Control Flow

Condition-based loop

```
while condition:
    ...
else:
    ...
```

Multiway conditional

```
if condition:
    ...
elif condition:
    ...
else:
    ...
```

Exception handling

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

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

Identity

is Identical
is not Non-identical

Membership and containment

in Membership
not in Non-membership

Boolean

and Logical and
or Logical or
not Negation

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

```
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

Selecting a Default Value

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

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
==	a == b	Equal to	True if the value of a is equal to the val ue of b False otherwise
!= <>	a != b	Not equal to	True if a is not equal to b False otherwise
<	a < b	Less than	True if a is less than b False otherwise
<=	a <= b	Less than or equal to	True if a is less than or equal to b False otherwise
>	a > b	Greater than	True if a is greater than b False otherwise
>=	a >= b	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

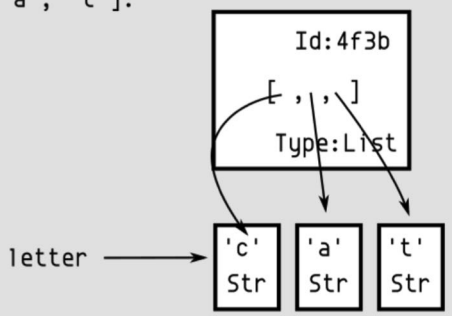
Code

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

Output

```
c
```

What Computer Does

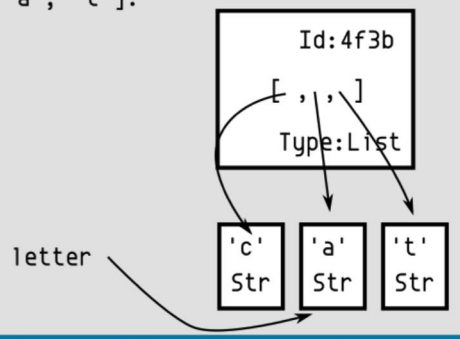


Step 1: During start, letter points to c

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

Output

```
c
a
```

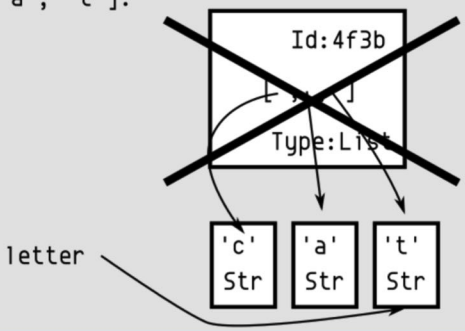


Step 2: Then, letter points to a

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

Output

```
c
a
t
```

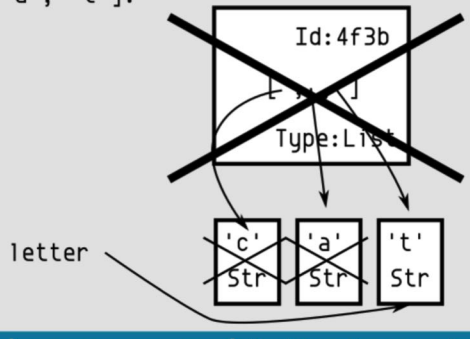


Step 3: Finally, letter points to t, list removed

```
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 C 나 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

Iterating over dictionaries

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

for code in airports:
    print(code)

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

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

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

Iterate over the keys

Iterate over the keys

Iterate over the values

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

- 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 해주는데, 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

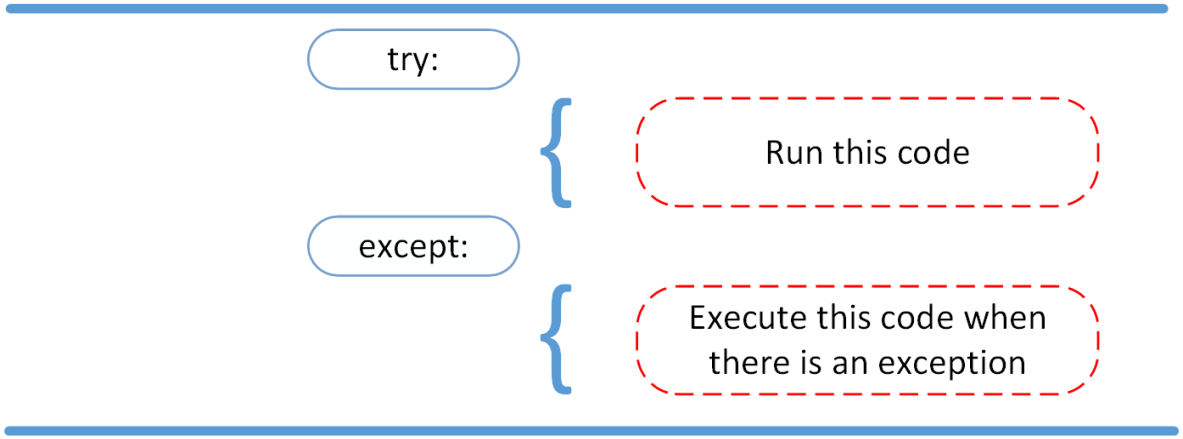
try

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

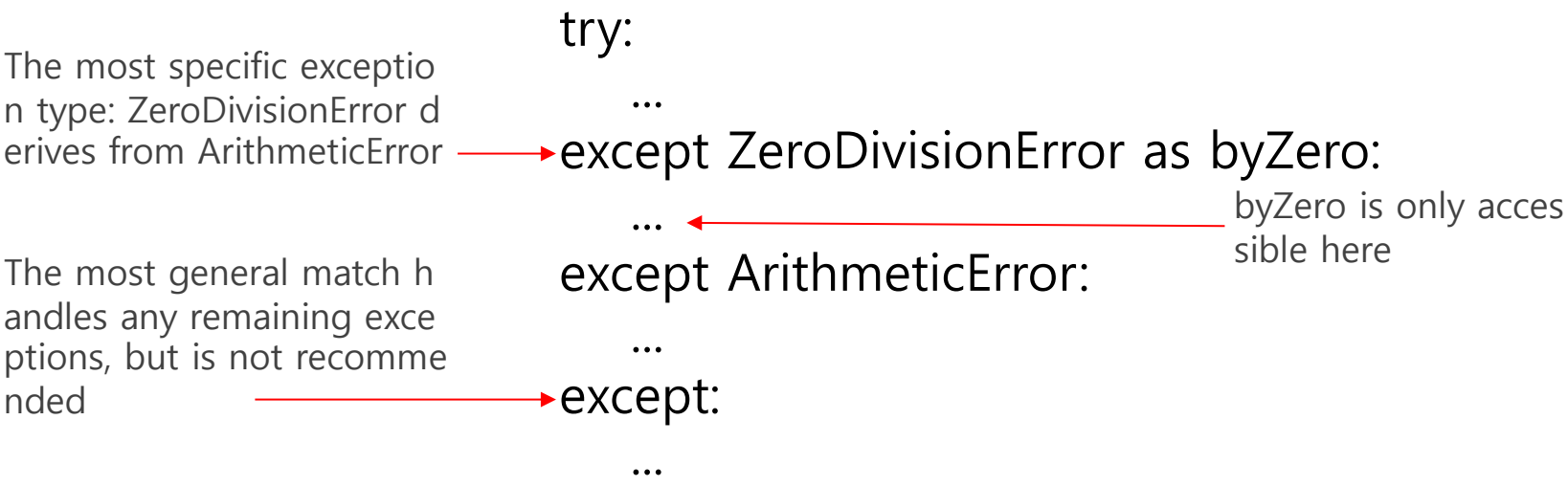
```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
                +-- StopIteration +-- StopAsyncIteration +--
ArithmeticError | +-- FloatingPointError | +-- OverflowError | +--
ZeroDivisionError +-- AssertionError +-- AttributeError +-- BufferError
+-- EOFError +-- 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 | +-- NotImplementedError | +--
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



■ **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
```

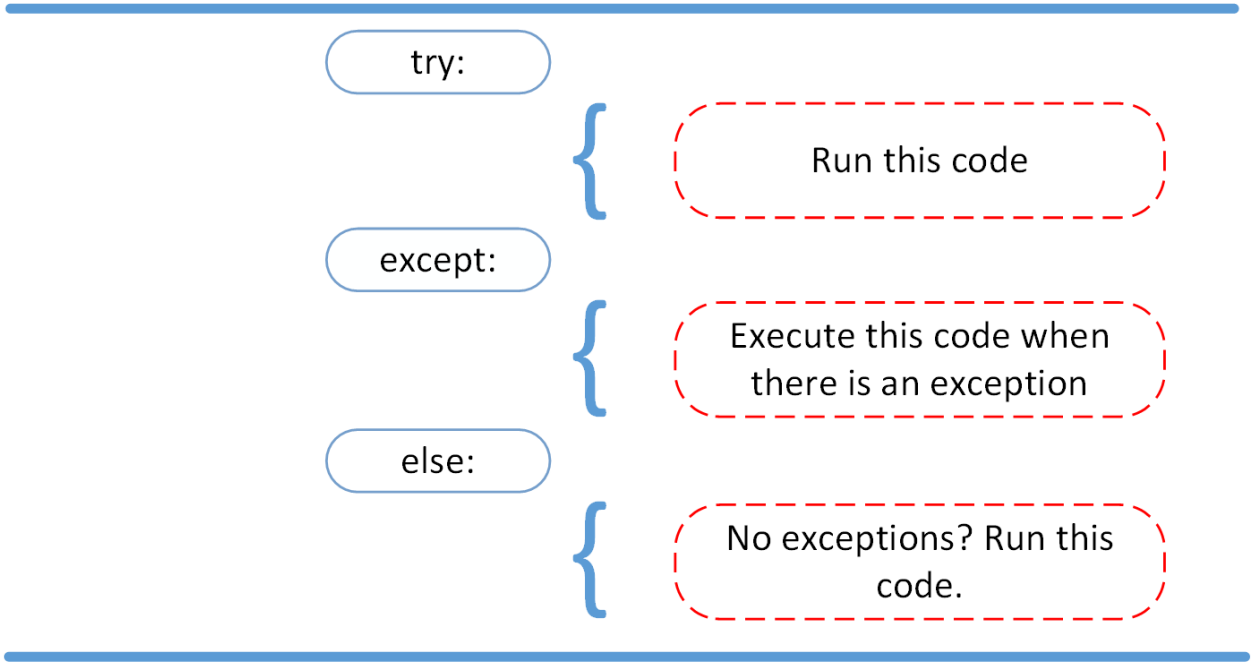
Re-raise current handled exception

Raise a new exception chained from an existing exception named caughtException

Raise a new exception ignoring any previous exception

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

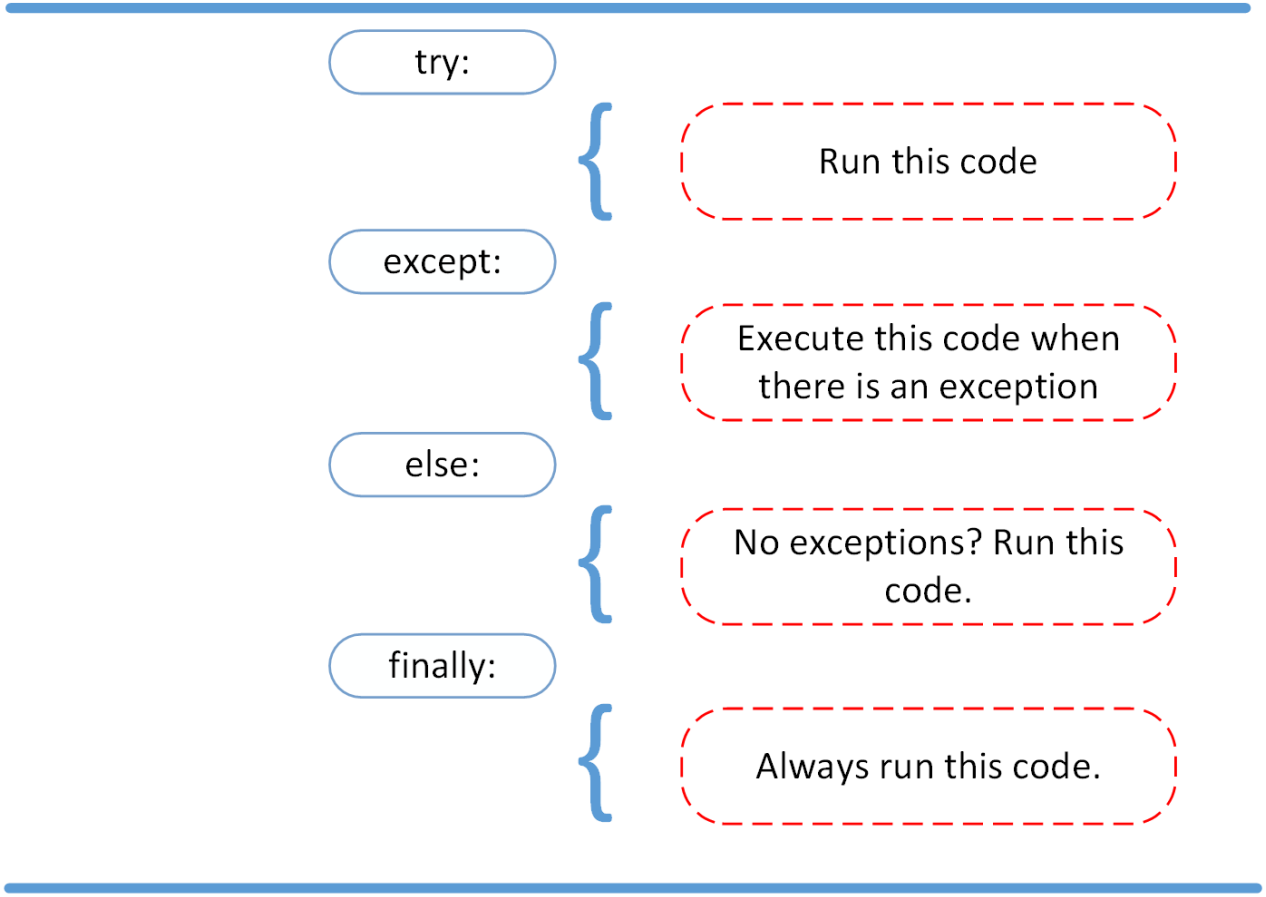
○ This is executed after the try if and only if no exception was raise




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

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

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




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

Permissible try forms

```
try:
    ...
except:
    ...
```

```
try:
    ...
finally:
    ...
```

One or more except handlers, but no more than a single finally or else

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

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

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

else cannot appear unless 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

Equivalent to...

```
assert value is not None

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 인자 로 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 예 에서 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 explicity
- **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 i n 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 도튜멘테이션 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 힌트 로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수 나 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 526을 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

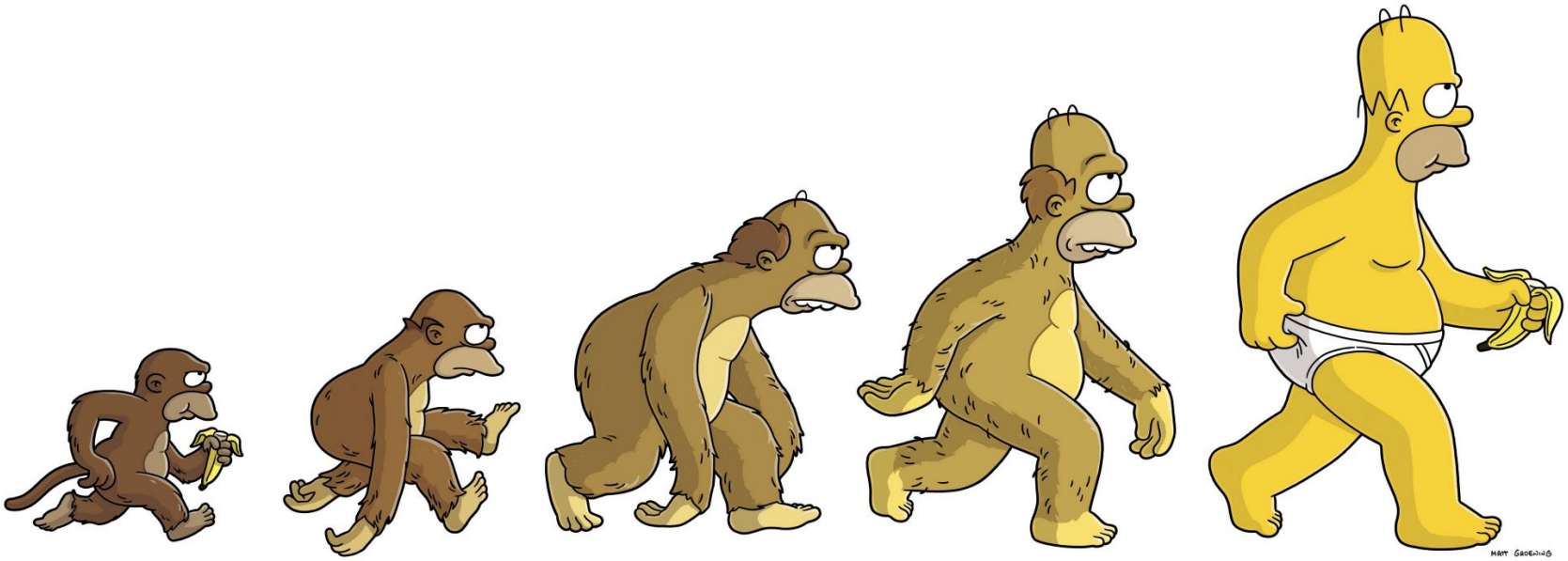
- But defining a one-line named function is preferred over global lamba assignment

```
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


MACHINE
ASSEMBLY
PROCEDURAL
OBJECT ORIENTED
FUNCTIONAL

■ 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 Wikipidia:

○ *"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 *선언됨* 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/…
  - result = func1(a, b) + func2(a, c)We can execute *func1* and *func2* in paralell because a won't be modified.
- Concurrence
  - With no shared data, concurrence 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 this를 첫번째 인자로 가진다.**
- **다만 그것을 self라고 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 go od __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 know 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 <= b
gt(a, b) a > b
ge(a, b) a >= n
is_(a, b) a is b
is_not(a, b) a is not b
contains(a, b) a 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) ~a

In mathematics and computer science, **currying** is the techniq ue of translating the evaluation of a function that takes multi ple arguments (or a tuple of arguments) into evaluating a se quence of functions, each with a single argument (partial app lication). It was introduced by Moses Schönfinkel and later de veloped 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 4 를 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')
```

Force non-positional use of subsequent parameters

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

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

■ Values can be bound to a function's parameters using functools.partial

○ Bound positional and keyword arguments are supplied on calling the resulting callable, other arguments are appended

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

```
print('>')
```

```
quoted('Hello, World!')
```

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

```
quoted('Hello, World!', end=' <₩n')
```

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

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

Default used if iterable is empty

The function to transform the values before determining the lowest one

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

chain(iterable, ...)
compress(iterable, selection)
dropwhile(predicate, iterable)
takewhile(predicate, iterable)
count()
count(start)
count(start, step)
islice(iterable, stop)
islice(iterable, start, stop)
islice(iterable, start, stop, step)
cycle(iterable)
repeat(value)
repeat(value, times)
zip_longest(iterable, ...)
zip_longest(iterable, ..., fillvalue=fill)
...

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}
```

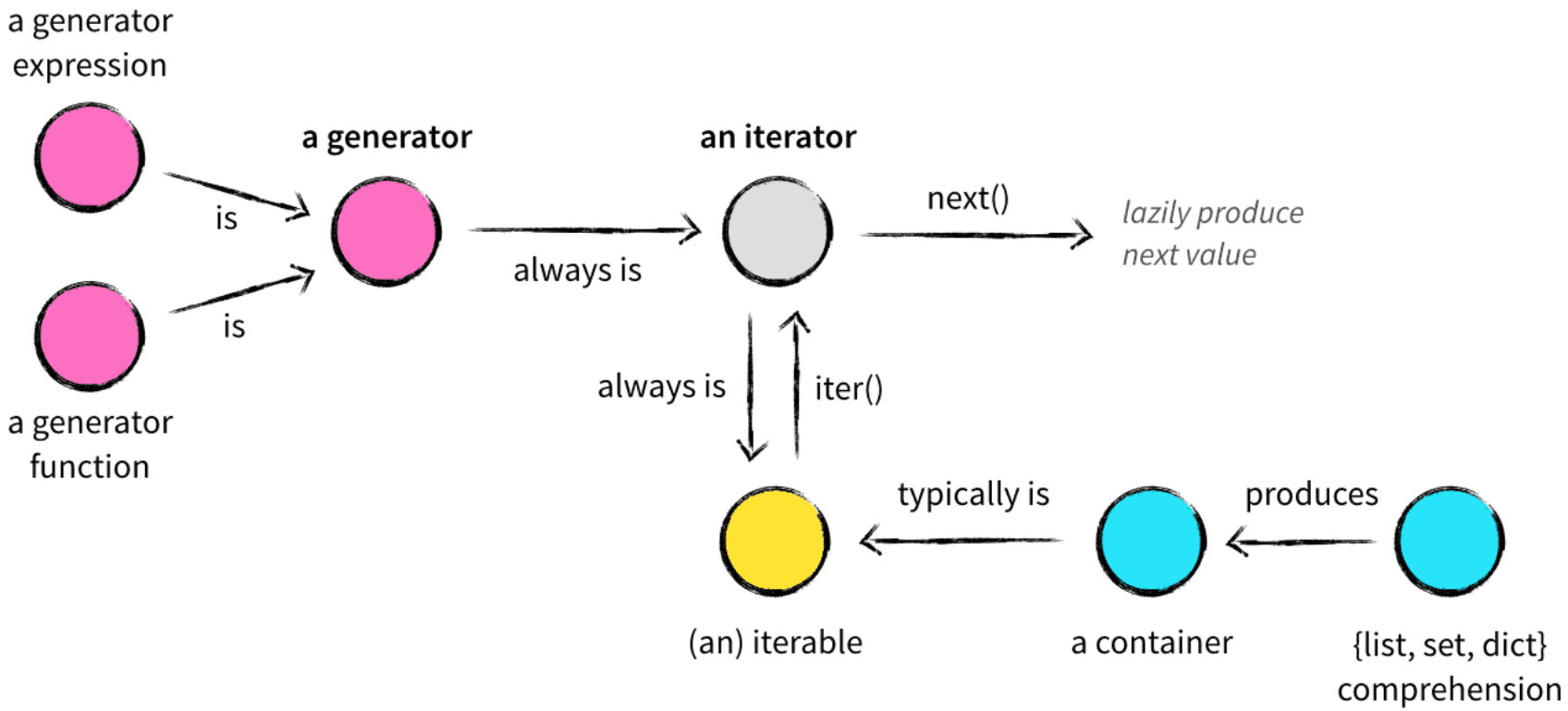
Dictionary comprehension

```
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 elem
ents in the comprehension to appear?

4. iterator & generator


a generator expression
is
a generator
always is
an iterator
next()
lazily produce next value
a generator function
is
always is
iter()
typically is
produces
(an) iterable
a container
{list, set, dict} comprehension

■ 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

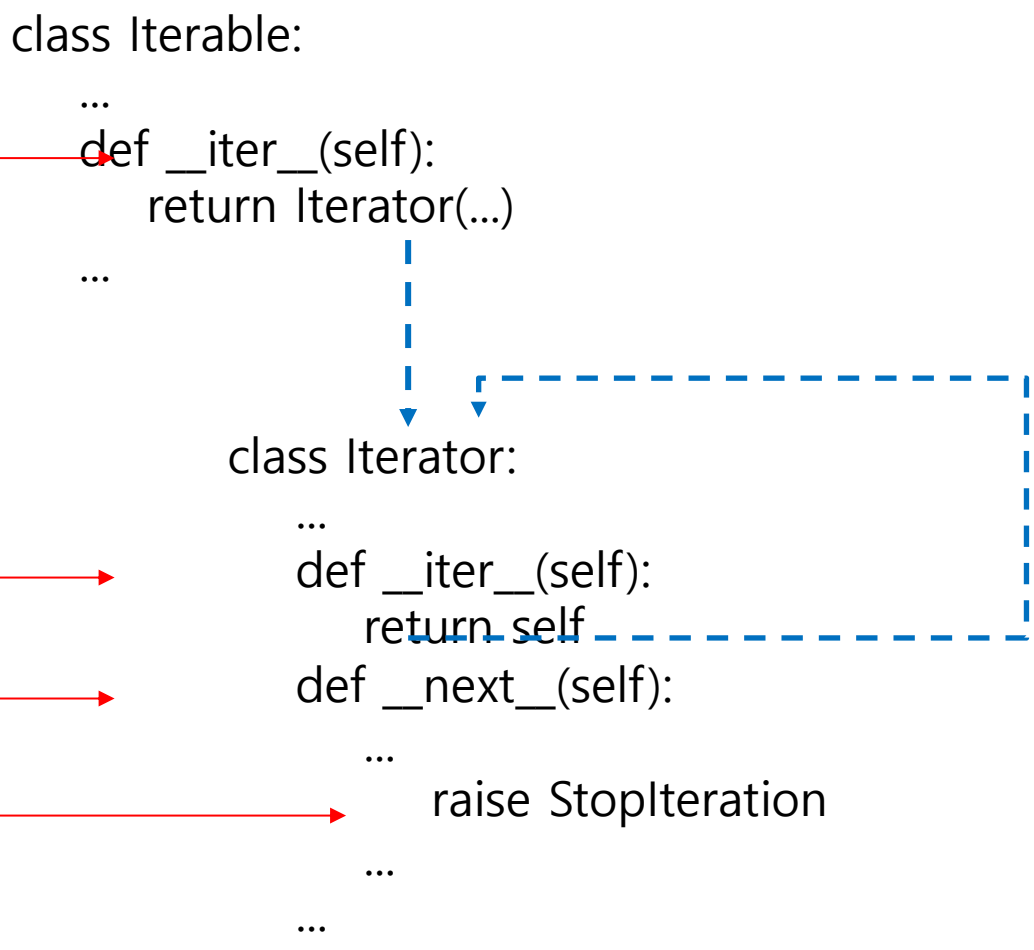
- **An iterator is an object that iterates, following the iterator protocol**
  - An iterable object can be used with for
- **Iterators and generators are lazy, returning values on demand**
  - You don't need to resolve everything into a list in order to use a series of values
  - Yield values in sequence, 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 s upports the __iter__ speci al method, which is called by the iter function

All iterators are iterable, s o 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



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']
```

Builtin Generation Function

- 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]

■ infinte 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

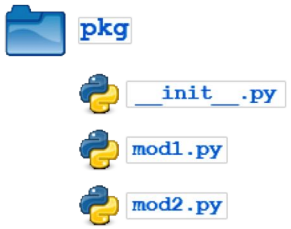
- A package hierarchically organises modules and other packages
- Where a module corresponds to a file, a regular package corresponds to a directory
- Modules and packages define global namespaces
- Extension modules allow extension of Python itself

■ 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	Imported
import sys	sys.stdin, ...
import sys as system	system.stdin, ...
from sys import stdin	stdin
from sys import stdin, stdout	stdin, stdout
from sys import stdin as source	source
from sys import *	stdin, ...

```
from mod import s as string, a as alist
```

Discouraged

This will place the names of *all* objects from <module_name> in the local symbol table, with the exception of any that begin with an 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

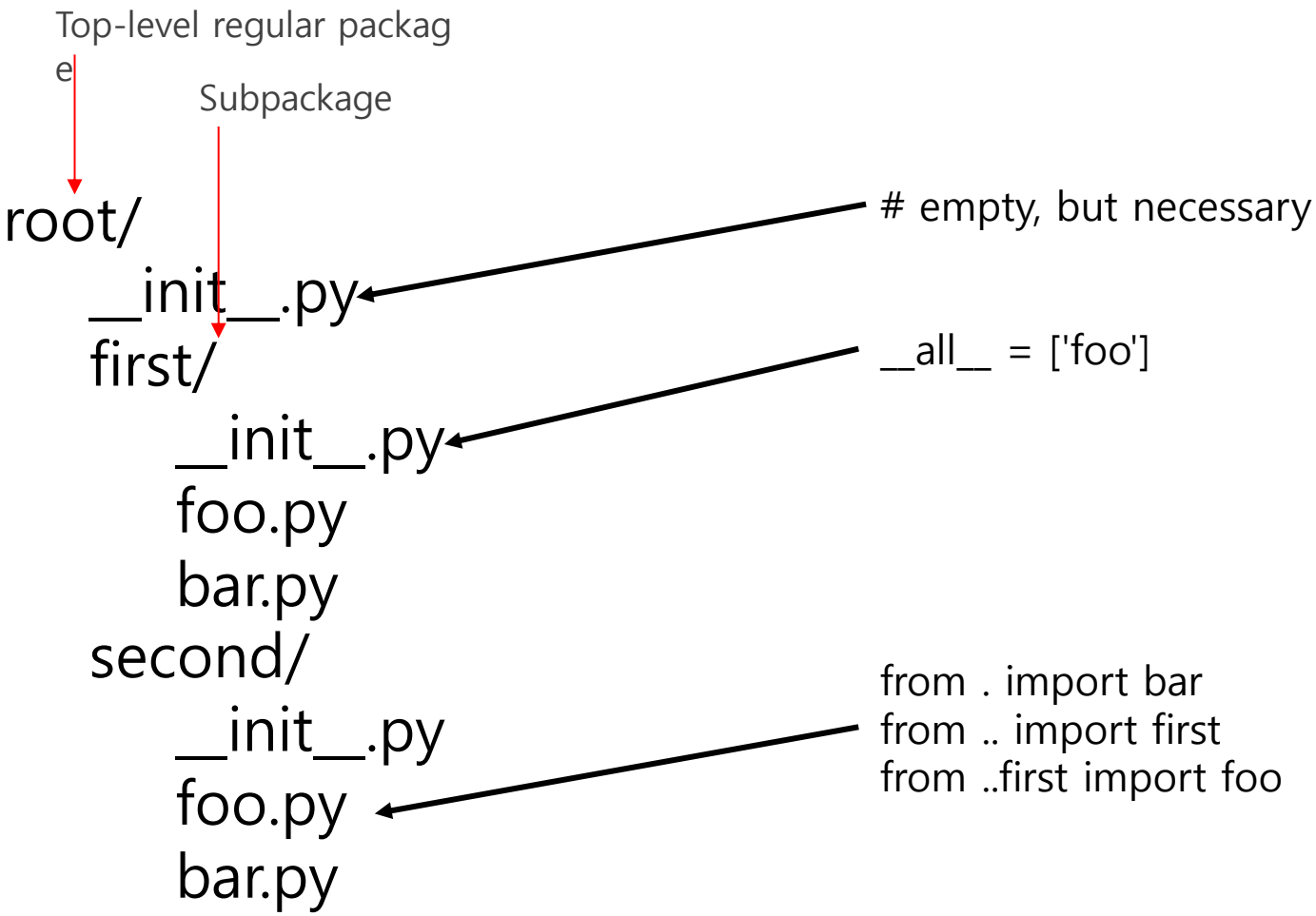
```
...
def main():
    ...
if __name__ == '__main__':
    main()
```

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


Top-level regular packag
e
Subpackage
root/
__init__.py
first/
__init__.py
foo.py
bar.py
second/
__init__.py
foo.py
bar.py
# empty, but necessary
__all__ = ['foo']
from . import bar
from .. import first
from ..first import foo

- 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)

Experiment

- Write a simple script that lists the pack age and module hierarchy
  - Use current directory if no path supplied
  - Use *os.listdir*, *os.walk* or *pathlib* for travers al
  - No need to open and load modules, just detect packages by matching *__init__.py* a nd 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 importstatement 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, whist __import__ will use a C-optimised version. Both versions call back to importlib._bootstrap._find_and_load() so the difference is mostly academic.

ttps://stackoverflow.com/questions/28231738/import-vs-import-vs-importlib-import-mod