhs)
assertGreaterEqual(lhs, rhs)
assertRegex(result)
assertNotRegex(result)
assertRaises(exception)
assertRaises(exception, callable, *args, *kwargs)
fail()
assertAlmostEqual(lhs, rhs)
assertAlmostNotEqual(lhs, rhs)
```

...



By default, approximate equality is established to within 7 decimal places — this can be changed using the places keyword argument or by providing a delta keyword argument

- assertRaises can be used as an ordinary assertion call or with with
- If a callable argument is not provided, assertRaises returns a context manager
- As a context manager, assertRaises fails if associated with body does not raise

```
self.assertRaises(ValueError, lambda: is_leap_year(-1))
```

with self.assertRaises(ValueError) as context:  
 is\_leap\_year(-1)



Optional, but can be used to access raised exception details

- assertRaisesRegex also matches string representation of raised exception
- Equivalent to regex search
- Like assertRaises, assertRaisesRegex can be used as function or context manager

```
self.assertRaises(  
    ValueError, 'invalid', lambda: is_leap_year(-1))
```

```
with self.assertRaises(ValueError, 'invalid') :  
    is_leap_year(-1)
```

So who should you be writing the tests for? For the person trying to understand your code.

Good tests act as documentation for the code they are testing. They describe how the code works. For each usage scenario, the test(s):

- Describe the context, starting point, or preconditions that must be satisfied

- Illustrate how the software is invoked

- Describe the expected results or postconditions to be verified

Different usage scenarios will have slightly different versions of each of these.

Gerard Meszaros  
"Write Tests for People"

- Example-based tests ideally have a simple linear flow: arrange, act, assert
- Given: set up data
- When: perform the action to be tested
- Then: assert desired outcome
- Tests should be short and focused
- A single objective or outcome — but not necessarily a single assertion — that is reflected in the name

- Common test fixture code can be factored out...
- By defining `setUp` and `tearDown` methods that are called automatically before and after each test case execution
- By factoring out common initialisation code, housekeeping code or assertion support code into its own methods, local to the test class

Tests that are not written with their role as specifications in mind can be very confusing to read. The difficulty in understanding what they are testing can greatly reduce the velocity at which a codebase can be changed.

Nat Pryce & Steve Freeman  
"Are your tests really driving your development"

A test is not a unit test if:

- It talks to the database

- It communicates across the network

- It touches the file system

- It can't run at the same time as any of your other unit tests

- You have to do special things to your environment (such as editing config files) to run it.

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.

Michael Feathers

- External resources are a common source of non-unit dependencies
- Registries and environment variables
- Network connections and databases
- Files and directories
- Current date and time
- Hardware devices
- Externals can often be replaced by test doubles or pre-evaluated objects

- It's not just about mocks...
- A test stub is used to substitute input and can be used for fault injection
- A test spy offers input and captures output behaviour
- A mock object validates expectations
- A fake object offers a usable alternative to a real dependency
- A dummy object fulfils a dependency

- Mocking and related techniques are easier in Python than many languages
- Duck typing means dependencies can be substituted without changing class design
- Passing functions as objects allows for simple pluggability
- It is easy to write interception code for new classes and to add interception code on existing objects or use existing libraries, e.g., `unittest.mock`