

Python

프로그래밍 언어마다 특징도 다양하고 장단점도 다양하다.



[http://www.inven.co.kr/mobile/board/powerbbs.php?come_idx=2097&category=%EF%BF%BD%EA%B3%97%EF%BF%BD%EF%BF%BD%3E%C2%A0%C2%A0%EF%BF%BD%EF%BF%BD%EF%BF%BD%EF%BF%BD%EF%BF%BD%2Fa%3E%20%7C%3Ca%20href%3D&l=832573](http://www.inven.co.kr/mobile/board/powerbbs.php?come_idx=2097&category=%EF%BF%BD%EA%B3%97%EF%BF%BD%EF%BF%BD%3E%C2%A0%C2%A0%EF%BF%BD%EF%BF%BD%EF%BF%BD%EF%BF%BD%2Fa%3E%20%7C%3Ca%20href%3D&l=832573)
<http://kstatic.inven.co.kr/upload/2017/08/03/bbs/i14621578868.jpg>

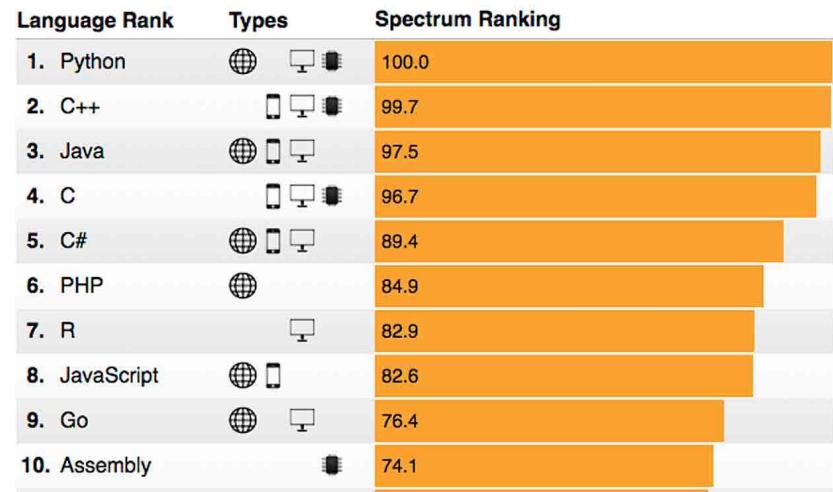
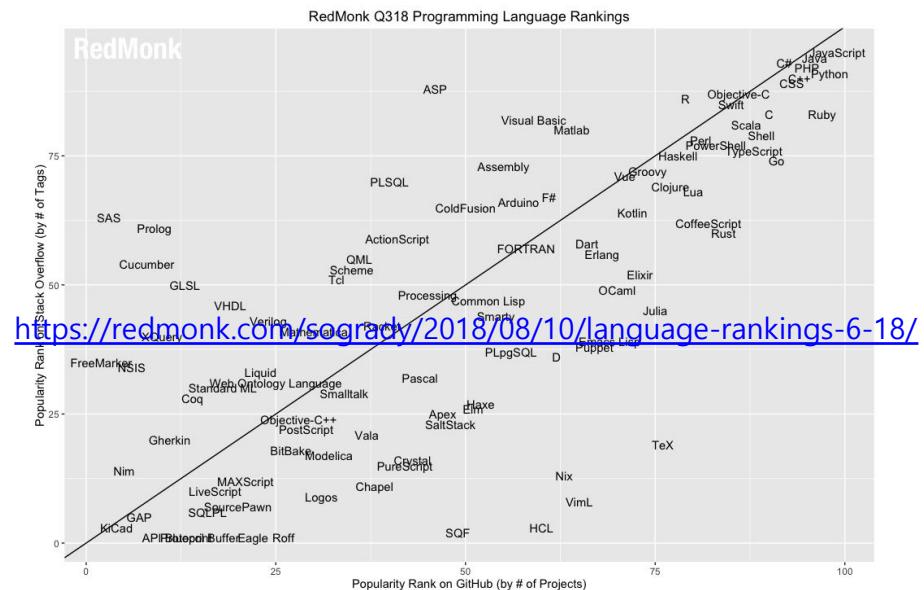
<http://redmist.tistory.com/21>
<http://redmist.tistory.com/30>

Worldwide, Sept 2018 compared to a year ago:				
Rank	Change	Language	Share	Trend
1	↑	Python	24.58 %	+5.7 %
2	↓	Java	22.14 %	-0.6 %
3	↑	Javascript	8.41 %	+0.0 %
4	↓	PHP	7.77 %	-1.4 %
5		C#	7.74 %	-0.4 %
6		C/C++	6.22 %	-0.8 %
7		R	4.04 %	-0.2 %
8		Objective-C	3.33 %	-0.9 %
9		Swift	2.65 %	-0.9 %
10		Matlab	2.1 %	-0.3 %

<http://pypl.github.io/PYPL.html>

Sep 2018	Sep 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.436%	+4.75%
2	2		C	15.447%	+8.06%
3	5	▲	Python	7.653%	+4.67%
4	3	▼	C++	7.394%	+1.83%
5	8	▲	Visual Basic .NET	5.308%	+3.33%
6	4	▼	C#	3.295%	-1.48%
7	6	▼	PHP	2.775%	+0.57%
8	7	▼	JavaScript	2.131%	+0.11%
9	-	▲	SQL	2.062%	+2.06%
10	18	▲	Objective-C	1.509%	+0.00%

<https://www.tiobe.com/tiobe-index/>



<https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages>



파이썬은 배우기 쉽고, 강력한 프로그래밍 언어입니다. 효율적인 자료 구조들과 객체 지향 프로그래밍에 대해 간단하고도 효과적인 접근법을 제공합니다. 우아한 문법과 동적 타이핑(typing)은, 인터프리터 적인 특징들과 더불어, 대부분 플랫폼과 다양한 문제 영역에서 스크립트 작성과 빠른 응용 프로그램 개발에 이상적인 환경을 제공합니다.

파이썬 인터프리터와 풍부한 표준 라이브러리는 소스나 바이너리 형태로 파이썬 웹 사이트, <https://www.python.org/>, 에서 무료로 제공되고, 자유롭게 배포할 수 있습니다. 같은 사이트는 제삼자들이 무료로 제공하는 확장 모듈, 프로그램, 도구, 문서들의 배포판이나 링크를 포함합니다.

파이썬 인터프리터는 C 나 C++ (또는 C에서 호출 가능한 다른 언어들)로 구현된 새 함수나 자료 구조를 쉽게 추가할 수 있습니다. 파이썬은 고객화 가능한 응용 프로그램을 위한 확장 언어로도 적합합니다.

<https://docs.python.org/ko/3/tutorial/index.html>

<https://docs.python.org/ko/3/tutorial/appetite.html>

Python

파이썬

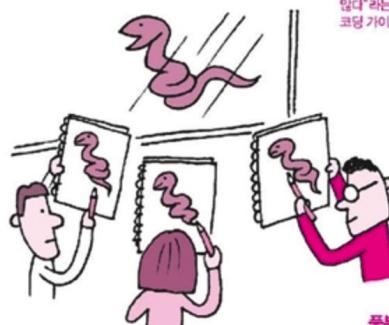
웹 앱에서 인공지능까지
인기가 급상승하며 주목받는 언어

탄생
1991년
만든 사람
Guido van Rossum
주요 용도
웹 애플리케이션, 인공지능
분류
설치형·할수형·格外지
행성/인터프리터

이란 언어

데이터 분석에 장점을 가진 스크립트 언어로 데이터 사이언티스
트러는 직종에 인기가 있다. 해외에서는 웹 애플리케이션의 개
발 언어로도 많이 사용되고 있으며, Python 프로그래머의 평균
연봉이 높은 것이 화제가 되기도 한다.

최근에는 기계학습 등 인공지능의 개발에도 많이 사용되고 있다.
버전 2x와 3x은 일부 호환성이 되지 않지만 두 버전 모두 이용자
가 많다.



다양한 구현이 있다

원래의 처리계인 'CPython'뿐만 아니라
'Jython'이나 'PyPy', 'Cython'이나 'IronPython'
등 다양한 구현이 있으며, 각각 특징이 있다.

인엔트가 중요

많은 언어가 '['과 ']' 등으로 복잡 구
조를 표현하는 반면, Python에서는
인엔트들이 쓰기로 표현한다.
"코드는 쓰는 것이다. 읽는 것이 더
많다"라는 의미에서 PEP8이라는
코딩 가이드도 제공되고 있다.



Column

The Zen of Python

Python 프로그래머가 가치야 할 마음가짐이 정리된 말.
Python으로 소스코드를 작성할 때 '단순'과 '가독성'을 실현하기
위한 19개의 문장으로 구성되어 있다.

기억해야 할 키워드



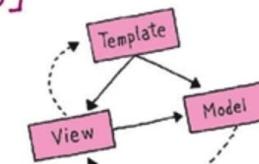
pip

Python 패키지 관리 시스템.
최신 Python이 기본으로 제공되어 검색
및 설치, 업데이트 등도 가능하다.
Git 저장소에서 직접 설치할 수도 있다.

[`x for x in range(10) if is_prime(x)`]

리스트 컴프리헨션

리스트를 생성할 때 사용되는 표기 방법.
수학에서 집합을 나타낼 때 사용되는 ($x \mid$
 $x < 10$ 이하의 소수)라는 표현에 기반.
for 루프보다 빠르게 처리할 수 있다.



Django

Python으로 사용되는 웹 애플리케이션
프레임워크 중에서도 가장 인기가 있다.
Instagram과 Pinterest의 개발에도 사용
되고 있는 것으로 알려져 있다.

프로그래밍 예제 하노이의 탑(hanoi.py)

```

# 하노이의 탑
def hanoi(n, from_, to, via):
    if n > 1:
        hanoi(n - 1, from_, via, to)
        print from_ + " -> " + to
        hanoi(n - 1, via, to, from_)
    else:
        print from_ + " -> " + to

# 표준 입력에서 단수를 받아서 실행
n = input()
hanoi(n, "a", "b", "c")
  
```

인엔트가 중요

탭 문자 또는 4 문자의 공백
이 사용되는 경우가 많다.
하나의 블록은 동일한 들여
쓰기로 동일이 필요하다.

Humorous



<https://www.youtube.com/watch?v=jiu0IYQIPqE>



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

Life is too short, You need

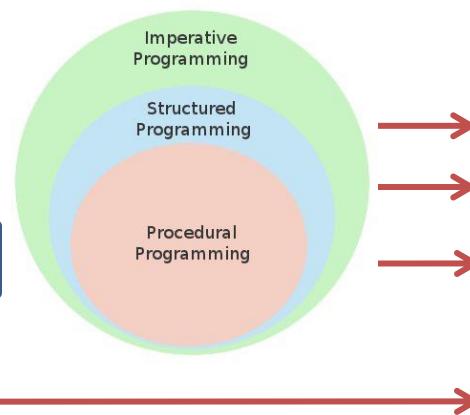
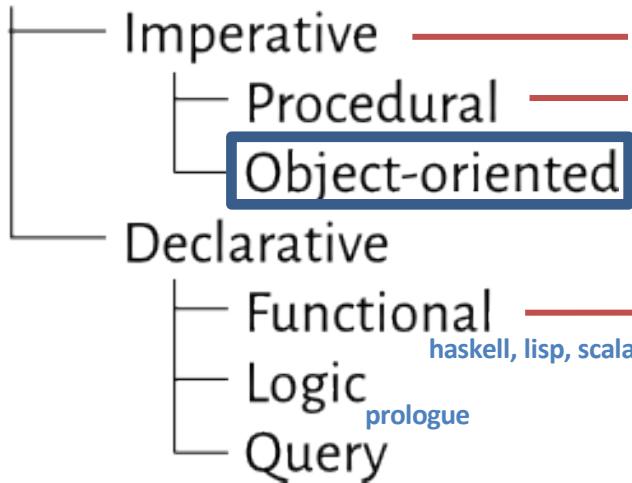


생산성

**Effective
Efficient**

Multi Paradigm

Programming languages



Everything is an object

The same problem can be solved and expressed in **different ways**
Paradigm can overlap or can be used w/ each other

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/omg-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

버전과 라이브러리에 대한 의존성 문제?

General Purpose



C.f)
Domain-specific



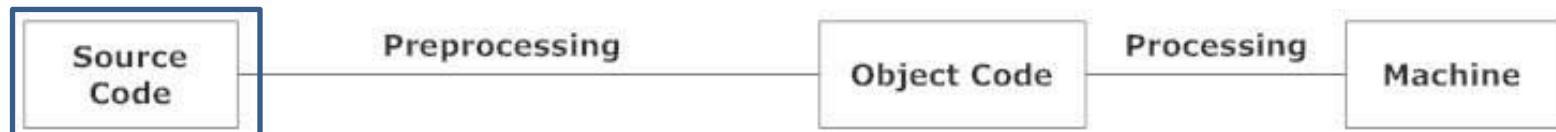
https://en.wikipedia.org/wiki/List_of_Python_software
<https://github.com/vinta/awesome-python>

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

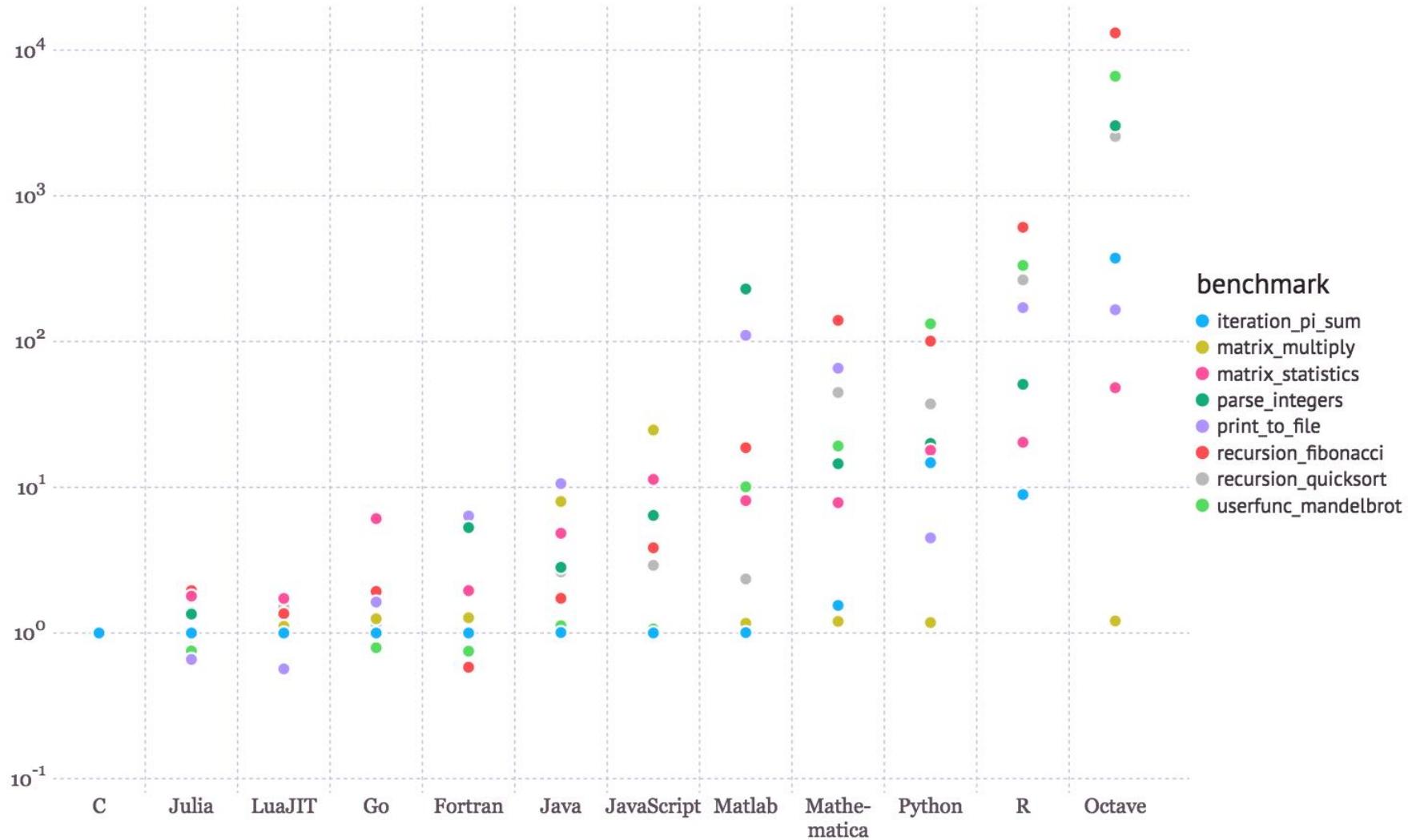


Text 형태

Interpreter vs Compiler



JIT 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
playground

https://en.wikipedia.org/wiki/Read–eval–print_loop

REPL



IDE



Text Editor



vi, emacs...

■ Online

- Python.org Online Console: www.python.org/shell
- Python Fiddle: pythonfiddle.com
- Repl.it: repl.it
- Trinket: trinket.io
- Python Anywhere: 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](#)

■ pip

- 모듈이나 패키지를 쉽게 설치할 수 있도록 도와주는 도구 (De Facto)

- 설치된 패키지 확인
pip list / pip freeze
pip show '패키지 이름'
- 패키지 검색
pip search '패키지 이름'
- 패키지 설치
pip install '패키지 이름'
- 패키지 업그레이드
pip intall --upgrade '패키지 이름' pip install -U '패키지 이름'
- 패키지 삭제
pip uninstall '패키지 이름'

■ conda

■ 윈도우 라이브러리

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

■ Distribution = a software bundle

- 용량 문제, 관리 문제, 충돌 문제, 버전 호환성 등의 문제를 해결 가능
- Python interpreter (Python standard library) + package managers



[Anaconda](#)



[ActivePython](#)



[Enthought Canopy](#)



[python\(x,y\)](#)

■ 가상환경

○ Application-level isolation of Python environments

- virtualenv / venv
 - VirtualenvWrapper
- buildout
 - <http://www.buildout.org/en/latest/>

○ System-level environment isolation

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

■ Packing isolation

- pipenv
 - dependencies를 간단하게 관리

■ REPL 실행

- 대화형 모드 - 인터프리터
- \$ python3

■ file.py 파일 실행

- \$ python3 file.py

■ "print('hi')" 문 실행

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

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

- \$ python3 -i file.py

■ json/tool.py 모듈 실행

- \$ python3 -m json.tool

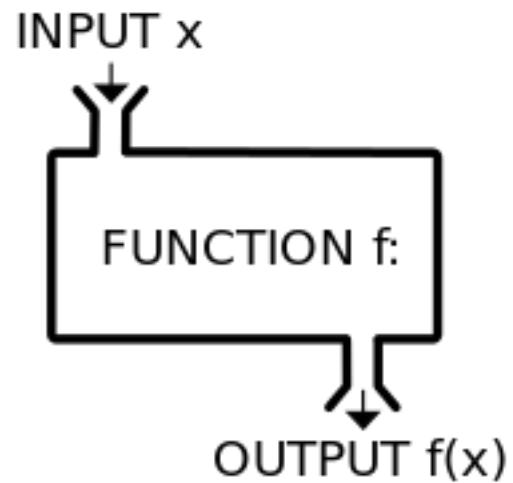
■ 대화형 환경에서는, 마지막에 인쇄된 표현식은 변수 _로 표현 가능

- tax = 12.5 / 100
- price = 100.50
- price * tax 12.5625
- price + _ 113.0625
- round(_, 2) 113.06

프로그래밍의 구성 요소

프로그래밍

문법(키워드, 식, 문)을 이용해서
값을 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!)

자비로운 종신 독재자 : [Guido van Rossum](#), 파이썬의 창시자

https://en.wikipedia.org/wiki/Benevolent_dictator_for_life

- 아름다운 것이 보기 싫은 것보다 좋다.
- 명시적인 것이 암묵적인 것보다 좋다.
- 간단한 것이 복합적인 것보다 좋다.
- 복합적인 것이 복잡한 것보다 좋다.
- 수평한 것이 중첩된 것보다 좋다.
- 희소한 것이 밀집된 것보다 좋다.
- 가독성이 중요하다.
- 규칙을 무시할 만큼 특별한 경우는 없다.
- 하지만 실용성이 순수함보다 우선한다.
- 에러가 조용히 넘어가서는 안된다.
- 명시적으로 조용히 만든 경우는 제외한다.
- 모호함을 만났을 때 추측의 유혹을 거부해라.
- 하나의 — 가급적 딱 하나의 — 확실한 방법이 있어야 한다.
- 하지만 네덜란드 사람(귀도)이 아니라면 처음에는 그 방법이 명확하게 보이지 않을 수 있다.
- 지금 하는 것이 안하는 것보다 좋다.
- 하지만 안하는 것이 이따금 지금 당장 하는 것보다 좋을 때가 있다.
- 설명하기 어려운 구현이라면 좋은 아이디어가 아니다.
- 설명하기 쉬운 구현이라면 좋은 아이디어다.
- 네임스페이스는 아주 좋으므로 더 많이 사용하자!

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
identifier로 사용하지 않음

Expression vs Statement

표현식 vs 구문

Declaration vs Assignment

선언 vs 할당(대입)

■ 표현식(expression) = 평가식 = 식

○ 값

○ 값들과 연산자를 함께 사용해서 표현한 것

○ 이후 “평가”되면서 하나의 특정한 결과값으로 축약

➤ 수

– $1 + 1$

» $1 + 1$ 이라는 표현식은 평가될 때 2라는 값으로 계산되어 축약

– 0과 같이 값 리터럴로 값을 표현해놓은 것

➤ 문자열

– 'hello world'

– "hello" + ", world"

➤ 함수

– lambda (일반적으로 Functional Paradigm에서 지원)

○ 궁극적으로 “평가”되며, 평가된다는 것은 결국 하나의 값으로 수렴한다는 의미

➤ python에서는 기본적으로 left-to-right로 평가

■ 구문(statement) = 문

○ 예약어(reserved word, keyword)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 함수나 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.

➤ 조건(분기) : 조건에 따라 수행할 작업을 나눌 때 사용.

– Ex) if 문

➤ 반복문 : 특정한 작업을 반복수행할때 사용.

– Ex) for 문 및 while 문

➤ 예외처리

■ 값에 대한 type이 중요함

■ 값에 따라 서로 다른 기술적인 체계가 필요

- 지원하는 연산 및 기능이 다르기 때문

■ 컴퓨터에서는 이진수를 사용해서 값을 표현하고 관리

- 정확하게 표현하지 못할 수가 있음

- 숫자형 = numeric

- 산술 연산을 적용할수 있는 값
- 정수 : 0, 1, -1 과 같이 소수점 이하 자리가 없는 수. 수학에서의 정수 개념과 동일. (int)
- 부동소수 : 0.1, 0.5 와 같이 소수점 아래로 숫자가 있는 수. (float)
- 복소수 : Python에서 기본적으로 지원

- 문자, 문자열

- 숫자 "1", "a", "A" 와 같이 하나의 낱자를 문자라 하며, 이러한 문자들이 1개 이상있는 단어/문장와 같은 텍스트
- Python에서는 낱자와 문자열 사이에 구분이 없이 기본적으로 str 타입을 적용
 - byte, bytearry

- 불리언 = boolean

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

○ Compound = Container = Collection

- 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
 - list : 순서가 있는 원소들의 묶음
 - tuple : 순서가 있는 원소들의 묶음. 리스트와 혼동하기 쉬운데 단순히 하나 이상의 값을 묶어서 하나로 취급하는 용도로 사용
 - range
- Lookup
 - mapping
 - » dict : 그룹내의 고유한 이름인 키와 그 키에 대응하는 값으로 이루어지는 키값 쌍(key-value pair)들의 집합.
 - set : 순서가 없는 고유한 원소들의 집합.

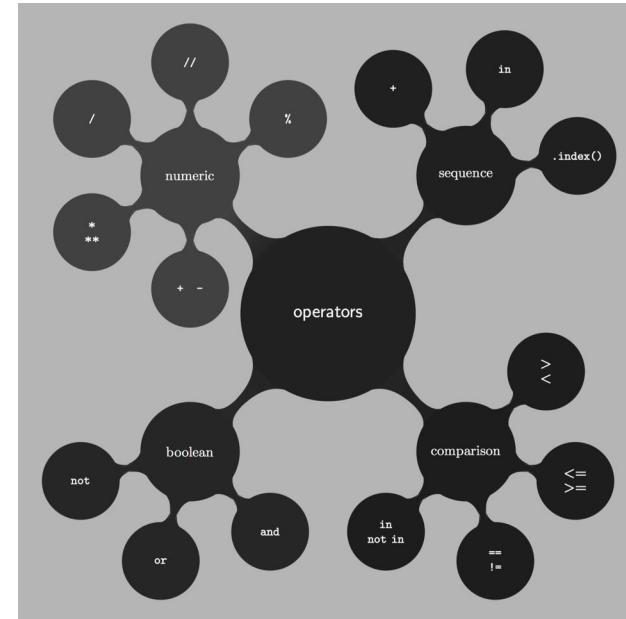
○ None

- 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
- None 이라고 대문자로 시작하도록 쓰며, 실제 출력해보아도 아무것도 출력되지 않음.
- 값이 없지만 False 나 0 과는 다르기 때문에 어떤 값으로 초기화하기 어려운 경우에 쓰기도 함

Literal vs Type(Object-oriented)

값, 기호로 고정된 값을 대표
간단하게 값 생성

Operation



■ 산술연산

- 계산기

■ 비교연산

- 동등 및 대소를 비교. 참고로 '대소'비교는 '전후'비교가 사실은 정확한 표현
- 비교 연산은 숫자값 뿐만 아니라 문자열 등에 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	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
	*, /, //, %	multiplication, division, floor division, modulo
	+x, -x, ~x	unary positive, unary negation, bitwise negation
highest precedence	**	exponentiation

PEMDAS :

Parentheses - Exponentiation - Multiplication - Division - Addition - Subtraction

Python 문법

■ Python 문법의 특징

- 규칙(키워드 등)의 수가 적고 대부분의 규칙이 일관된 맥락을 가지고 있음

- Python 3 : 35 (True, False)
- Python 2 : 31
- C++ : 62
- Java : 53
- Visual Basic : >120

이해하고 배우기 쉬움?

- **case sensitive**

- **space** sensitive (Indentation / line oriented)

- Indentation used to define **block structure**
 - **Multiple statements can occur at same level of indentation**
- Implicit continuation across unclosed bracketing ((...), [...], {...})
- Forced continuation after `\` at end of line
- possible to combine multiple statements (Compound statements) on a single line
 - semi-colon(;) is a statement separator
 - **strongly discouraged**

■ <https://docs.python.org/ko/3/index.html>

■ 주석

- 해시 문자(#로 시작하고 줄의 끝까지 이어집니다.

- 주석은 줄의 처음에서 시작할 수도 있고, 공백이나 코드 뒤에 나올 수도 있습니다.
- 하지만 문자열 리터럴 안에는 들어갈 수 없습니다. 문자열 리터럴 안에 등장하는 해시 문자는 주석이 아니라 해시 문자일 뿐입니다.
- 주석은 코드의 의미를 정확히 전달하기 위한 것이고, 파이썬이 해석하지 않는 만큼, 예를 입력할 때는 생략해도 됩니다.

■

- Line-based, i.e., but independent of indentation (but advisable to do so)
- There are no multi-line comments

- " " "

■ #!

- Shell-script friendly, i.e., **#!** as first line
- [https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))

■ # -*- coding: <encoding-name> -*-

- https://docs.python.org/3/reference/lexical_analysis.html#encoding-declarations

- 대부분 언어는 서로 다른 스타일로 작성될 (또는 더 간략하게, 포맷될) 수 있음. 어떤 것들은 다른 것들보다 더 읽기 쉬움. 다른 사람들이 코드를 읽기 쉽게 만드는 것은 항상 좋은 생각이고, 훌륭한 코딩 스타일을 도입하는 것은 그렇게 하는 데 큰 도움을 줌.

■ 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

■ Pythonic style

- 다른 언어와 다른 특색
- 간단 명료, 풍부한 표현력
- Encouraged style is idiomatically more functional than procedural

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

■ 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 = CapWords

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

■ Snake Case

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

■ PEP8

○ 들려 쓰기에 4-스페이스를 사용하고, 탭을 사용하지 마세요.

- 4개의 스페이스는 작은 들여쓰기 (더 많은 중첩 도를 허락한다) 와 큰 들여쓰기 (읽기 쉽다) 사이의 좋은 절충입니다.
탭은 혼란을 일으키고, 없애는 것이 최선입니다.

○ 79자를 넘지 않도록 줄 넘김 하세요.

- 이것은 작은 화면을 가진 사용자를 돋고 큰 화면에서는 여러 코드 파일들을 나란히 볼 수 있게 합니다.

○ 함수, 클래스, 함수 내의 큰 코드 블록 사이에 빈 줄을 넣어 분리하세요.

- 가능하다면, 주석은 별도의 줄로 넣으세요.
- 독스트링을 사용하세요.

○ 연산자들 주변과 콤마 뒤에 스페이스를 넣고, 괄호 바로 안쪽에는 스페이스를 넣지 마세요

- `a = f(1, 2) + g(3,4).`

○ 클래스와 함수들에 일관성 있는 이름을 붙이세요;

- 관례는 클래스의 경우 CamelCase, 함수와 메서드의 경우 lower_case_with_underscores 입니다. 첫 번째 메서드 인자
의 이름으로는 항상 `self` 를 사용하세요

○ 여러분의 코드를 국제적인 환경에서 사용하려고 한다면 특별한 인코딩을 사용하지 마세요. 어떤 경우에 도 파이썬의 기본, UTF-8, 또는 단순 ASCII조차, 이 최선입니다.

○ 마찬가지로, 다른 언어를 사용하는 사람이 코드를 읽거나 유지할 약간의 가능성만 있더라도, 식별자에 ASCII 이외의 문자를 사용하지 마세요.

Assignment

binding

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



Assignment operator

```
parrot = 'Dead'
```

Simple assignment

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

Assignment of single value to multiple targets
= Aliasing

```
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

`& =`
`| =`
`^ =`
`>> =`
`<< =`

■ 일반적 의미

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

■ 프로그래밍에서의 변수

- 값을 기억해 두고 필요할 때 활용할 수 있음
 - 중간 계산값을 저장하거나 누적 등

■ Python

- 값(객체)을 저장하는 메모리 상의 공간을 가르키는(**object reference**) **이름**
 - A variable is a name for an object **within a scope**
 - **None** is a reference to nothing
- Python은 모든 것이 객체이므로 변수보다는 **식별자**로 언급. 변수로 통용해서 사용하기도 함
 - **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
 - **Python에는 이름있는 상수 개념 없음**
- **변수의 경우, 선언 및 할당이 동시에 이루어져야 함**
 - 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

■ case-sensitive names

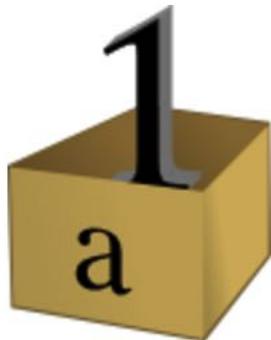
- E.g., functions, classes and variables

■ 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

Object Reference

C



```
int a = 1;
```



```
a = 2;
```



```
int b = a;
```



Python



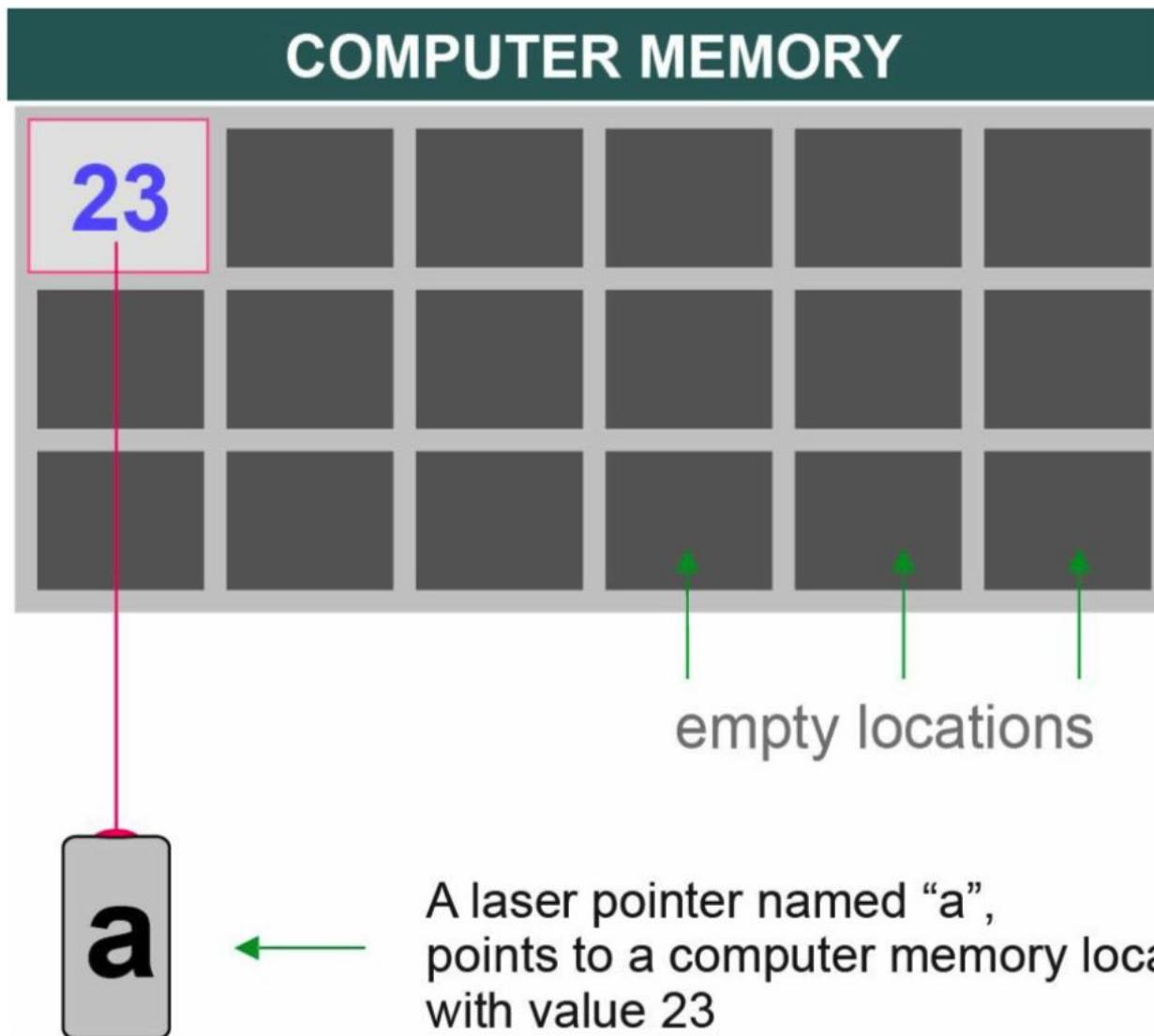
```
a = 1
```



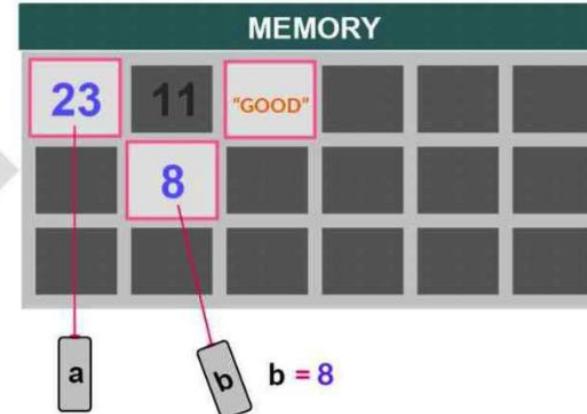
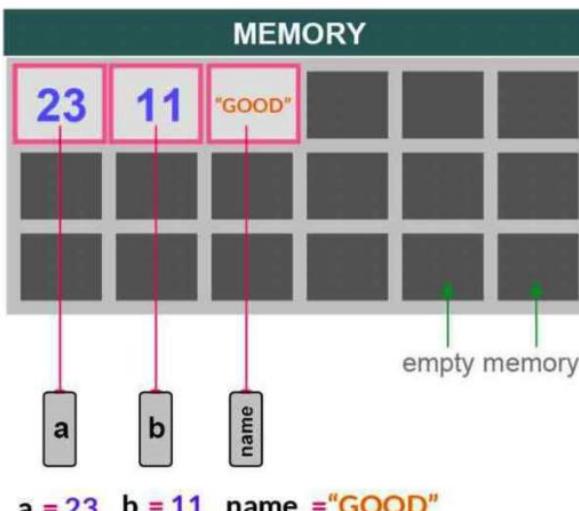
```
a = 2
```



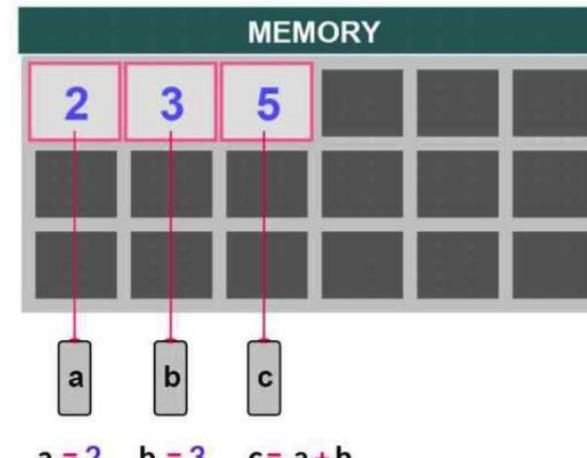
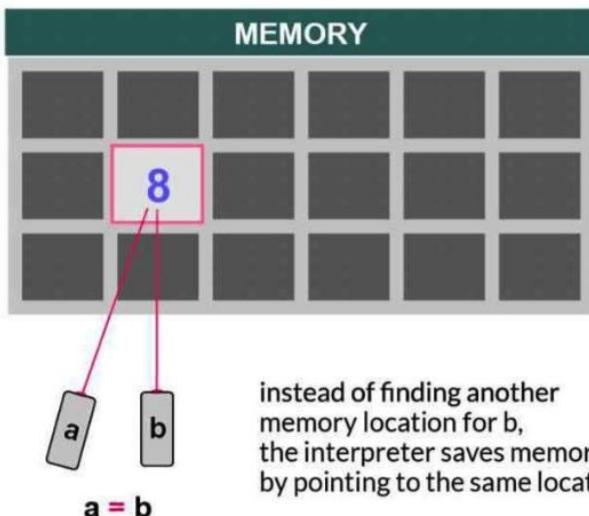
```
b = a
```



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



Variable Creation

Code

```
status = "off"
```

What Computer Does

Variables

Objects



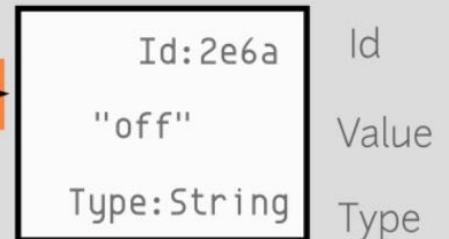
Step 1: Python creates an object

```
status = "off"
```

Variables

Objects

status →



Step 2: A variable is created

Rebinding Variables

Code

a = 400

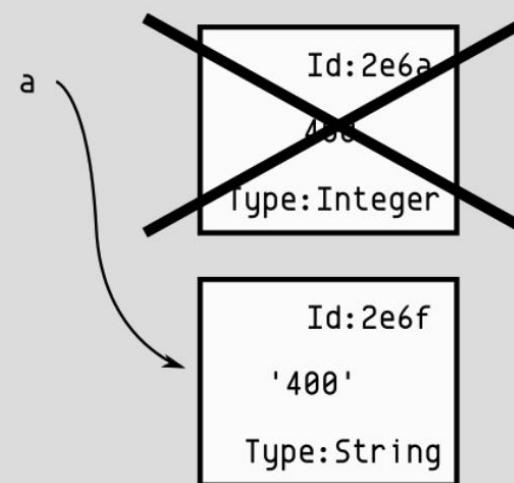
What Computer Does

Variables Objects



a = '400'

Variables Objects



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

Old Object is Garbage Collected

Mutability

■ 객체 (Objects)

- 데이터(data)와 행위를 추상화한 것(abstraction)
- Python 프로그램의 모든 데이터는 객체나 객체 간의 관계로 표현
 - (폰 노이만(Von Neumann)의 "프로그램 내장식 컴퓨터(stored program computer)" 모델을 따르고, 또 그 관점에서 코드 역시 객체로 표현
- 모든 객체는 **아이덴티티(identity)**, **형(type)**, **값(value)**을 가짐.
 - **아이덴티티**
 - 한 번 만들어진 후에는 변경되지 않음.
 - 메모리상에서의 객체의 주소로 생각.
 - 'is' 연산자는 두 객체의 아이덴티티를 비교;
 - id() 함수는 아이덴티티를 정수로 표현한 값을 반환.
 - **형**
 - 객체가 지원하는 연산들을 정의 (예를 들어, "길이를 갖고 있나? ")
 - 그 형의 객체들이 가질 수 있는 가능한 값들을 정의.
 - type() 함수는 객체의 형(이것 역시 객체다)을 돌려줌.
 - 아이덴티티와 마찬가지로, 객체의 형 (*type*) 역시 변경되지 않음
 - » 어떤 제한된 조건으로, 어떤 경우에 객체의 형을 변경하는 것이 가능.

■ 값을 변경할 수 있는 객체들을 **가변**(*mutable*)

■ 만들어진 후에 값을 변경할 수 없는 객체들을 **불변**(*immutable*)

○ 가변 객체에 대한 참조를 저장하고 있는 불변 컨테이너의 값은 가변 객체의 값이 변할 때 변경된다고 볼 수도 있음; 하지만 저장하고 있는 객체들의 집합이 바뀔 수 없으므로 컨테이너는 여전히 불변이라고 여겨짐. 따라서 불변성은 엄밀하게는 변경 불가능한 값을 갖는 것과는 다름.

○ 객체의 가변성(*mutability*)은 그것의 형에 의해 결정

○ 제자리(*inplace*)에서 멤버를 추가, 삭제 또는 재배치하고 특정 항목을 반환하지 않는 메서드는 컬렉션 인스턴스 자체를 반환하지 않고 `None` 을 반환

■ 형은 거의 모든 측면에서 객체가 동작하는 방법에 영향을 줌.

○ 불변형의 경우, 새 값을 만드는 연산은 실제로는 이미 존재하는 객체 중에서 같은 형과 값을 갖는 것을 돌려줄 수 있음. 반면에 가변 객체에서는 이런 것이 허용되지 않음.

➤ `a = 1; b = 1` 후에, `a` 와 `b` 는 값 1을 갖는 같은 객체일 수도 있고, 아닐 수도 있음.

➤ `c = []; d = []` 후에, `c` 와 `d` 는 두 개의 서로 다르고, 독립적이고, 새로 만들어진 빈 리스트임이 보장

➤ `c = d = []` 는 같은 객체를 `c` 와 `d` 에 대입

Built-in Types

immutable

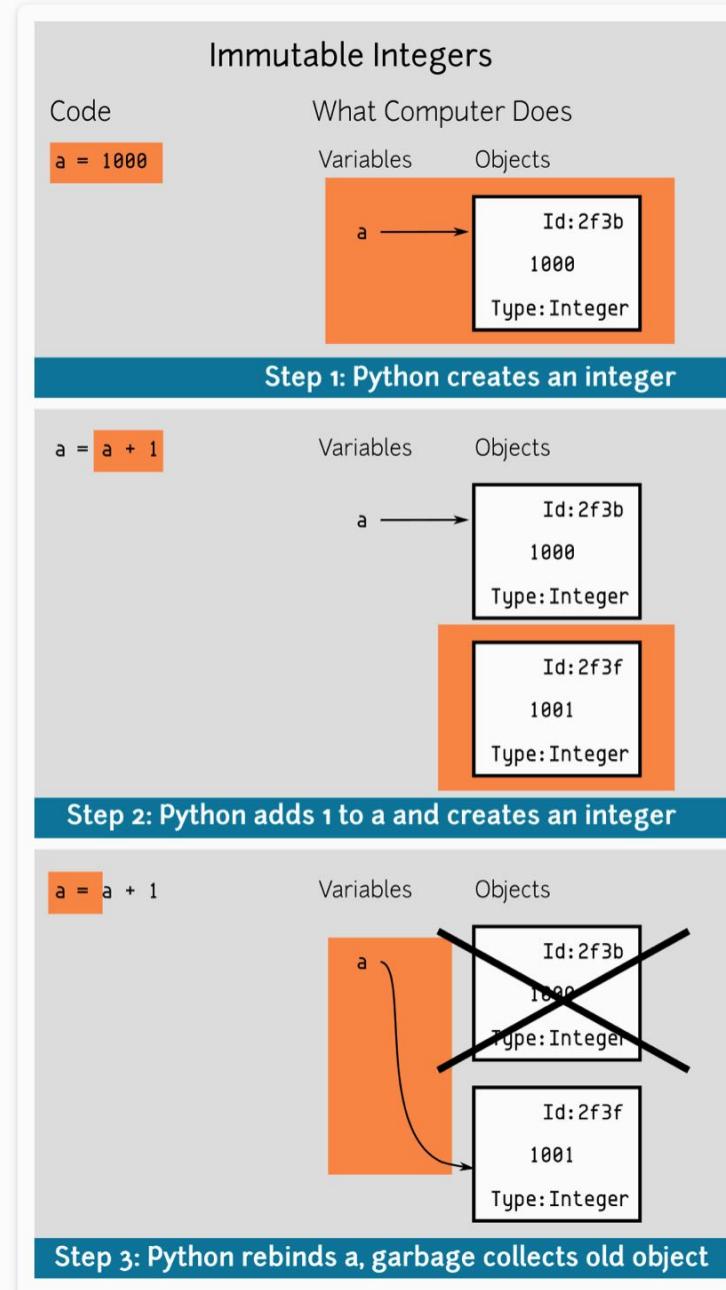
1. Numeric

1.1. int

1.2. float

1.3. complex

1.4. bool



Type		예시
int	int	14, -14
	int (Hex)	0xe
	int (Octal)	0o16
	int (Binary)	0b1110
float	float	14.0, 0.5, .3, 1.
	float	1.4e1, 1.79e+308 1.8e-308
	infinity	case insensible float("inf"), "Inf", "INFINITY" 및 "iNfINity"는 모두 가능
	Not a Number	case insensible float('nan')
complex	complex	14+0j
bool	bool	True, False
Underscore (readability)		1_000

■ 수를 표현하는 기본 리터럴

○ 정수

- 0, 100, 123 과 같은 표현

○ 실수

- 중간에 소수점 0.1, 4.2, 3.13123 와 같은 식
- 0. 으로 시작하는 실수값에서는 흔히 앞에 시작하는 0 제외 가능. .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 2진법, 8진법, 16진법이 존재.

- 2진법 숫자는 0b로 시작(대소문자를 구분하지 않음)
- 8진법 숫자는 0o로 시작(대소문자를 구분하지 않음)
- 16진법숫자는 0x로 시작(대소문자를 구분하지 않음)

■ 숫자 리터럴 중간에 _ 를 쓰는 것은 무시.

○ 원하는 아무자리에나 가능

■ 참/ 거짓을 의미하는 부울대수값. lssubclass(bool,int)

○ 자체가 키워드로 True/False를 사용하여 표현

■ int (Decimal library)

○ arbitrary precision

- Not limited to machine **word size** (overflow/underflow 없음)
- https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic

○ import sys; sys.maxsize

○ It supports the usual operators

- / results in a float, so use // if integer division is needed

■ float (<https://docs.python.org/3.6/tutorial/floatingpoint.html>) (Fraction library)

○ 64-bit double-precision (Almost all platforms) according to the IEEE 754 standard

- https://en.wikipedia.org/wiki/IEEE_754_revision
- maximum value a floating-point number can have is approximately 1.8×10^{308} . (overflow 있음)
 - 이보다 더 큰수는 inf
- The closest a nonzero number can be to zero is approximately 5.0×10^{-324} .
 - Anything closer to zero than that is effectively zero:

>>> x = 1.1 + 2.2

>>> x == 3.3

False

○ import sys; sys.float_info

○ 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.

■ 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

■ 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
- `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

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

$10 // -4 == -3$ or $-10 // 4 == -3$

When the result is negative, the result is rounded down to the next smallest (greater negative) integer:

Operator	Example	Meaning	Result
&	a & b	bitwise AND	Each bit position in the result is the logical AND of the bits in the corresponding position of the operands. (1 if both are 1, otherwise 0.)
	a b	bitwise OR	Each bit position in the result is the logical OR of the bits in the corresponding position of the operands. (1 if either is 1, otherwise 0.)
~	~a	bitwise negation bitwise complement	Each bit position in the result is the logical negation of the bit in the corresponding position of the operand. (1 if 0, 0 if 1.)
^	a ^ b	bitwise XOR (exclusive OR)	Each bit position in the result is the logical XOR of the bits in the corresponding position of the operands. (1 if the bits in the operands are different, 0 if they are the same.)
>>	a >> n	Shift right n places	Each bit is shifted right n places.
<<	a << n	Shift left places	Each bit is shifted left n places.

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
<code>num & num2</code>	<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 << num2</code>	<code>_lshift_</code>	Left shift
<code>num num2</code>	<code>_or_</code>	Bitwise or
<code>num >> 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
<code>f.as_integer_ratio()</code>	Returns num, denom tuple
<code>f.is_integer()</code>	Boolean if whole number

[math](#) 모듈의 복소수 버전이 필요하면, [cmath](#)를 사용

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

Sequence

hold values in an indexable and sliceable order

■ by holding items

○ Container sequences

- list, tuple, and collections.deque can **hold items of different types**. (Heterogeneous)
- 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**. (Homogeneous)
- 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

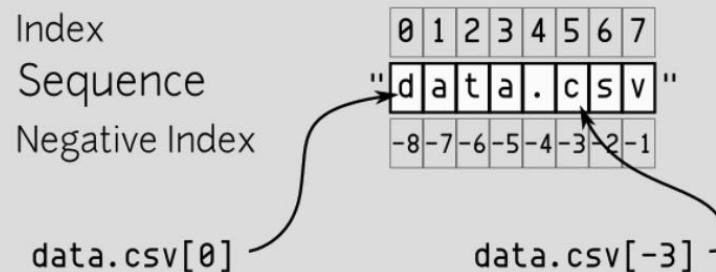
- list, bytearray, array.array, collections.deque, and memoryview

○ Immutable sequences

- tuple, str, and bytes

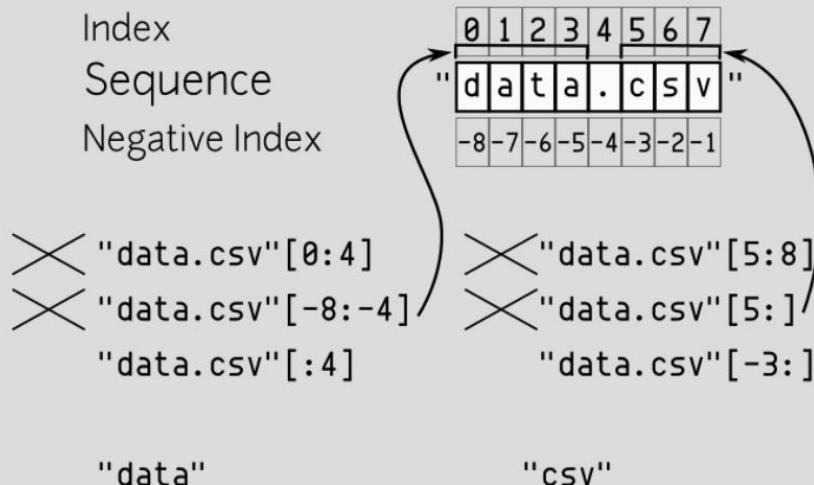
Index Examples

index
= subscript



모든 슬라이스 연산은
요청한 항목들을 포함하
는 새 리스트를 반환. 슬
라이스가 리스트의 새로
운 (얕은) 복사본을 돌려
준다는 뜻

Slicing Examples



■ 너무 큰 값을 인덱스로 사용하는 것은 예러

```
word[42] # the word only has 6 characters
```

```
Traceback (most recent call last): File "<stdin>", line 1, in <module> IndexError: string index out of range
```

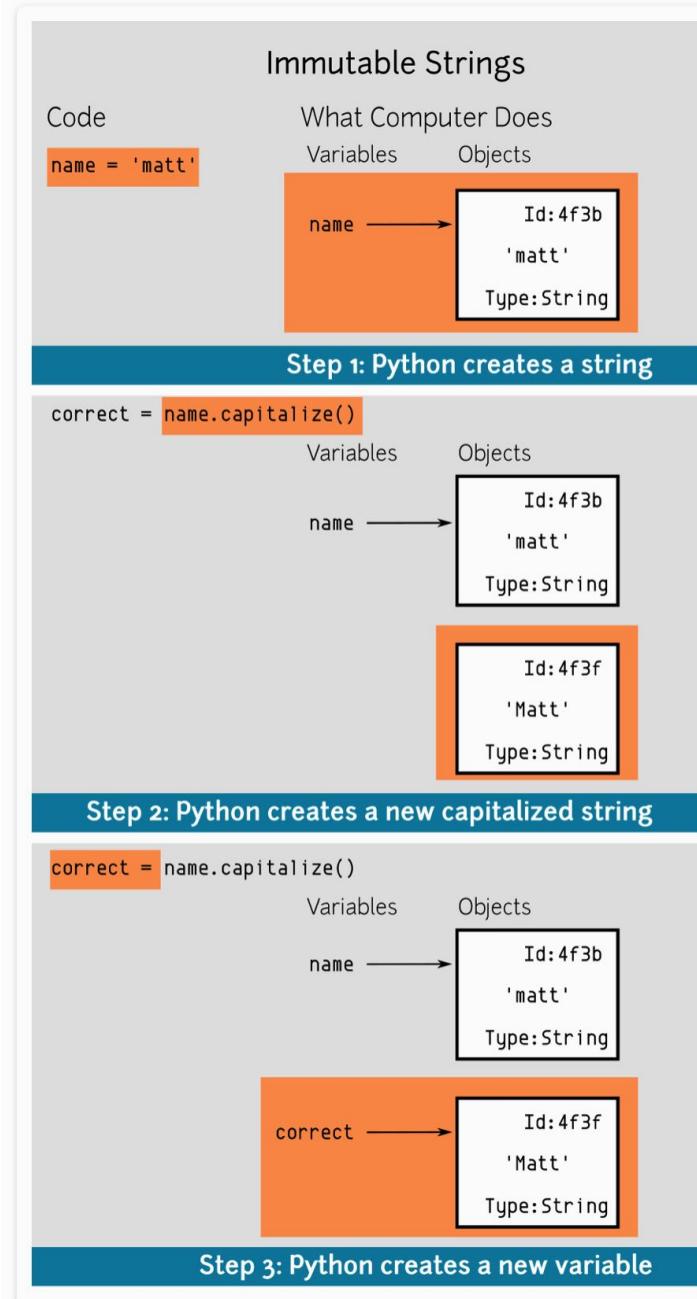
■ 범위를 벗어나는 슬라이스 인덱스는 슬라이싱할 때 부드럽게 처리

```
word[4:42]
```

```
word[42:]
```

immutable, flat sequence

2. String



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

Escape Sequence	Output
\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

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

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

'Unbroken\
 string'
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

■ str

- 글자, 글자가 모여서 만드는 단어, 문장, 여러 줄의 단락이나 글 전체

■ literal

- 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 """
- escape를 허용하지 않는 raw string 리터럴 : r"
- 다른 값을 삽입하는 format string 리터럴 f::
- byte를 정의하는 byte string 리터럴 b"

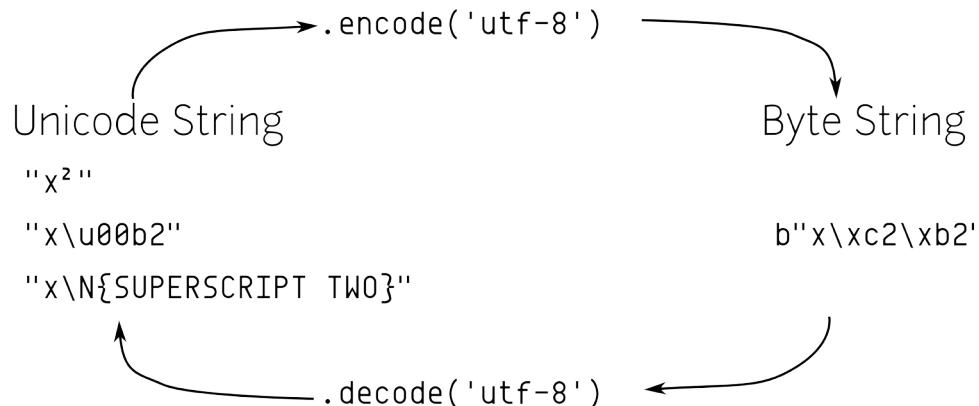
■ str type holds **Unicode** characters

- There is no type for single characters

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

- The bytearray type is used for modifiable byte sequences

Unicode Encoding & Decoding



- bytes는 (HTTP 응답과 같은 파일)과 (네트워크 리소스)는 바이트 스트림으로 전송되기 때문에 이해하는 것이 중요
- 반면에 사람은 유니 코드 문자열의 편의성을 선호. 그렇기에 상호변환을 하는 경우가 많음

<https://feifeiyum.github.io/2017/07/15/python-fp-textbytes/>

Built-in query functions

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

String concatenation

first + second
string * repeat
repeat * string

Substring containment

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

String comparison (lexicographical ordering)

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

Operation	Provided By	Result
s + s2	__add__	String concatenation
"foo" in s	__contains__	Membership
s == s2	__eq__	Equality
s >= s2	__ge__	Greater or equal
s[0]	__getitem__	Index operation
s > s2	__gt__	Greater
s <= s2	__le__	Less than or equal
len(s)	__len__	Length
s < s2	__lt__	Less than
s % (1, 'foo')	__mod__	Formatting
s * 3, 3* s	__mul__	Repetition
s != s2	__ne__	Not equal
repr(s)	__repr__	Programmer friendly string
s.__sizeof__()	__sizeof__	Bytes for internal representation
str(s)	__str__	User friendly string

문자열에 실수를 곱하거나 문자열에 정수를 더하는 연산 ?
ValueError 에러

■ 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

■ interpolation (내삽)

○ 문자열에 대한 특별한 연산

- "Tom has 3 bananas and 4 apples." 라는 문자열이 있다고 할 때, 이것을 리터럴로 정의하는 것은 그 내용이 소스코드에 고정되는, 하드 코딩(hard coding)
- 문자열은 한 번 생성된 이후로 변경되지 않는 immutable이므로 Tom이 가지고 있는 사과나 바나나의 개수가 바뀌었을 때, 그 내용을 적절히 변경해 줄 수가 없음

○ 문자열내에 동적으로 변할 수 있는 값을 삽입하여 상황에 따라 다른 문자열을 만드는 방법

- mini-language
- 문자열 내삽의 기본원리는 문자열 내에 다른 값으로 바뀔 치환자를 준비해두고, 필요한 시점에 치환자를 실제 값의 내용으로 바꿔 문자열을 생성
 - 전통적인 포맷 치환자(%)를 사용하는 방법 (Old Style)
 - » 문자열 % (값, ...)의 형식을 이용해서 문자열 내로 변수값을 밀어넣는 방법
 - 문자열의 .format() 메소드를 사용하는 방법 (New Style)
 - » 치환자의 구분없이 사용할 수 있으며, 각 값을 포맷팅할 수 있는 장점
 - Literal String (python 3.6+)
 - » 포맷 메소드를 사용하지 않고 리터럴만으로 2.의 방법을 사용
 - Template String

○ **format string**을 사용자가 입력한 값을 이용하려면, security 이슈를 피하기 위하여 **Template String** 사용. 그렇지 않을 경우, **python3.6+** 사용시 **Literal String** 방식 사용. **python3.6+** 사용하지 않을 경우 **New Style** 사용

■ 포맷 치환자 (printf-like)

- 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
- 문자열 치환자는 퍼센트 문자 뒤에 포맷형식을 붙여서 치환자를 정의. 치환자를 포함하는 문자열과 각 치환자에 해당하는 값의 tuple을 % 기호로 연결하여 표현
 - %d
 - 정수값
 - d 앞에는 자리수와 채움문자를 넣을 수 있음
 - %04d : 앞의 0은 채움문자이고 뒤는 포맷의 폭 (%04d 는 0으로 시작하는 네자리 정수를 의미)
 - 13이라는 값을 포맷팅할 때, %d 에 치환하면 "13"이 되지만, %04d 에 치환되는 경우에는 "0013"으로 치환
 - %f
 - 실수값
 - f 앞에는 .3 과 같이 소수점 몇 째자리까지 표시할 것인지를 결정하는 확장정보
 - %.3f : 1.5를 "1.500"으로 표시
 - 정수값은 숫자 리터럴과 같이 %b, %o, %x를 이용해서 각각 이진법, 8진법, 16진법으로 표시
 - %s
 - 문자열
 - %r
 - representation으로 타입을 구분하지 않는 값의 표현형
 - 표시되고자 하는 값의 타입이 분명하지 않을 때 사용
 - %d, %f, %s 에 대해서 올바르지 않은 타입의 값을 치환하려 하면 TypeError가 발생할때 사용

```
>>> first_name = "Eric"
>>> last_name = "Idle"
>>> age = 74
>>> profession = "comedian"
>>> affiliation = "Monty Python"
>>> "Hello, %s %. You are %s. You are a %s. You were a me
mber of %s." % (first_name, last_name, age, profession, af
filiation)
'Hello, Eric Idle. You are 74. You are a comedian. You wer
e a member of Monty Python.'
```

readable enough. However, once you start using several parameters and longer strings, your code will quickly become much less easily readable. Things are starting to look a little messy already: Unfortunately, this kind of formatting isn't great because it is verbose and leads to errors, like not displaying tuples or dictionaries correctly.

- 두 개 이상의 문자열 리터럴(즉, 따옴표로 둘러싸인 것들) 가 연속해서 나타나면 자동으로 이어 붙여짐.

```
>>> 'Py' 'thon' 'Python'
```

- 긴 문자열을 쪼개고자 할 때 사용

```
>>> text = ('Put several strings within parentheses ' ... 'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

- 오직 두 개의 리터럴에만 적용될 뿐 변수나 표현식에는 해당하지 않음

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal ...
SyntaxError: invalid syntax
```

```
>>> ('un' * 3) 'ium' ...
SyntaxError: invalid syntax
```

■ format() : python 2.7+ (PEP 3101)

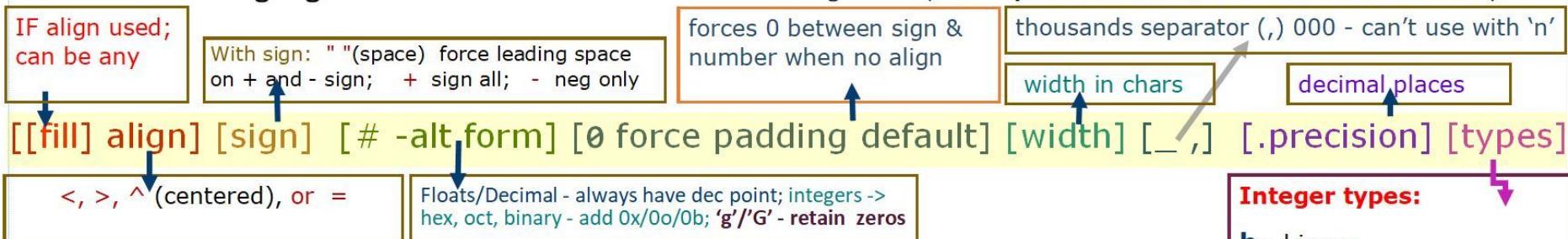
- 전통적인 포맷 치환자가 타입을 가린다는 제약이 있고, 포맷팅의 방법이 제한
 - format()은 Default-ordered, positional and named parameter substitution are supported
 - format()은 숫자값인 경우에는 특별히 d, f, x, b, o 등의 표현타입도 정의 가능
 - format()은 왼쪽 정렬이나 중앙정렬도 설정
- python의 built-in format() 함수와 str의 format 메소드와 동일한 동작
 - format(문자열, 치환값)
 - str.format(치환값)
- {} 를 사용
 - {} 자체가 하나의 값을 의미하며, 번호를 부여해서 한 번 받은 값을 여러번 사용할 수 있음.
 - <https://docs.python.org/3/library/string.html#format-specification-mini-language>

```
'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

`format(object, "mini-language-string")
 "{:mini-language-string}".format(object)`

How the **mini-language** statements are ordered and structured in general: (*Note: symbols must be in the order as shown below!*)



"Hello {name}, your balance is {blc:9.3f}".format(name="Adam", blc=230.2346)

Hello Adam, your balance is ███230.█35

Integer types:	
b	- binary
c	- Unicode char
d	- base 10 integer
o	- Octal
x	- Hex - lower cs
X	- Hex - upper cs
n	- like d but uses local separator definitions
Float/decimal types:	
e	- scientific, e - exponent
E	- E for exponent
f	- fixed point (default 6)

{**0**} – reference first positional argument
{ } – reference implicit positional argument
{result} – reference keyword argument
{bike.tire} – reference attribute of argument
{names[**0**]}) – reference first element of argument

!s – Call `str()` on argument
!r – Call `repr()` on argument
!a – Call `ascii()` on argument

```
class Cat:  
... def __init__(self, name):  
...     self.name = name  
... def __format__(self, data):  
...     return "Format"  
... def __str__(self):  
...     return "Str"  
... def __repr__(self):  
...     return "Repr"  
  
cat = Cat("Fred")  
print("{} {!s} {!a} {!r}".format(cat, cat, cat, ... cat))
```

Format Str Repr Repr

```
>>> first_name = "Eric"
>>> last_name = "Idle"
>>> age = 74
>>> profession = "comedian"
>>> affiliation = "Monty Python"
>>> print(("Hello, {first_name} {last_name}. You are {age}. " +
>>> "You are a {profession}. You were a member of {affiliation}.") \
>>> .format(first_name=first_name, last_name=last_name, age=age, \
>>> profession=profession, affiliation=affiliation)) 'Hello, Eric Idle.
You are 74. You are a comedian. You were a member of Monty Python.'
```

If you had the variables you wanted to pass to **.format()** in a dictionary, then you could just unpack it with **.format(**some_dict)** and reference the values by key in the string, but there has got to be a better way to do this.

■ 복잡하지 않음 (simpler and less powerful mechanism)

- 임의의 변수에 접근할 수 없음
- 복잡한 경우, 다른 mini-language를 사용하는 format 방식 사용

```
from string import Template
t = Template('Hey, $name!')
t.substitute(name=name)

template_string = 'Hey $name, there is a $error error!'
Template(template_string).substitute(name=name, error=hex(errno))

SECRET = 'this-is-a-secret'

class Error:
    def __init__(self):
        pass

err = Error()
user_input = '{error.__init__.globals().__[SECRET]}'
user_input.format(error=err)

user_input = '${error.__init__.globals().__[SECRET]}'
Template(user_input).substitute(error=err)
```

Operation	Result
s.capitalize()	Capitalizes a string
s.casefold()	Lowercase in a unicode compliant manner
s.center(w, [char])	Center a string in w spaces with char (default " ")
s.count(sub, [start, [end]])	Count sub in s between start and end
s.encode(encoding, errors= 'strict')	Encode a string into bytes
s.endswith(sub)	Check for a suffix
s.expandtabs(tabszie=8)	Replaces tabs with spaces
s.find(sub, [start, [end]])	Find substring or return -1
s.format(*args, **kw)	Format string
s.format_map(mapping)	Format strings with a mapping
s.index(sub, [start, [end]])	Find substring or raise ValueError
s.isalnum()	Boolean if alphanumeric
s.isalpha()	Boolean if alphabetic
s.isdecimal()	Boolean if decimal
s.isdigit()	Boolean if digit
s.isidentifier()	Boolean if valid identifier
s.islower()	Boolean if lowercase
s.isnumeric()	Boolean if numeric
s.isprintable()	Boolean if printable
s.isspace()	Boolean if whitespace
s.istitle()	Boolean if titlecased
s.isupper()	Boolean if uppercased

Operation	Result
s.join(iterable)	Return a string inserted between sequence
s.ljust(w, [char])	Left justify in w spaces with char (default ' ')
s.lower()	Lowercase
s.lstrip([chars])	Left strip chars (default spacing).
s.partition(sub)	Split string at first occurrence of substring, return (before, sub, after)
s.replace(old, new, [count])	Replace substring with new string
s.rfind(sub, [start, [end]])	Find rightmost substring or return -1
s.rindex(sub, [start, [end]])	Find rightmost substring or raise ValueError
s.rjust(w, [char])	Right justify in w spaces with char (default " ")
s.rpartition(sub)	Rightmost partition
s.rsplit([sep, [maxsplit=-1]])	Rightmost split by sep (defaults to whitespace)
s.rstrip([chars])	Right strip
s.split([sep, [maxsplit=-1]])	Split a string into sequence around substring
s.splitlines(keepends=False)	Break string at line boundaries
s.startswith(prefix, [start, [end]])	Check for prefix
s.strip([chars])	Remove leading and trailing whitespace (default) or chars
s.swapcase()	Swap casing of string
s.title()	Titlecase string
s.translate(table)	Use a translation table to replace strings
s.upper()	Uppercase
s.zfill(width)	Left fill with 0 so string fills width (no truncation)

Container

3. Another Sequence

- 3.1. list (mutable, container)
- 3.2. tuple (immutable, container)
- 3.3. range (immutable, container)

■ 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

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

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

Search methods

Raises exception if value not found



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

Lexicographical ordering
(not for range)

Indexing and slicing

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

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

Concatenation (not for range)

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

■ 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)`

Operation	Provided By	Result
<code>I + I2</code>	<code>__add__</code>	List concatenation (see <code>.extend</code>)
<code>"name" in I</code>	<code>__contains__</code>	Membership
<code>del I[idx]</code>	<code>__del__</code>	Remove item at index idx (see <code>.pop</code>)
<code>I == I2</code>	<code>__eq__</code>	Equality
<code>"{}".format(I)</code>	<code>__format__</code>	String format of list
<code>I >= I2</code>	<code>__ge__</code>	Greater or equal. Compares items in lists from left
<code>I[idx]</code>	<code>__getitem__</code>	Index operation
<code>I > I2</code>	<code>__gt__</code>	Greater. Compares items in lists from left
No hash	<code>__hash__</code>	Set to None to ensure you can't insert in dictionary
<code>I += I2</code>	<code>__iadd__</code>	Augmented (mutates I) concatenation
<code>I *= 3</code>	<code>__imul__</code>	Augmented (mutates I) repetition
<code>for thing in I:</code>	<code>__iter__</code>	Iteration
<code>I <= I2</code>	<code>__le__</code>	Less than or equal. Compares items in lists from left
<code>len(I)</code>	<code>__len__</code>	Length
<code>I < I2</code>	<code>__lt__</code>	Less than. Compares items in lists from left
<code>I * 2</code>	<code>__mul__</code>	Repetition
<code>I != I2</code>	<code>__ne__</code>	Not equal
<code>repr(I)</code>	<code>__repr__</code>	Programmer friendly string
<code>reversed(I)</code>	<code>__reversed__</code>	Reverse
<code>foo * I</code>	<code>__rmul__</code>	Called if foo doesn't implement <code>__mul__</code>
<code>I[idx] = 'bar'</code>	<code>__setitem__</code>	Index operation to set value
<code>I.__sizeof__()</code>	<code>__sizeof__</code>	Bytes for internal representation
<code>str(I)</code>	<code>__str__</code>	User friendly string

Operation	Result
<code>l.append(item)</code>	Append item to end
<code>l.clear()</code>	Empty list (mutates l)
<code>l.copy()</code>	Shallow copy
<code>l.count(thing)</code>	Number of occurrences of thing
<code>l.extend(l2)</code>	List concatenation (mutates l)
<code>l.index(thing)</code>	Index of thing else ValueError
<code>l.insert(idx, bar)</code>	Insert bar at index idx
<code>l.pop([idx])</code>	Remove last item or item at idx
<code>l.remove(bar)</code>	Remove first instance of bar else ValueError
<code>l.reverse()</code>	Reverse (mutates l)
<code>l.sort([key=], reverse=False)</code>	In-place sort, by optional key function (mutates l)

■ 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))

Operation	Provided	Result
t + t2	<code>_add_</code>	Tuple concatenation
"name" in t	<code>_contains_</code>	Membership
t == t2	<code>_eq_</code>	Equality
"{}".format(t)	<code>_format_</code>	String format of tuple
t >= t2	<code>_ge_</code>	Greater or equal. Compares items in tuple from left
t[idx]	<code>_getitem_</code>	Index operation
t > t2	<code>_gt_</code>	Greater. Compares items in tuple from left
hash(t)	<code>_hash_</code>	For set/dict insertion
for thing in t:	<code>_iter_</code>	Iteration
t <= t2	<code>_le_</code>	Less than or equal. Compares items in tuple from left
len(t)	<code>_len_</code>	Length
t < t2	<code>_lt_</code>	Less than. Compares items in tuple from left
t * 2	<code>_mul_</code>	Repetition
t != t2	<code>_ne_</code>	Not equal
repr(t)	<code>_repr_</code>	Programmer friendly string
foo * t	<code>_rmul_</code>	Called if foo doesn't implement <code>_mul_</code>
t. <code>_sizeof_()</code>	<code>_sizeof_</code>	Bytes for internal representation
str()	<code>_str_</code>	User friendly string

Operation	Result
<code>t.count(item)</code>	Count of item
<code>t.index(thing)</code>	Index of thing else ValueError

- 많은 경우에 `range()` 가 돌려준 객체는 리스트인 것처럼 동작하지만, 사실 리스트가 아님
- iterate할 때 원하는 시퀀스 항목들을 순서대로 돌려주는 객체이지만, 실제로 리스트를 만들지 않아서 공간을 절약.
- 공급이 소진될 때까지 일련의 항목들을 얻을 수 있는 무엇인가를 기대하는 함수와 구조물들의 타깃으로 적합

```
range(5, 10)
      5, 6, 7, 8, 9
```

```
range(0, 10, 3)
      0, 3, 6, 9
```

```
range(-10, -100, -30)
      -10, -40, -70
```

4. dict(ionary)

1. mapping
2. mutable

■ 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)` Equivalent to calling get with a default of None`dict[key]``dict[key] = value` Automagically creates entry if key not already in dict`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)`

Operation	Provided By	Result
key in d	<code>__contains__</code>	Membership
<code>del d[key]</code>	<code>__delitem__</code>	Delete key
<code>d == d2</code>	<code>__eq__</code>	Equality. Dicts are equal or not equal
<code>"{}".format(d)</code>	<code>__format__</code>	String format of dict
<code>d[key]</code>	<code>__getitem__</code>	Get value for key (see <code>.get</code>)
<code>for key in d:</code>	<code>__iter__</code>	Iteration over keys
<code>len(d)</code>	<code>__len__</code>	Length
<code>d != d2</code>	<code>__ne__</code>	Not equal
<code>repr(d)</code>	<code>__repr__</code>	Programmer friendly string
<code>d[key] = value</code>	<code>__setitem__</code>	Set value for key
<code>d.__sizeof__()</code>	<code>__sizeof__</code>	Bytes for internal representation

Operation	Result
d.clear()	Remove all items (mutates d)
d.copy()	Shallow copy
d.fromkeys(iter, value=None)	Create dict from iterable with values set to value
d.get(key, [default])	Get value for key or return default (None)
d.items()	View of (key, value) pairs
d.keys()	View of keys
d.pop(key, [default])	Return value for key or default (KeyError if not set)
d.popitem()	Return arbitrary (key, value) tuple. KeyError if empty
d.setdefault(k, [default])	Does d.get(k, default). If k missing, sets to default
d.update(d2)	Mutate d with values of d2 (dictionary or iterable of (key, value) pairs)
d.values()	View of values

5. set & frozenset

5.1. set (mutable)

5.2. frozenset (immutable)

■ **set and frozenset both define containers of unique hashable values**

■ **set is mutable and has a display form**

- set() is the empty set, not {}

■ **frozenset is immutable and can be constructed from a set or other iterable**

■ **set classes are implemented using dictionaries.**

- Accordingly, the requirements for set elements are the same as those for dictionary keys; namely, that the element defines both eq() and hash().
- **sets cannot contain mutable elements** such as lists or dictionaries

```
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 returns
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)

Operation	Provided By	Result
<code>s & s2</code>	<code>__and__</code>	Set intersection (see <code>.intersection</code>)
<code>"name" in s</code>	<code>__contains__</code>	Membership
<code>s == s2</code>	<code>__eq__</code>	Equality. Sets are equal or not equal
<code>"{}".format(s)</code>	<code>__format__</code>	String format of set
<code>s >= s2</code>	<code>__ge__</code>	<code>s</code> in <code>s2</code> (see <code>.issuperset</code>)
<code>s > s2</code>	<code>__gt__</code>	Strict superset (<code>s >= s2</code> but <code>s != s2</code>).
No hash	<code>__hash__</code>	Set to None to ensure you can't insert in dictionary
<code>s &= s2</code>	<code>__iand__</code>	Augmented (mutates <code>s</code>) intersection (see <code>.intersection_update</code>)
<code>s = s2</code>	<code>__ior__</code>	Augmented (mutates <code>s</code>) union (see <code>.update</code>)
<code>s -= s2</code>	<code>__isub__</code>	Augmented (mutates <code>s</code>) difference (see <code>.difference_update</code>)
<code>for thing in s:</code>	<code>__iter__</code>	Iteration
<code>s ^= s2</code>	<code>__ixor__</code>	Augmented (mutates <code>s</code>) xor (see <code>.symmetric_difference_update</code>)
<code>s <= s2</code>	<code>__le__</code>	<code>s2</code> in <code>s</code> (see <code>.issubset</code>)
<code>len(s)</code>	<code>__len__</code>	Length
<code>s < s2</code>	<code>__lt__</code>	Strict subset (<code>s <= s2</code> but <code>s != s2</code>).
<code>s != s2</code>	<code>__ne__</code>	Not equal
<code>s s2</code>	<code>__or__</code>	Set union (see <code>.union</code>)
<code>foo & s</code>	<code>__rand__</code>	Called if <code>foo</code> doesn't implement <code>__and__</code>
<code>repr(s)</code>	<code>__repr__</code>	Programmer friendly string
<code>foo s</code>	<code>__ror__</code>	Called if <code>foo</code> doesn't implement <code>__or__</code>
<code>foo - s</code>	<code>__rsub__</code>	Called if <code>foo</code> doesn't implement <code>__sub__</code>
<code>foo ^ s</code>	<code>__rxor__</code>	Called if <code>foo</code> doesn't implement <code>__xor__</code>
<code>s.__sizeof__()</code>	<code>__sizeof__</code>	Bytes for internal representation
<code>str(s)</code>	<code>__str__</code>	User friendly string
<code>s - s2</code>	<code>__sub__</code>	Set difference (see <code>.difference</code>)
<code>s ^ s2</code>	<code>__xor__</code>	Set xor (see <code>.symmetric_difference</code>)

Operation	Result
s.add(item)	Add item to s (mutates s)
s.clear()	Remove elements from s (mutates s)
s.copy()	Shallow copy
s.difference(s2)	Return set with elements from s and not s2
s.difference_update(s2)	Remove s2 items from s (mutates s)
s.discard(item)	Remove item from s (mutates s). No error on missing item
s.intersection(s2)	Return set with elements from both sets
s.intersection_update(s2)	Update s with members of s2 (mutates s)
s.isdisjoint(s2)	True if there is no intersection of these two sets
s.issubset(s2)	True if all elements of s are in s2
s.issuperset(s2)	True if all elements of s2 are in s
s.pop()	Remove arbitrary item from s (mutates s). KeyError on missing item
s.remove(item)	Remove item from s (mutates s). KeyError on missing item
s.symmetric_difference(s2)	Return set with elements only in one of the sets
s.symmetric_difference_update(s2)	Update s with elements only in one of the sets (mutates s)
s.union(s2)	Return all elements of both sets
s.update(s2)	Update s with all elements of both sets (mutates s)

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}

coercion (코어션) : 같은 형의 두 인자를 수반하는 연산이 일어나는 동안, 한 형의 인스턴스를 다른 형으로 묵시적으로 변환하는 것. 예를 들어, int(3.15) 는 실수를 정수 3 으로 변환. 하지만, 3+4.5 에서, 각 인자는 다른 형이고 (하나는 int, 다른 하나는 float), 둘을 더하기 전에 같은 형으로 변환해야 함.. 그렇지 않으면 TypeError 를 일으킴. 코어션 없이는, 호환되는 형들조차도 프로그래머가 같은 형으로 정규화해주어야 함. 예를 들어, 그냥 3+4.5 하는 대신 float(3)+4.5

Control Flow

Condition-based loop

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:

...

1. 조건문

- 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

not 은 비논리 연산자들보다 낮은 우선순위를 갖습니다. 그래서, not a == b 는 not (a == b) 로 해석되고, a== not b 는 문법 오류

Boolean

and Logical and

or Logical or

not Negation

Identity

is Identical

is not Non-identical

Membership and containment

in Membership

not in Non-membership

Conditional

if else Conditional (ternary) operator

Truthy	Falsey
True	False
Most objects (Non-null references and not otherwise)	None
1 (Non-zero int)	0
3.2 (Non-zero float)	0.0
[1, 2] (Non-empty containers)	[] (empty list)
{'a': 1, 'b': 2} (Non-empty containers)	{ } (empty dict)
'string' (Non-empty strings)	"" (empty string)
'False'	
'0'	

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

■ partially evaluating = short-circuiting evaluation

- I.e., they do not evaluate their right-hand operand if they do not need to
- i.e., if the first comparison is false, no further evaluation occurs

■ They return the last evaluated operand, without any conversion to bool

'Hello' and 'World'
'Hello' or 'World'
[] and [3, 1, 4]
{42, 97} or {42}
{0} and {1} or [2]

'World'
'Hello'
[]
{42, 97}
{1}

Avoiding an Exception

```
a = 0
b = 1
(b / a) > 0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    (b / a) > 0
ZeroDivisionError: division by zero
```

Selecting a Default Value

```
s = string or '<default_value>'
```

```
a = 0
b = 1
a != 0 and (b / a) > 0
a and (b / a) > 0
```

7>1 and "t" in "it" and 4>2+2 and 7>3/0

■ without using an intermediate

○ short-circuiting evaluation

Brief form...

minimum < value < maximum

" != selection in options

Equivalent to...

minimum < value **and** value < maximum

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

■ 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:
- this has similar behavior to an if statement, but it is an **expression, and not a statement**. Python distinguishes these two.
- else is mandatory
- no elif

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

```
last = 'Lennon' if band == 'Beatles' else 'Jones'
```

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><></code>	<code>a != b</code>	Not equal to	True if a is not equal to b False otherwise
<code><</code>	<code>a < b</code>	Less than	True if a is less than b False otherwise
<code><=</code>	<code>a <= b</code>	Less than or equal to	True if a is less than or equal to b False otherwise
<code>></code>	<code>a > b</code>	Greater than	True if a is greater than b False otherwise
<code>>=</code>	<code>a >= b</code>	Greater than or equal to	True if a is greater than or equal to b False otherwise

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

2. 반복문

For Loops Create a Variable

Code

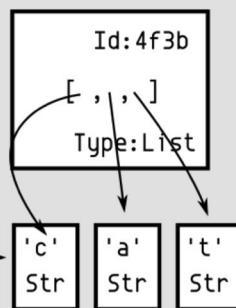
```
for letter in ['c', 'a', 't']:
    print(letter)
```

Output

c

What Computer Does

Variables Objects

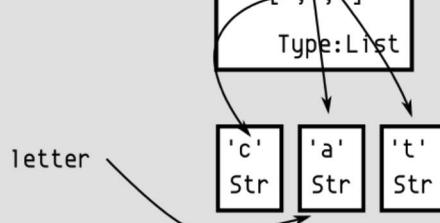


Step 1: During start, letter points to c

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

Output

c
a

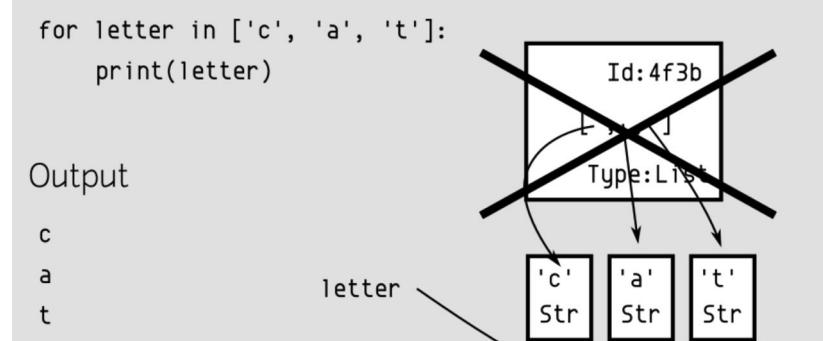


Step 2: Then, letter points to a

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

Output

c
a
t

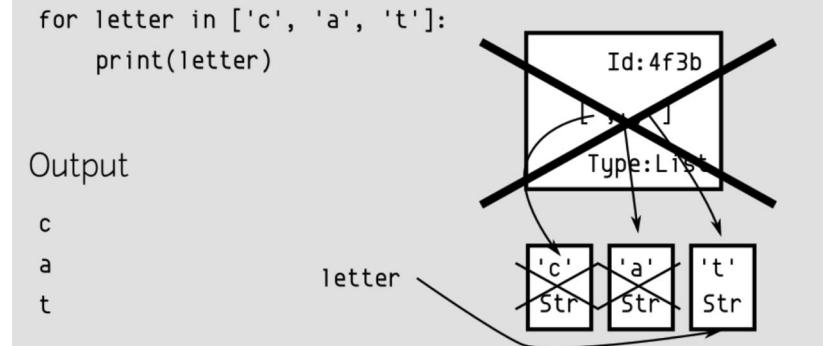


Step 3: Finally, letter points to t, list removed

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

Output

c
a
t



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

■ 파이썬에서 `for` 문은 C나 파스칼에서 사용하던 것과 약간 다릅니다. (파스칼처럼) 항상 숫자의 산술적인 진행을 통해 이터레이션 하거나, (C처럼) 사용자가 이터레이션 단계와 중지 조건을 정의할 수 있도록 하는 대신, 파이썬의 `for` 문은 임의의 시퀀스 (리스트나 문자열)의 항목들을 그 시퀀스에 들어있는 순서대로 이터레이션 합니다.

■ 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)
```

■ 루프 안에서 이터레이트하는 시퀀스를 수정할 필요가 있다면 (예를 들어, 선택한 항목들을 중복시키기), 먼저 사본을 만들 것을 권합니다. 시퀀스를 이터레이트할 때 묵시적으로 사본이 만들어지지는 않습니다. 슬라이스 표기법은 이럴 때 특히 편리합니다:

```
for w in words[:]: # Loop over a slice copy of the entire list.  
    if len(w) > 6:  
        words.insert(0, w)
```

```
words  
['defenestrate', 'cat', 'window', 'defenestrate']
```

for w in words: 를 쓰면, 위의 예는 defenestrate 를 반복해서 넣고 또 넣음으로써, 무한한 리스트를 만들려고 시도하게 됩니다

■ 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
- else 절은 [if](#) 문보다는 [try](#) 문의 else 절과 비슷한 면이 많습니다: [try](#) 문의 else 절은 예외가 발생하지 않을 때 실행되고, 루프의 else 절은 break 가 발생하지 않을 때 실행

```
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
```

- collections module offers variations on standard sequence and lookup types
- collections.abc supports container usage and definition
- set은 list에 비해서 iteration 성능은 떨어지는 지지만 hash 기반으로 만들어지기 때문에 검색 속도는 list에 비해 훨씬 뛰어남

3. Iteration

3.1. iterable & iterator

3.2. comprehension

3.3. iterator & generator

- A number of functions **eliminate** the need to many common loop patterns
- Functional programming tools **reduce** many loops to simple expressions
- A **comprehension** creates a list, set or dictionary without explicit looping
- Iterators and generators support a **lazy approach** to handling series of values

■ The conventional approach to iterating a series of values is to use *for*

- Iterating over one or more iterables directly, as opposed to index-based loops

■ However, part of the secret to good iteration is... don't iterate explicitly

- Think more functionally — use functions and other objects that set up or perform the iteration for you

iterable

■ iterable (이터러블)

- 멤버들을 한 번에 하나씩 돌려줄 수 있는 객체.
- 모든 ([list](#), [str](#), [tuple](#) 같은) 시퀀스 형들, [dict](#) 같은 몇몇 비시퀀스 형들, [파일 객체들](#), [__iter__\(\)](#) 나 [시퀀스](#) 개념을 구현하는 [__getitem__\(\)](#) 메서드를 써서 정의한 모든 클래스의 객체들
- [for](#) 루프에 사용될 수 있고, 시퀀스를 필요로 하는 다른 많은 곳 ([zip\(\)](#), [map\(\)](#), ...)에 사용될 수 있습니다. 이터러블 객체가 내장 함수 [iter\(\)](#)에 인자로 전달되면, 그 객체의 이터레이터를 돌려줍니다. 이 이터레이터는 값들의 집합을 한 번 거치는 동안 유효합니다. 이터러블을 사용할 때, 보통은 [iter\(\)](#)를 호출하거나, 이터레이터 객체를 직접 다룰 필요는 없습니다. [for](#) 문은 이것들을 여러분을 대신해서 자동으로 해주는 데, 루프를 도는 동안 이터레이터를 잡아둘 이름 없는 변수를 만듭니다.

■ iterator (이터레이터)

- 데이터의 스트림을 표현하는 객체.
- 이터레이터의 [__next__\(\)](#) 메서드를 반복적으로 호출하면 (또는 내장 함수 [next\(\)](#)로 전달하면) 스트림에 있는 항목들을 차례대로 돌려줍니다. 더 이상의 데이터가 없을 때는 대신 [StopIteration](#) 예외를 일으킵니다. 이 지점에서, 이터레이터 객체는 소진되고, 이후의 모든 [__next__\(\)](#) 메서드 호출은 [StopIteration](#) 예외를 다시 일으키기만 합니다.
- 이터레이터는 이터레이터 객체 자신을 돌려주는 [__iter__\(\)](#) 메서드를 가질 것이 요구되기 때문에, 이터레이터는 이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있습니다. 중요한 예외는 여러 번의 이터레이션을 시도하는 코드입니다. ([list](#) 같은) 컨테이너 객체는 [iter\(\)](#) 함수로 전달하거나 [for](#) 루프에 사용할 때마다 새 이터레이터를 만듭니다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만듭니다.

■ 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'
```

4. 예외처리



assert

raise

try

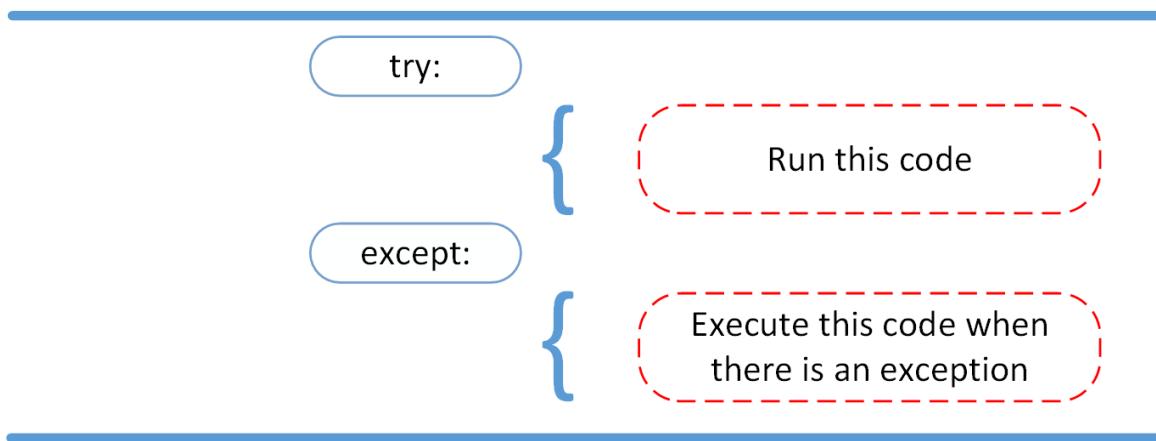
except

else

finally

- **Syntax errors** occur when the parser detects an incorrect statement.
- **Exception error** occurs whenever syntactically correct Python code results in an error.
- 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 are 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

■ Exceptions are objects and their type is defined in a class hierarchy

- The root of the hierarchy is BaseException
- But yours should derive from Exception

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration +-- StopAsyncIteration +--
ArithmetError | +-- FloatingPointError | +-- OverflowError | +--
ZeroDivisionError +-- AssertionError +-- AttributeError +-- BufferError
+-- EOFError +-- ImportError | +-- ModuleNotFoundError +--
LookupError | +-- IndexError | +-- KeyError +-- MemoryError +--
NameError | +-- UnboundLocalError +-- OSError | +-- BlockingIOError
| +-- ChildProcessError | +-- ConnectionError || +-- BrokenPipeError ||
+-- ConnectionAbortedError || +-- ConnectionRefusedError || +--
ConnectionResetError | +-- FileExistsError | +-- FileNotFoundError | +--
InterruptedError | +-- IsADirectoryError | +-- NotADirectoryError | +--
PermissionError | +-- ProcessLookupError | +-- TimeoutError +--
ReferenceError +-- RuntimeError | +-- NotImplemented | +--
RecursionError +-- SyntaxError | +-- IndentationError | +-- TabError +--
SystemError +-- TypeError +-- ValueError | +-- UnicodeError | +--
UnicodeDecodeError | +-- UnicodeEncodeError | +--
UnicodeTranslateError +-- Warning +-- DeprecationWarning +--
PendingDeprecationWarning +-- RuntimeWarning +-- SyntaxWarning
+-- UserWarning +-- FutureWarning +-- ImportWarning +--
UnicodeWarning +-- BytesWarning +-- ResourceWarning
```

■ Exceptions can be handled by type

- except statements are tried in turn until there is a base or direct match

■ 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

The most specific exception type: ZeroDivisionError derives from ArithmeticError

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

try:

...

except ZeroDivisionError as byZero:

...

except ArithmeticError:

...

except:

...

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:



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

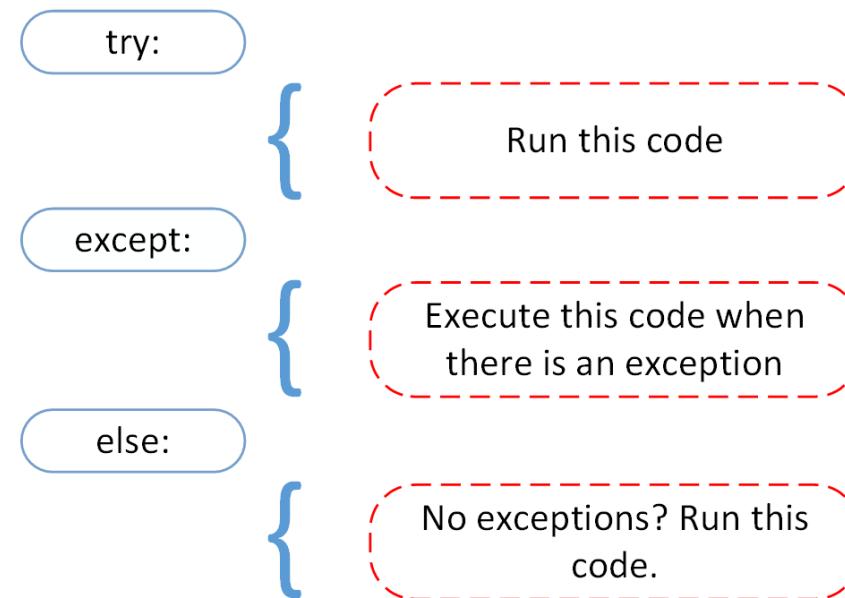
Re-raise current handled exception → raise

Raise a new exception from an existing exception named caughtException → raise Exception from caughtException

Raise a new exception ignoring any previous exception ↑

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

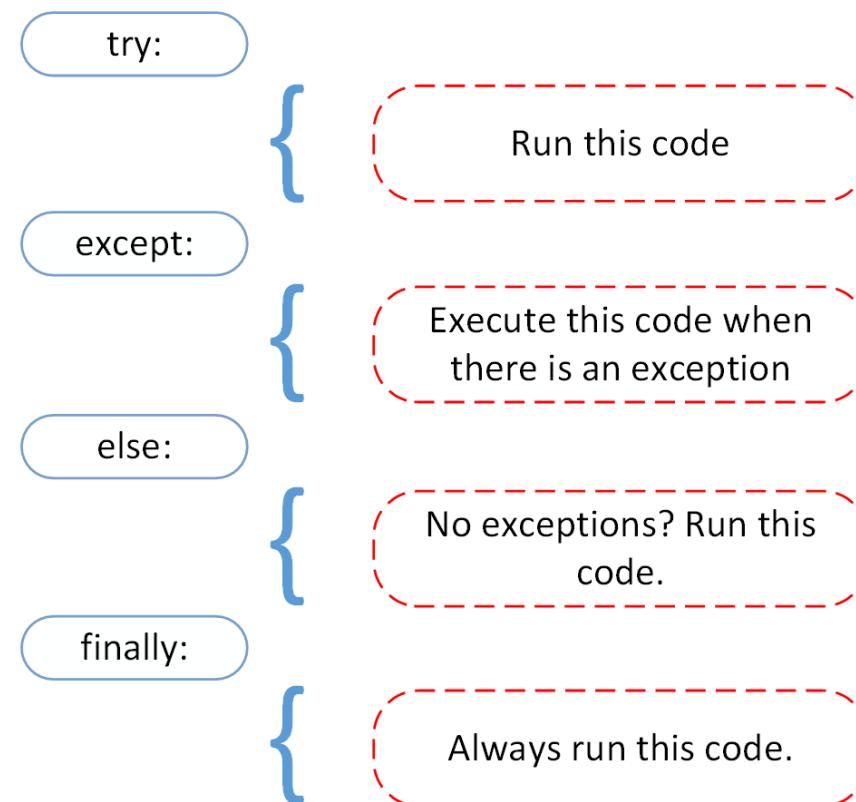
- This is executed after the try if and only if no exception was raised



```
try:  
    hal.open(pod_bay.door())  
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

■ 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

assert value is not None

Equivalent to...

```
if __debug__ and not (value is not None):  
    raise AssertionError
```

assert value is not None, 'Value missing'

Equivalent to...

```
if __debug__ and not (value is not None):  
    raise AssertionError('Value missing')
```

■ 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

■ It's easier to ask forgiveness than permission vs Look before you leap

■ Generally EAFP is preferred, but not always.

- Duck typing

- If it walks like a duck, and talks like a duck, and looks like a duck: it's a duck. (Goose? Close enough.)

- Exceptions

- Use coercion if an object must be a particular type. If x must be a string for your code to work, why not call
 - str(x)
 - instead of trying something like
 - isinstance(x, str)

■ EAFP try/except Example

- You can wrap exception-prone code in a try/except block to catch the errors, and you will probably end up with a solution that's much more general than if you had tried to anticipate every possibility.
- Note: Always specify the exceptions to catch. Never use bare except clauses. Bare except clauses will catch unexpected exceptions, making your code exceedingly difficult to debug.

```
try:
```

```
    return str(x)
```

```
except TypeError:
```

```
...
```

5. Context Manager

■ 어떤 객체들은 열린 파일이나 창 같은 "외부(external)" 자원들에 대한 참조를 포함.

- 객체가 가비지 수거될 때 반납된다고 이해되지만, 가비지 수거는 보장되는 것이 아니므로, 그런 객체들은 외부자원을 반납하는 명시적인 방법 또한 제공.
- 보통 close() 메서드.
- 프로그램을 작성할 때는 그러한 객체들을 항상 명시적으로 닫아야(close) 한다.
- 'try...finally' 문과 'with' 문은 이렇게 하는 편리한 방법을 제공.

function

Defining, calling & passing

■ parameter (매개변수)

- 함수 (또는 메서드) 정의에서 함수가 받을 수 있는 인자 (또는 어떤 경우 인자들) 를 지정하는 이름 붙은 엔티티. 다섯 종류의 매개변수가 있음
 - 위치-키워드 (positional-or-keyword): 위치 인자 나 키워드 인자로 전달될 수 있는 인자를 지정합니다. 이것이 기본 형태의 매개변수입니다, 예를 들어 다음에서 foo 와 bar:
`def func(foo, bar=None): ...`
 - 위치-전용 (positional-only): 위치로만 제공될 수 있는 인자를 지정합니다. 파이썬은 위치-전용 매개변수를 정의하는 문법을 갖고 있지 않습니다. 하지만, 어떤 매장 함수들은 위치-전용 매개변수를 갖습니다 (예를 들어, abs()).
 - 키워드-전용 (keyword-only): 키워드로만 제공될 수 있는 인자를 지정합니다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 * 를 그대로 포함해서 정의할 수 있습니다. 예를 들어, 다음에서 kw_only1 와 kw_only2:
`def func(arg, *, kw_only1, kw_only2): ...`
 - 가변-위치 (var-positional): (다른 매개변수들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정합니다. 이런 매개변수는 매개변수 이름에 * 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 다음에서 args:
`def func(*args, **kwargs): ...`
 - 가변-키워드 (var-keyword): (다른 매개변수들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정합니다. 이런 매개변수는 매개변수 이름에 ** 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 위의 예에서 kwargs.

■ 매개변수는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있습니다.

■ argument (인자)

○ 함수를 호출할 때 함수 (또는 메서드)로 전달되는 값

○ 키워드 인자 (keyword argument)

➤ 함수 호출 때 식별자가 앞에 붙은 인자 (예를 들어, name=) 또는 ** 를 앞에 붙인 딕셔너리로 전달되는 인자. 예를 들어, 다음과 같은 complex() 호출에서 3 과 5 는 모두 키워드 인자:

```
complex(real=3, imag=5)
```

```
complex(**{'real': 3, 'imag': 5})
```

○ 위치 인자 (positional argument)

➤ 키워드 인자가 아닌 인자. 위치 인자들은 인자 목록의 처음에 나오거나 이터러블 의 앞에 * 를 붙여 전달할 수 있음. 예를 들어, 다음과 같은 호출에서 3 과 5 는 모두 위치 인자.

```
complex(3, 5)
```

```
complex(*(3, 5))
```

■ Can be called both with **positional** and with **keyword** arguments

■ Can be defined with **default** arguments

■ Functions always return a value

- **None** if nothing is returned explicitly

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

- callable is a built-in predicate function

➤ A predicate function is a function which purpose is to assert, such as something being either *true* or *false*.

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

```
def a():  
    return 0  
def b():  
    print(0)
```

```
a() + 1 # Yes  
b() + 1 # No
```

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

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

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

```
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)
```

■ 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

■ 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

```
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)
```

```
import time  
def report_arg(my_default=time.time()):  
    print(my_default)  
  
report_arg()  
  
time.sleep(5)  
  
report_arg()
```

```
def append_to_list(value, def_list=[]):  
    def_list.append(value)  
    return def_list
```

```
my_list = append_to_list(1)  
print(my_list)
```

```
my_other_list = append_to_list(2)  
print(my_other_list)
```

■ 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

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

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

```
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
```

■ 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))
```

■ 도큐멘테이션 문자열의 내용과 포매팅에 관한 몇 가지 관례

- 첫 줄은 항상 객체의 목적을 짧고, 간결하게 요약해야 합니다. 간결함을 위해, 객체의 이름이나 형을 명시적으로 언급하지 않아야 하는데, 이것들은 다른 방법으로 제공되기 때문입니다 (이름이 함수의 작업을 설명하는 동사라면 예외입니다). 이 줄은 대문자로 시작하고 마침표로 끝나야 합니다.
- 도큐멘테이션 문자열에 여러 줄이 있다면, 두 번째 줄은 비어있어서, 시각적으로 요약과 나머지 설명을 분리해야 합니다. 뒤따르는 줄들은 하나나 그 이상의 문단으로, 객체의 호출 규약, 부작용 등을 설명해야 합니다.
- 파이썬 파서는 여러 줄 문자열 리터럴에서 들여쓰기를 제거하지 않기 때문에, 도큐멘테이션을 처리하는 도구들은 필요하면 들여쓰기를 제거합니다. 이것은 다음과 같은 관례를 사용합니다. 문자열의 첫줄 뒤에 오는 첫 번째 비어있지 않은 줄이 전체 도큐멘테이션 문자열의 들여쓰기 수준을 결정합니다. (우리는 첫 줄을 사용할 수 없는데, 일반적으로 문자열을 시작하는 따옴표에 붙어있어서 들여쓰기가 문자열 리터럴의 것을 반영하지 않기 때문입니다.) 이 들여쓰기와 "동등한" 공백이 문자열의 모든 줄의 시작 부분에서 제거됩니다. 덜 들여쓰기 된 줄이 나타나지는 말아야 하지만, 나타난다면 모든 앞부분의 공백이 제거됩니다. 공백의 동등성은 탭 확장 (보통 8개의 스페이스) 후에 검사됩니다.

■ 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

■ annotation (어노테이션) 관습에 따라 형 힌트로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수나 반환 값과 연결된 레이블입니다.

- 지역 변수의 어노테이션은 실행 시간에 액세스할 수 없지만, 전역 변수, 클래스 속성 및 함수의 어노테이션은 각각 모듈, 클래스, 함수의 `_annotations_` 특수 어트리뷰트에 저장됩니다.
- 이 기능을 설명하는 [변수 어노테이션](#), [함수 어노테이션](#), [PEP 484](#), [PEP 526](#)을 참조하세요.

■ 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 f(ham: str, eggs: str = 'eggs') -> str:  
    print("Annotations:", f.__annotations__)  
    print("Arguments:", ham, eggs)  
    return ham + ' and ' + eggs  
  
f('spam')  
Annotations: {'ham': <class 'str'>, 'return':  
<class 'str'>, 'eggs': <class 'str'>}  
Arguments: spam eggs  
'spam and eggs'
```

- 함수 어노테이션 은 사용자 정의 함수가 사용하는 형들에 대한 완전히 선택적인 메타데이터 정보입니다 (자세한 내용은 [PEP 3107](#) 과 [PEP 484](#) 를 보세요).
- 어노테이션은 함수의 `_annotations_` 어트리뷰트에 딕셔너리로 저장되고 함수의 다른 부분에는 아무런 영향을 미치지 않습니다. 매개변수 어노테이션은 매개변수 이름 뒤에 오는 콜론으로 정의되는데, 값을 구할 때 어노테이션의 값을 주는 표현식이 뒤따릅니다. 반환 값 어노테이션은 리터럴 -> 와 그 뒤를 따르는 표현식으로 정의되는데, 매개변수 목록과 `def` 문의 끝을 나타내는 콜론 사이에 놓입니다. 다음 예에서 위치 인자, 키워드 인자, 반환 값이 어노테이트 됩니다:

■ 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]
```

■ Lambda calculus

■ 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

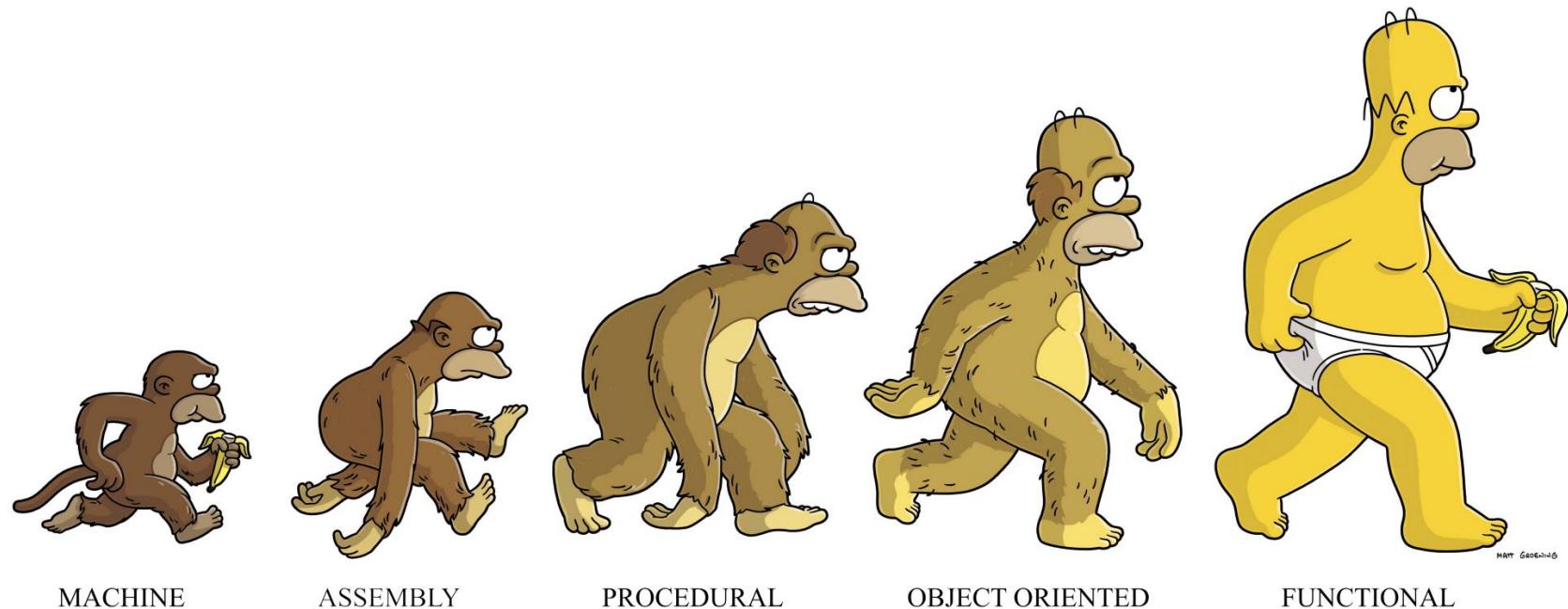
- But defining a one-line named function is preferred over global lambda assignment

```
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.

Functional Programming



■ first class objects = function

- function을 first-class citizen으로 취급 (functions themselves are values or data or objects)
 - In languages where functions are not first class, functions are defined as a relationship between values, which makes them "second class".
- 함수 자체를 인자 (argument)로써 다른 함수에 전달하거나 다른 함수의 결과값으로 반환(return)할수 있고, 함수를 변수에 할당하거나 데이터 구조 안에 저장할 수 있는 함수
 - Functions can be passed as arguments and can be used in assignment
 - This supports many callback techniques and functional programming idioms

■ higher order function = fucntor

- Takes and returns functions
 - Takes one or more functions as arguments, Returns one or more function as its result
- 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
- Not all functions in Pythons 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.)

■ Functional Programming by Wikipedia:

- "Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data". In other words, functional programming promotes code with no side effects, no change of value in variables. It opposes to imperative programming, which emphasizes change of state".
 - No mutable data (no side effect).
 - No state (no implicit, hidden state).
- Functions are **pure** functions in the mathematical sense: their output depend only in their inputs, there is not "environment".
 - 숨겨진 출력은 "부작용(side-effect)"으로, 숨겨진 입력은 "부원인(side-cause)"
 - 모든 입력이 입력으로 선언(숨겨진 것이 없어야 함)
 - 마찬가지로 모든 출력이 출력으로 선언된 함수를 '순수(pure)'
- Functions as principal units of **composition**
- **Immutability**, so querying state and creating new state rather than updating it
 - Prefer to return new state rather than modifying arguments and attributes
- Same result returned by functions called with the same inputs.
- A **declarative** rather than imperative style, emphasizing data structure **relations**

■ What are the advantages?

○ Cleaner code

- “variables” are not modified once defined, so we don’t have to follow the change of state to comprehend what a function, a method, a class, a whole project works.

○ Referential transparency

- Expressions can be replaced by its values. If we call a function with the same parameters, we know for sure the output will be the same (there is no state anywhere that would change it).

- Memoization

- Cache results for previous function calls.

- Idempotence

- Same results regardless how many times you call a function.

- Modularization

- We have no state that pervades the whole code, so we build our project with small, black boxes that we tie together, so it promotes bottom-up programming.

- Ease of debugging

- Functions are isolated, they only depend on their input and their output, so they are very easy to debug.

- Parallelization

- Functions calls are independent.
 - We can parallelize in different process/CPUs/computers/...
 - $\text{result} = \text{func1}(a, b) + \text{func2}(a, c)$ We can execute *func1* and *func2* in parallel because *a* won’t be modified.

- Concurrency

- With no shared data, concurrency gets a lot simpler:
 - No (semaphores, monitors, locks, race-conditions, dead-locks.)

1. FP in Python

■ 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
- Declarative over imperative style, e.g., use of comprehensions over loops

■ Deferred evaluation, i.e., be lazy

- Lazy rather than eager access to values
- 적용 결과 heavy computing을 줄일 수 있음.
 - 수식이 변수에 접근하는 순간 계산되는 것이 아니라, 결과를 구하려고 할 때까지 연산을 미룸으로서 불필요한 연산을 줄일 수 있는 연산 전략의 일종

- 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

■ 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 helps you write in functional style but it doesn't force you to it.**■ Writing in functional style enhances your code and makes it more self documented.
Actually it will make it more thread-safe also.****■ The main support for FP in Python comes from the use of**

- comprehension, lambdas, closures, iterators and generators
- modules functools and itertools.

■ Python still lack an important aspect of FP: Pattern Matching and Tails Recursion.

- There should be more work on tail recursion optimization, to encourage developers to use recursion.

■ Python is **reference-based** language, so objects are shared by default

- Easily accessible data or state-modifying methods can give aliasing surprises
 - An object with more than one reference has more than one name, so we say that the object is **aliased**.
- 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_`.

```
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

2. 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

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

Binary arithmetic operations

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

Unary operations

pos(a)	$+a$
neg(a)	$-a$
invert(a)	$\sim a$

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>

$$f(x, y, z) = x * y + z$$

3, 4, 5라는 인자들을 동시에 적용하여 다음과 같은 결과를 얻을 것이다.

$$f(3, 4, 5) = 3 * 4 + 5 = 17$$

하지만 3 하나만을 적용한다면, 다음과 같을 것이다.

$$f(3, y, z) = g(y, z) = 3 * y + z$$

자 이제 두개의 인자를 가지는 g라는 새로운 함수가 생겼다. 우리는 4를 y에 적용하여 이 함수를 다시 커링할 수 있다.

$$g(4, z) = h(z) = 3 * 4 + z$$

■ Nested functions and lambdas bind to their surrounding scope

- I.e., they are closures

➤ 내부 함수(함수 안의 함수)가 내부 함수를 둘러싼 외부 함수의 변수나 매개변수를 이용했는데 이 외부 함수가 종료되어도 내부 함수에서 사용한 변수 값은 내부 함수 내에서 계속 유지되는 기능

```
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')
```

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

Force non-positional use
of subsequent parameter
s
↓

```
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)
...

3. Comprehensions

■ 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
 - 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]
```

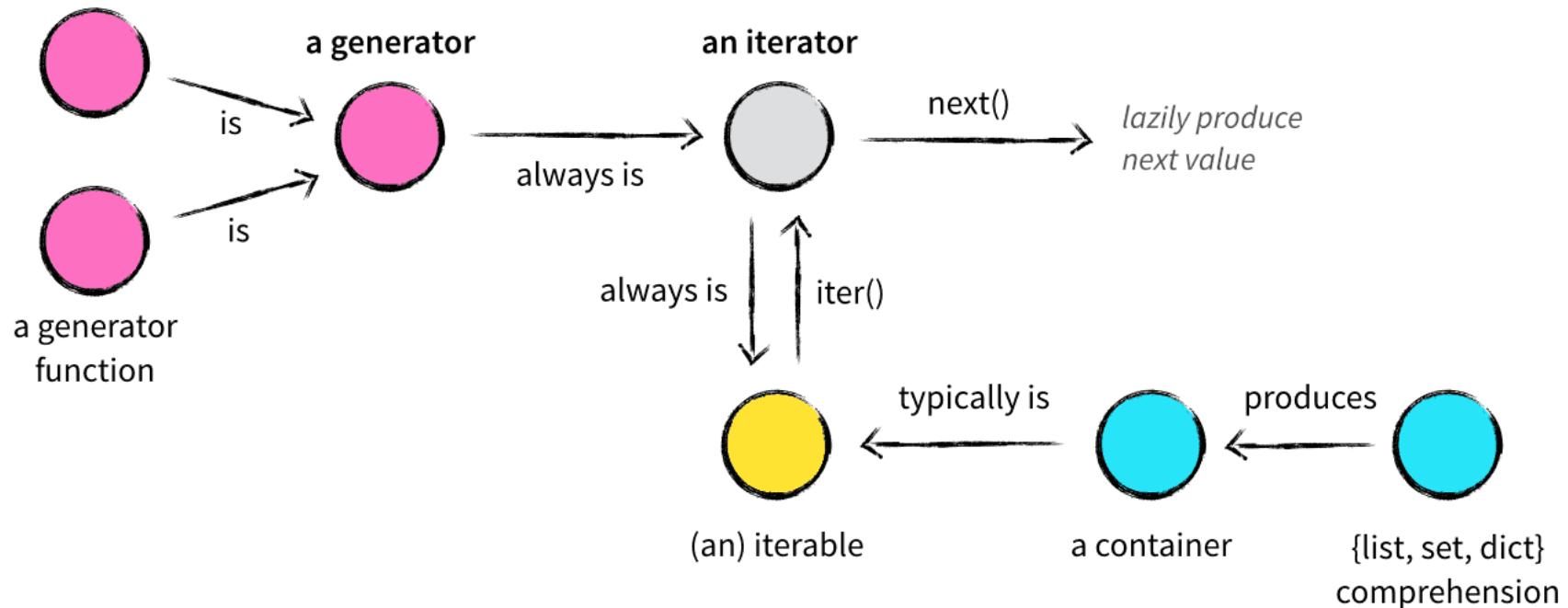
```
values = ['A'] + list(range(2, 11)) + ['J', 'Q', 'K']
suits = ['spades', 'hearts', 'diamonds', 'clubs']
[(value, suit) for suit in suits for value in values]
```



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

4. iterator & generator

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 a n identity operation

The `__next__` special meth od, called by `next`, advanc es the iterator

Iteration is terminated by raising `StopIteration`

```
class Iterator:
```

...

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

```
    return self
```

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

...

```
    raise 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

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

■ 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))
```

■ 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 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))
```

■ 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))
```

■ 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

```
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']
```

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

■ Iterating a list without an index is easy... but what if you need the index?

- Use *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]

■ 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

The mechanics of organising source code

■ Modular programming

- 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.

■ several advantages to **modularizing** code in a large application:

○ 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!*)

■ 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

■ There are actually **three** different ways to define a **module** in Python:

- A module can be written in Python itself.
- A module can be written in C and loaded dynamically at run-time, like the re (regular expression) module.
- A built-in module is intrinsically contained in the interpreter, like the [itertools module](#).

```
sys.path.append(r'C:\Users\john')
sys.path
```

```
import re
re.__file__
import mod
mod.__file__
```

■ 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
- C++, boost

- 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_`



■ 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

Note: Much of the Python documentation states that an `__init__.py` file **must** be present in the package directory when creating a package. This was once true. It used to be that the very presence of `__init__.py` signified to Python that a package was being defined. The file could contain initialization code or even be empty, but it **had** to be present.

Starting with **Python 3.3**, [Implicit Namespace Packages](#) were introduced. These allow for the creation of a package without any `__init__.py` file. Of course, it **can** still be present if package initialization is needed. But it is no longer required.

■ 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

■ extensive library of standard modules

- Much of going beyond the core language is learning to navigate the library
- <https://docs.python.org/3/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 *
from mod import s as string, a as alist
```

Discouraged

Imported

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

This will place the names of *all* objects from <module_name> in the local symbol table, with the exception of any that begin with underscore (_) character.

- does not allow the indiscriminate import * syntax from within a function:

```
def bar():
    from mod import *
```

SyntaxError: import * only allowed at module level

```
try:
    # Non-existent module
    import baz
except ImportError:
    print('Module not found')
```

Module not found

```
try:
    # Existing module, but non-existent object
    from mod import baz
except ImportError:
    print('Object not found in module')
```

Object not found in module

- 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

- When a .py file is imported as a module, Python sets the special **dunder** variable `_name_` to the name of the module. However, if a file is run as a standalone script, `_name_` is (creatively) set to the string '`'__main__'`'. Using this fact, you can discern which is the case at run-time and alter behavior accordingly:

```
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

...

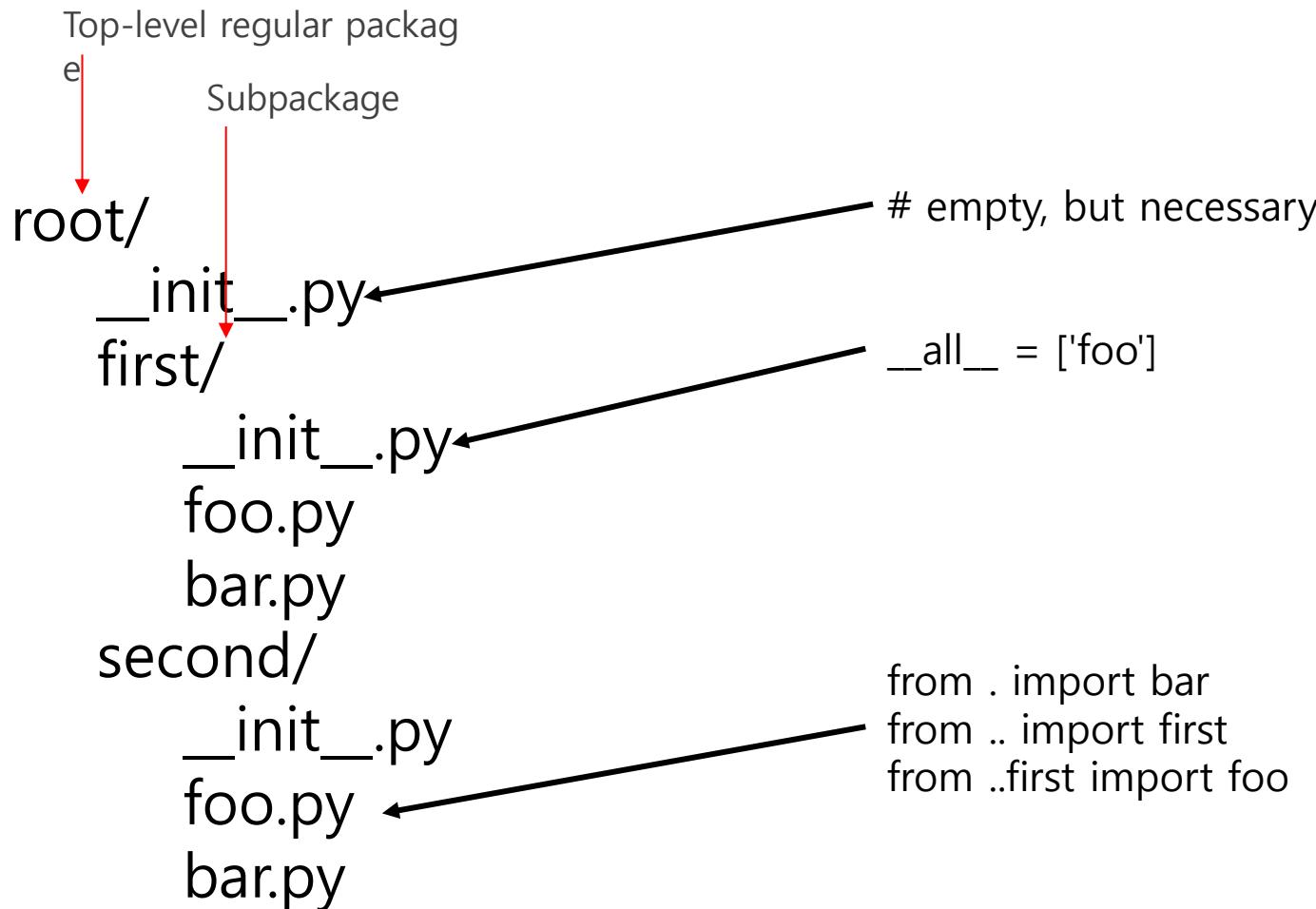
```
def main():  
    ...  
if __name__ == '__main__':  
    main()
```

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

- `__all__` is used by both **packages** and **modules** to control what is imported when import `*` is specified. But *the default behavior differs*.
 - For a package, when `__all__` is not defined, import `*` does not import anything.
 - For a module, when `__all__` is not defined, import `*` imports everything (except—you guessed it—names starting with an underscore).

■ 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`



- Namespaces can also be further nested, for example if we import modules, or if we are defining new classes. In those cases we have to use prefixes to access those nested namespaces. Let me illustrate this concept in the following code block:

- ```
import numpy
import math
import scipy
```
- ```
print(math.pi, 'from the math module')
print(numpy.pi, 'from the numpy package')
print(scipy.pi, 'from the scipy package')
```



```
3.141592653589793 from the math module
3.141592653589793 from the numpy package
3.141592653589793 from the scipy package
```
- (This is also why we have to be careful if we import modules via "from a_module import *", since it loads the variable names into the global namespace and could potentially overwrite already existing variable names)

■ Write a simple script that lists the package and module hierarchy

- Use current directory if no path supplied
- Use `os.listdir`, `os.walk` or `pathlib` for traversal
- No need to open and load modules, just detect packages by matching `__init__.py` and modules by matching `*.py`
- List the results using indentation and/or in some XML-like form

- `_import_` is a low-level hook function that's used to import modules; it can be used to import a module *dynamically* by giving the module name to import as a variable, something the `import` statement won't let you do.
- `importlib.import_module()` is a wrapper around that hook* to produce a nice API for the functionality; it is a very recent addition to Python 2, and has been more fleshed out in Python 3. Codebases that use `_import_` generally do so because they want to remain compatible with older Python 2 releases, e.g. anything before Python 2.7.
- One side-effect of using `_import_` can be that it returns the imported module and doesn't add anything to the namespace; you can import with it without having then to delete the new name if you didn't want that new name; using `import somename` will add `somename` to your namespace, but `_import_('somename')` instead returns the imported module, which you can then ignore. Werkzeug uses the hook for that reason in one location.
- All other uses are to do with dynamic imports. Werkzeug supports Python 2.6 still so cannot use `importlib`.
- * `importlib` is a Pure-Python implementation, and `import_module()` will use that implementation, whilst `_import_` will use a C-optimised version. Both versions call back to `importlib._bootstrap._find_and_load()` so the difference is mostly academic.

class & object

1. class & object

■ Object orientation is founded on encapsulation and polymorphism

- Encapsulation combines function and data into a behavioural entity with identity
- Polymorphism supports uniform object use across compatible object protocols
- Inheritance factors out common and introduces hooks for polymorphism, but is an auxiliary not essential feature of OO

객체 (*Objects*)는 파이썬이 데이터(data)를 추상화한 것(abstraction)이다. 파이썬 프로그램의 모든 데이터는 객체나 객체 간의 관계로 표현된다. (폰 노이만(Von Neumann)의 "프로그램 내장식 컴퓨터(stored program computer)" 모델을 따르고, 또 그 관점에서 코드 역시 객체로 표현된다.)

모든 객체는 아이덴티티(identity), 형(type), 값(value)을 갖는다. 객체의 *아이덴티티*는 한 번만 들어진 후에는 변경되지 않는다. 메모리상에서의 객체의 주소로 생각해도 좋다. '[is](#)' 연산자는 두 객체의 아이덴티티를 비교한다; [`id\(\)`](#) 함수는 아이덴티티를 정수로 표현한 값을 돌려준다.

■ Python classes support a full OO programming model

- Class- and instance-level methods
- Class- and instance-level attributes
- Multiple inheritance

Class statement introduces name and any base classes
`__init__` is automatically invoked after object creation
Convention rather than language dictates that the current object is called `self`

```
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
```

- 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

- 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 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
```

■ Objects are instantiated by using a class name as a function

- Providing any arguments expected by the special `_init_` function
- The `self` argument is provided implicitly

■ 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

```
origin = Point()
print('Is at origin?', origin.is_at_origin())
unit = Point(1, 1)
print(unit.x, unit.y)
```

■ 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`

```
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
```

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

■ 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):      ← Private method  
        return Spam()  
    def egg(self):  
        return self.__bacon() ← Private method call
```

■ 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
```

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

James Whitcomb Riley

In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-game s with.

Alex Martelli

```
class PolarPoint:  
    def __init__(self, r=0, theta=0):  
        self.__r, self.__theta = r, theta  
    def x(self):  
        from math import cos  
        return self.__r * cos(self.__theta)  
    def y(self):  
        from math import sin  
        return self.__r * sin(self.__theta)  
    def r(self):  
        return self.__r  
    def theta(self):  
        return self.__theta
```

Instances of `PolarPoint` and instances of `CartesianPoint` support the same method calls, and are therefore substitutable for one another — quack, quack

```
class CartesianPoint:  
    def __init__(self, x=0, y=0):  
        self.__x, self.__y = x, y  
    def x(self):  
        return self.__x  
    def y(self):  
        return self.__y  
    def r(self):  
        return (self.__x**2 + self.__y**2)**0.5  
    def theta(self):  
        from math import atan  
        return atan(self.__y / self.__x)
```

Leading `_` in identifier names keeps them effectively private

2. Special Methods & Attributes

1. Integrating new types with operators & built-in functions

- Special methods take the place of operator 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

neg(self)	-self
pos(self)	+self
invert(self)	~self
abs(self)	abs(self)

Binary arithmetic operators

add(self, other)	self + other
sub(self, other)	self - other
mul(self, other)	self * other
truediv(self, other)	self / other
floordiv(self, other)	self // other
mod(self, other)	self % other
pow(self, other)	self**other

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,
it 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

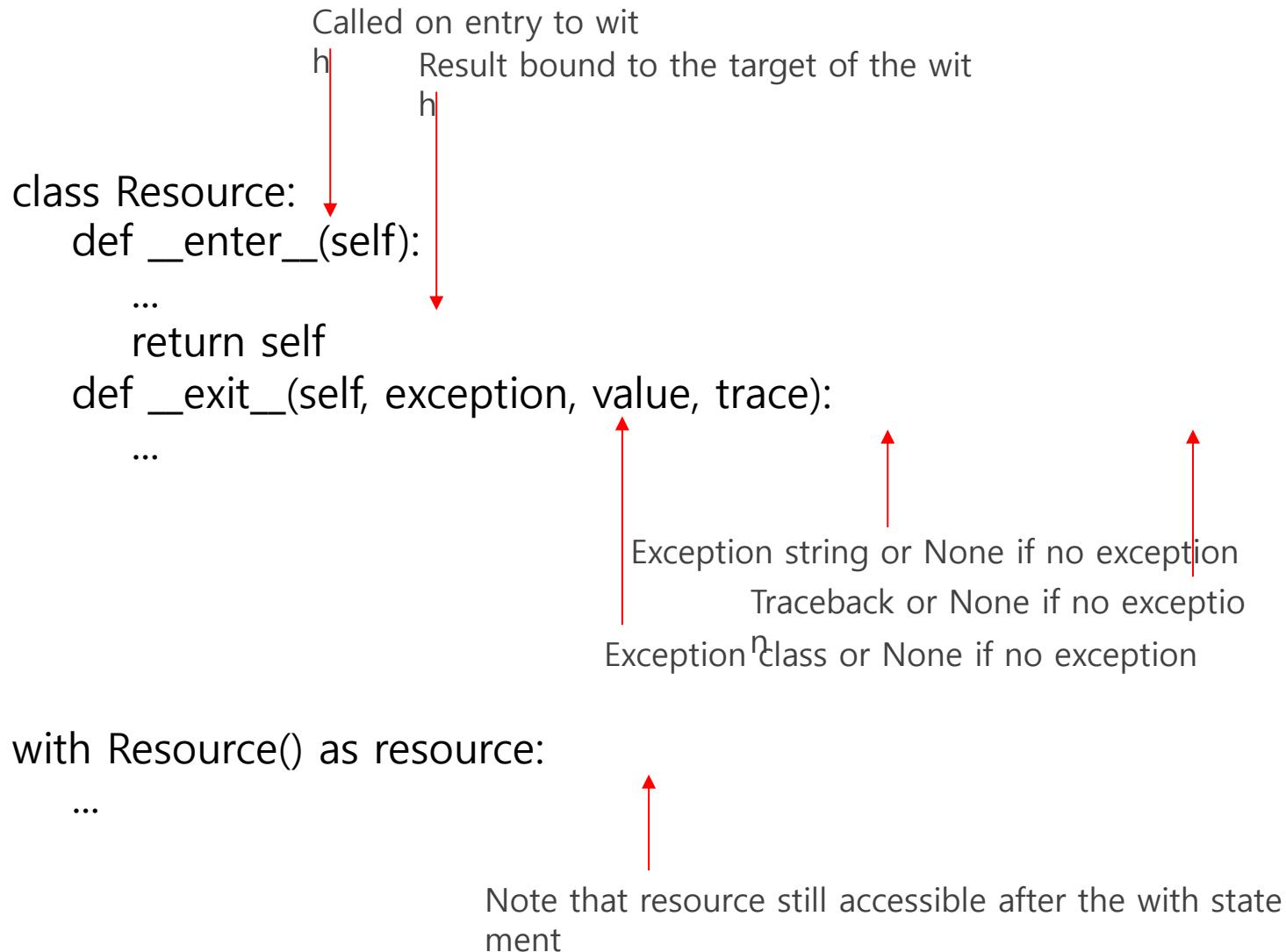
`_enter_(self)`
`_exit_(self, exception, value, traceback)`

■ **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

Record current time using time

Print elapsed time, optionally indicating whether an exception was thrown

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

■ 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
```

Decorator

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

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
```

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

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

```
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

- A metaclass can be considered the class of class
- A class defines how objects behaves; a metaclass 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

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 is the default metaclass in Python**
- **And type is its own class, i.e., type(type) is an identity operation**
- **type is most often used as a query, but it can 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

- 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!

Design Patterns

- 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

- 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

■ 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

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"

■ Substitutability is the strongest form of conformance to a protocol

- A contract of behavioural equivalence
- Following the contract means...

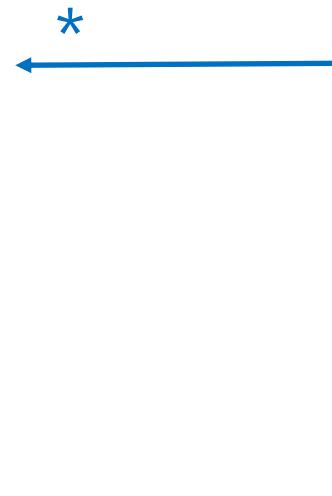
■ An overridden method cannot result in a wider range or cover a narrower domain

- Concrete classes should not be bases
- A derived class should support the same initialisation protocol as its base

- 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

- 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

```
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
    ...
...
```

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

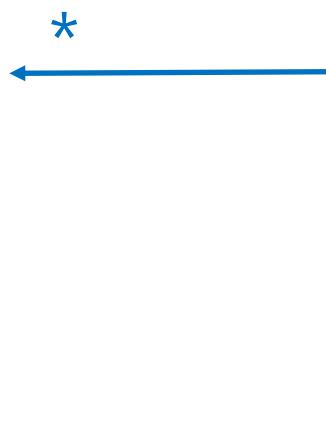
```
class Primitive(Node):
    ...
    def accept(self, visitor):
        visitor.visit_primitive(self)
    ...
```

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

```
class Visitor:
    ...
    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
    ...
```

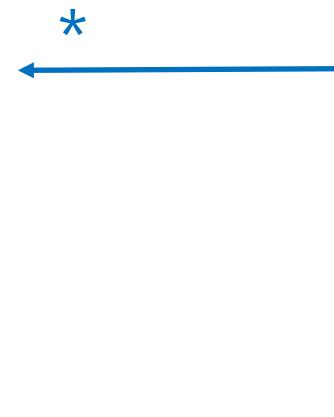


```
class Group(Node):
    def __init__(self, name, children):
        self._name = name
        self._children = tuple(children)
    def map(self, callback):
        for child in self._children:
            callback(child)
    ...
```

```
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`