Fluent Python

Lorenzo Vecchietti

January 2024

Contents

Python Data Model

Il Python Data Model è fondamentale per sfruttare appieno le potenzialità di Python. Conoscere i metodi speciali consente di personalizzare il comportamento delle classi, mantenendo familiarità con la sintassi e le funzionalità del linguaggio.

Ad esempio, implementare il metodo `__getitem__` in una classe permette di: - Abilitare operazioni come lo slicing. - Rendere la classe iterabile.

> **Nota:** I metodi speciali sono chiamati implicitamente dall'interprete Python e non devono essere invocati direttamente dall'utente. Ad esempio, l'istruzione `for i in x:` attiva implicitamente `iter(x)`, che a sua volta chiama `x.__iter__()` o `x.__getitem__()`.

Utilizzi comuni del Python Data Model

1. **Rendere una classe iterabile**
   - Implementando metodi come `__len__` e `__getitem__`.
2. **Rendere una classe numerica**
   - Utilizzando metodi come `__add__`, `__mul__`, `__abs__`, ecc.
3. **Rappresentazione della classe**
   - `__repr__`: Deve essere univoco e non ambiguo.
   - `__str__`: Deve essere leggibile (es. `str(3)` è uguale a `str("3")`).
   - Se `__repr__` non è implementato, `str()` utilizzerà `repr()` come fallback.
4. **Valutazione booleana**
   - Per impostazione predefinita, le istanze delle classi personalizzate sono considerate "truthy", a meno che non siano implementati `__bool__` o `__len__`.
   - `bool(x)` chiama `x.__bool__()` se disponibile; altrimenti, invoca `x.__len__()`. Se `__len__` restituisce zero, `bool(x)` restituisce `False`.

Categorie di metodi speciali

Metodi comuni

Categoria	Metodi
String/bytes representation	`__repr__`, `__str__`, `__format__`, `__bytes__`, `__fspath__`
Conversione a numero	`__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`, `__index__`
Emulazione di collezioni	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
Iterazione	`__iter__`, `__aiter__`, `__next__`, `__anext__`, `__reversed__`
Esecuzione di funzioni	`__call__`, `__await__`
Gestione del contesto	`__enter__`, `__exit__`, `__aenter__`, `__aexit__`
Creazione/distruzione istanze	`__new__`, `__init__`, `__del__`
Gestione attributi	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`
Descriptor di attributi	`__get__`, `__set__`, `__delete__`, `__set_name__`
Classi astratte	`__instancecheck__`, `__subclasscheck__`
Metaprogrammazione	`__prepare__`, `__init_subclass__`, `__class_getitem__`, `__mro_entries__`

Operatori

Categoria operatore	Simboli	Metodi
Numerico unario	`-`, `+`, `abs()`	`__neg__`, `__pos__`, `__abs__`
Confronto	`<`, `<=`, `==`, `!=`, `>`, `>=`	`__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`
Aritmetica	`+`, `-`, `*`, `/`, `//`, `%`, `**`	`__add__`, `__sub__`, `__mul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__pow__`

Categoria operatore	Simboli	Metodi
Aritmetica invertita	Operatori con operandi invertiti	`__radd__`, `__rsub__`, `__rmul__`, `__rtruediv__`, ecc.
Assegnazione aumentata	`+=`, `-=`, `*=`, `/=`, `//=`, ecc.	`__iadd__`, `__isub__`, `__imul__`, `__itruediv__`, ecc.
Bitwise	`&`, `|`, `^`, `<<`, `>>`, `~`	`__and__`, `__or__`, `__xor__`, `__lshift__`, `__rshift__`, `__invert__`
Bitwise invertito	Operatori con operandi invertiti	`__rand__`, `__ror__`, `__rxor__`, `__rlshift__`, ecc.
Assegnazione aumentata bitwise	`&=`, `|=`, `^=`, `<<=`, `>>=`	`__iand__`, `__ior__`, `__ixor__`, `__ilshift__`, ecc.

Curiosità: `len(x)` è estremamente veloce per i tipi predefiniti in Python. Nel caso degli oggetti integrati di CPython, la lunghezza è letta direttamente da un campo in una struttura C, senza invocare alcun metodo.

Array of Sequences

Gestione uniforme delle sequenze. Stringhe, liste, sequenze di byte, array, elementi XML e risultati di database condividono un ricco set di operazioni comuni, tra cui iterazione, slicing, ordinamento e concatenazione.

Ci sono due tipi principali di sequenze:

- **Uniformi**: contengono solo un tipo di oggetto, come `str`, `bytes` e `array.array`.
- **Contenitori**: possono contenere oggetti di tipi differenti, come `list`, `tuple` e `collections.deque`. Ogni oggetto ha un tipo associato.

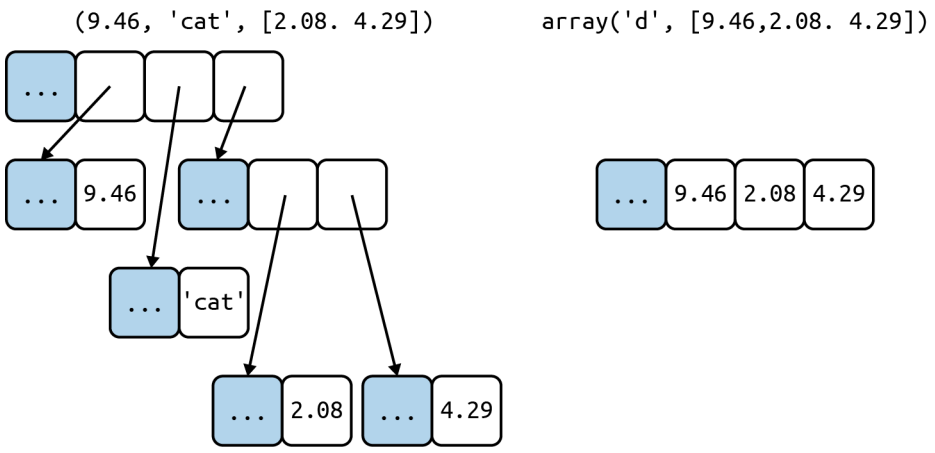

Figure 1: Allocazione memoria per container vs array

Ogni oggetto ha un header. Ad esempio, un semplice `float` contiene:

- `ob_refcnt`: il contatore di riferimento dell'oggetto.
- `ob_type`: un puntatore al tipo dell'oggetto.
- `ob_fval`: un valore `double` in C che rappresenta il valore del `float`.

Un altro modo per classificare le sequenze è in base alla mutabilità:

- **Sequenze mutabili**: ad esempio, `list`, `bytearray`, `array.array` e `collections.deque`.
- **Sequenze immutabili**: ad esempio, `tuple`, `str` e `bytes`.

Tuples vs Lists

Le tuple non sono semplicemente liste immutabili.

Le tuple rappresentano record: ogni elemento nella tupla rappresenta un campo e la posizione dell'elemento ne definisce il significato.

- **Chiarezza**: Una tupla nel codice indica che la sua lunghezza non cambierà mai.
- **Performance**: Una tupla utilizza meno memoria rispetto a una lista di pari lunghezza e consente a Python di ottimizzare alcune operazioni.

L'immutabilità può essere compromessa se una tupla contiene una lista:

```
a = (1, 2, 3, (4, 5))
b = (1, 2, 3, [4, 5])
hash(a)  # Funziona
hash(b)  # Solleva TypeError
```

List Comprehension vs Generator Expressions

Per inizializzare tuple, array e altri tipi di sequenze, si può utilizzare una list comprehension, ma una generator expression (genexp) risparmia memoria perché produce elementi uno alla volta utilizzando il protocollo iteratore, invece di creare un'intera lista da passare a un costruttore.

Le genexp utilizzano la stessa sintassi delle list comprehension, ma sono racchiuse tra parentesi tonde invece che tra parentesi quadre.

Unpacking con *

L'operatore `*` consente di "spacchettare" una sequenza in più variabili:

```
a, b, *rest = range(5)
# 0, 1, [2, 3, 4]

def fun(a, b, c, d, *rest):
    return a, b, c, d, rest

fun(*[1, 2], 3, *range(4, 7))
# 1, 2, 3, 4, (5, 6)
```

Slicing

Tutti i tipi di sequenze in Python supportano le operazioni di slicing. `s[a:b:c]` consente di specificare uno stride o passo `c`.

Slicing in NumPy

Le sequenze incorporate in Python sono unidimensionali, ma con `numpy.ndarray` si possono utilizzare sintassi come `a[i, j]` per accedere a elementi e `a[m:n, k:l]` per ottenere uno slice bidimensionale.

NumPy utilizza `...` come scorciatoia per il slicing di array multidimensionali. Ad esempio, se `x` è un array a quattro dimensioni, `x[i, ...]` è equivalente a `x[i, :, :, :]`.

List vs Array

Un array Python è snello come un array in C. Un `array` di valori `float` non contiene istanze `float` complete, ma solo i byte che rappresentano i loro valori macchina, in modo simile a un array di `double` in C.

```
from array import array
from random import random

floats = array('d', (random() for _ in range(100)))
fp = open('floats.bin', 'wb')
floats.tofile(fp)
fp.close()

floats2 = array('d')
fp = open('floats.bin', 'rb')
floats2.fromfile(fp, 100)  # Legge 100 double
```

List vs Deque

I metodi `.append` e `.pop` rendono le liste utilizzabili come stack o code. Tuttavia, inserire e rimuovere elementi dalla testa di una lista (indice 0) è costoso perché l'intera lista deve essere spostata in memoria.

La classe `collections.deque` è una coda a doppia estremità thread-safe progettata per inserimenti e rimozioni rapidi da entrambe le estremità. È utile per mantenere una lista di elementi "visti per ultimi" o simili, perché un `deque` può avere una lunghezza massima fissa. Se un `deque` è pieno, quando si aggiunge un nuovo elemento, ne scarta uno dall'estremità opposta.

Summary of sequences types

Operation	List	Tuple	Array	Deque	Description
`s.__add__(s2)`	•	•	•		`s + s2`—concatenation
`s.__iadd__(s2)`	•		•	•	`s += s2`—in-place concatenation
`s.append(e)`	•		•	•	Append one element to the right (after last)
`s.appendleft(e)`				•	Append one element to the left (before first)
`s.byteswap()`			•		Swap bytes of all items in array for endianness conversion
`s.clear()`	•			•	Delete all items
`s.__contains__(e)`	•	•	•		`e in s`
`s.copy()`	•				Shallow copy of the list
`s.__copy__()`			•	•	Support for `copy.copy` (shallow copy)
`s.count(e)`	•	•	•	•	Count occurrences of an element
`s.__deepcopy__()`			•		Optimized support for `copy.deepcopy`
`s.__delitem__(p)`	•		•	•	Remove item at position `p`
`s.extend(it)`	•		•	•	Append items from iterable `it` to the right
`s.extendleft(it)`				•	Append items from iterable `it` to the left
`s.frombytes(b)`			•		Append items from byte sequence interpreted as packed machine values
`s.fromfile(f, n)`			•		Append `n` items from binary file `f` interpreted as packed machine values
`s.fromlist(l)`			•		Append items from list; if one causes `TypeError`, none are appended
`s.__getitem__(p)`	•	•	•	•	`s[p]`—get item or slice at position
`s.__getnewargs__()`		•			Support for optimized serialization with `pickle`
`s.index(e)`	•	•	•		Find position of first occurrence of `e`

Operation	List	Tuple	Array	Deque	Description
`s.insert(p, e)`	•		•		Insert element `e` before the item at position `p`
`s.itemsize`			•		Length in bytes of each array item
`s.__iter__()`	•	•	•	•	Get iterator
`s.__len__()`	•	•	•	•	`len(s)`—number of items
`s.__mul__(n)`	•	•	•		`s * n`—repeated concatenation
`s.__imul__(n)`	•		•		`s *= n`—in-place repeated concatenation
`s.__rmul__(n)`	•	•	•		`n * s`—reversed repeated concatenation
`s.pop([p])`	•		•	•	Remove and return last item or item at optional position `p`
`s.popleft()`				•	Remove and return first item
`s.remove(e)`	•		•	•	Remove first occurrence of element `e` by value
`s.reverse()`	•		•	•	Reverse the order of the items in place
`s.__reversed__()`	•			•	Get iterator to scan items from last to first
`s.rotate(n)`				•	Move `n` items from one end to the other
`s.__setitem__(p, e)`	•		•	•	`s[p] = e`—put `e` in position `p`, overwriting existing item or slice
`s.sort([key], [reverse])`	•				Sort items in place with optional keyword arguments `key` and `reverse`
`s.tobytes()`			•		Return items as packed machine values in a `bytes` object
`s.tofile(f)`			•		Save items as packed machine values to binary file `f`
`s.tolist()`			•		Return items as numeric objects in a `list`
`s.typecode`			•		One-character string identifying the C type of the items

Dizionari in Python

I dizionari sono un oggetto fondamentale in Python. Molte classi aggiuntive si basano su di essi. Sono oggetti molto efficienti, costruiti su *hash tables*, così come i `set`.

Unpacking Mapping

È possibile utilizzare l'operatore `**` per "spacchettare" un mapping:

```
def dump(**kwargs):
    return kwargs

dump(**{'x': 1}, y=2, **{'z': 3})
# Output: {'x': 1, 'y': 2, 'z': 3}
```

Un altro esempio:

```
{'a': 0, **{'x': 1}, 'y': 2, **{'z': 3, 'x': 4}}
# Output: {'a': 0, 'x': 4, 'y': 2, 'z': 3}
```

In caso di chiavi duplicate, le occorrenze successive sovrascrivono quelle precedenti (vedi il valore della chiave `x`).

Unione dei Mapping (Python 3.9+)

Per unire i mapping, si può utilizzare:

- `|` per creare un nuovo dizionario
- `|=` per modificare il dizionario in-place.

API Mapping Types

Il modulo `collections.abc` fornisce le interfacce `Mapping` e `MutableMapping`, utili per implementare tipi simili a `dict`.
Per creare un dizionario personalizzato, è consigliabile utilizzare `collections.UserDict`.

Perché UserDict?
`UserDict` utilizza un dizionario interno (`self.data`), che consente di sovrascrivere metodi come `__setitem__` senza rischio di incorrere in errori di ricorsione.

Dizionari Default

Un oggetto è **hashable** se:
1. Ha un codice hash immutabile durante la sua vita (`__hash__()`). 2. Può essere confrontato con altri oggetti (`__eq__()`).

Esempio di utilizzo di `setdefault`:

```
my_dict.setdefault(key, []).append(new_value)
```

Equivalente a:

```
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value)
```

`setdefault` effettua una sola ricerca della chiave, rendendolo più efficiente.

Il Metodo `__missing__`

Se subclassiamo `dict` e definiamo un metodo `__missing__`, questo viene chiamato ogni volta che una chiave non viene trovata, invece di sollevare un `KeyError`.

Altre Mapping Utility

- **`collections.ChainMap`**
  Consente di concatenare più mapping e cercare chiavi in ordine:

```
from collections import ChainMap
d1 = {'a': 1, 'b': 3}
d2 = {'a': 2, 'b': 4, 'c': 6}
chain = ChainMap(d1, d2)

chain['a']  # Output: 1
chain['c']  # Output: 6
```

- **`collections.Counter`**
  Per contare elementi in una sequenza:

```
from collections import Counter
ct = Counter('abracadabra')
ct.update('aaaaazzz')
```

- **`shelve.Shelf`**
  Per la persistenza di dati.

Dizionari Immutabili

Per creare dizionari immutabili, usa `MappingProxyType`:

```
from types import MappingProxyType

d = {'key': 'value'}  # Dizionario mutabile
d_proxy = MappingProxyType(d)  # Dizionario immutabile
```

Se `d` viene aggiornato, anche `d_proxy` riflette l'aggiornamento. Tentare di modificare `d_proxy` genera un `TypeError`.

Sets

I `set` sono collezioni di oggetti unici introdotte in Python 2.3. Un `set` è mutabile, mentre `frozenset` è immutabile.

Caratteristiche

- Gli elementi devono essere **hashable**.
- Operazioni di appartenenza (`in`) e unione/intersezione sono molto efficienti.
- L'ordine degli elementi dipende dall'ordine di inserimento, ma non è affidabile.

Operazioni sui Set

Simbolo matematico	Operatore Python	Metodo	Descrizione
$S \cap Z$	`s & z`	`s.__and__(z)`	Intersezione
	`s &= z`	`s.__iand__(z)`	Aggiorna `s` con l'intersezione con `z`
$S \cup Z$	`s | z`	`s.__or__(z)`	Unione
	`s |= z`	`s.__ior__(z)`	Aggiorna `s` con l'unione con `z`
$S \setminus Z$	`s - z`	`s.__sub__(z)`	Differenza
	`s -= z`	`s.__isub__(z)`	Aggiorna `s` con la differenza con `z`
$S \Delta Z$	`s ^ z`	`s.__xor__(z)`	Differenza simmetrica
	`s ^= z`	`s.__ixor__(z)`	Aggiorna `s` con la differenza simmetrica con `z`

Relazioni sui Set

Simbolo matematico	Metodo	Descrizione
$S \cap Z = \emptyset$	`s.isdisjoint(z)`	Verifica che `s` e `z` siano disgiunti
$e \in S$	`e in s`	Verifica che `e` sia in `s`
$S \subseteq Z$	`s <= z`	Verifica che `s` sia sottoinsieme di `z`
$S \supseteq Z$	`s >= z`	Verifica che `s` sia sovrainsieme di `z`
$S \subset Z$	`s < z`	Verifica che `s` sia un sottoinsieme proprio
$S \supset Z$	`s > z`	Verifica che `s` sia un sovrainsieme proprio

Stringhe e Byte

Una stringa è una sequenza di caratteri, un carattere è un numero compreso fra 0 e 1114111 che nello standard Unicode è rappresentato da 'U+' e da 4 a 6 hex digits. I caratteri Unicode vengono rappresentati in byte con vari encoding (il più comune è UTF-8, default in python). In python si usa `s.encode('utf8')` e `b.decode('utf8')`

Se si decodifica una sequenza di byte, bisogna conoscere a priori la codifica usata. Talvolta questa informazione è contenuta nell'header. Ci sono però alcuni metodi di fare delle ipotesi: il modulo `chardet` implementa varie strategie per fare queste ipotesi.

Little endian vs Big endian

Quando una stringa viene codificata da UTF-16, ci sono due extra byte all'inizio del `bytearray`: è il BOM, *byte order mark*, che indica se l'encoding è stato fatto su una CPU little endian o big endian. Ciò viene fatto con il carattere U+FEFF, poichè non esiste U+FFFE, se il sistema legge b`\xff\xfe` allora è chiaro che la codifica è little endian. Ci sono anche le varianti UTF-16LE e UTF-16BE, in questi casi non viene gerato il BOM.

Best Practices

1. Mai fare encoding e decoding a metà del programma. Bisogna cercare di lavorare sempre con le stringhe. Bisogna quindi usare all'inizio `my_file.read()` e alla fine `my_file.write(text)` (output e argomento sono `str`).
2. Quando si fanno confronti tra stringhe bisognerebbe usare sempre la stessa cifratura. Si può usare `unicodedata.normalize()`.
3. Oltre che usare lo stesso encoding, si può sfruttare la funzione casefolding che rende lowercase tutti i caratteri ma con alcune trasformazioni aggiuntive (ad esempio `'ß'` diventa `'ss'`).
4. Il modo standard per ordinare il testo non ASCII in Python è usare la funzione locale.strxfrm che, secondo la documentazione del modulo locale, "trasforma una stringa in una che può essere utilizzata per confronti in base alle impostazioni locali".

Costruire Data Class

Si possono costruire delle dataclass in diversi modi:

1. `collections.namedtuple`
2. `typing.NamedTuple`, namedtuple con type hint
3. `@dataclasses.dataclass`, class decorator

`namedtuple`

Si costruisce con nome della classe + nome dei dati della classe.

```
from collections import namedtuple

Coordinate = namedtuple('Coordinate', 'lat lon')
moscow = Coordinate(55.756, 37.617)
```

`moscow` viene rappresentato con `Coordinate(lat=55.756, lon=37.617)`

`typing.NamedTuple`

E' una variazione di `namedtuple` in cui si può specificare il tipo:

```
import typing
Coordinate = typing.NamedTuple('Coordinate', lat=float, lon=float)
```

Un utilizzo alternativo è in una classe:

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float
    lon: float
```

`dataclass` decorator

La sintassi è analoga a quella dell'opzione precedente.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Coordinate:
    lat: float
    lon: float
```

Mutabilità

Attraverso l'argomento `frozen=True` si rende la classe immutabile, mentre di default tutte le classi ottenute dalle tre opzioni sono mutabili.

Quando si usano parametri di default non si dovrebbero utilizzare oggetti mutabili come liste. Quindi, ad esempio, al posto che

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
    guests: list = field(default_factory=list)
```

si deve usare

```python
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
    guests: list[str] = field(default_factory=list)
```

Data Classes e Code Smell

L'utilizzo estensivo di classi di dati può essere sintomo di un codice non ottimale: *"These are classes that have fields, getting and setting methods for fields, and nothing else. Such classes are dumb data holders and are often being manipulated in far too much detail by other classes."*

Oggetti, reference e mutabilità

Variabili

Le variabili devono essere interpretate come delle etichette sugli oggetti. Più etichette possono essere legate allo stesso oggetto. Ciò significa che con un oggetto mutabile (ad esempio una lista) associato a due variabili, modificando l'oggetto attraverso una variabile, si modifica anche l'altra variabile.

```
a = [1, 2, 3]
b = a
a.append(4)
print(b) # stampa [1, 2, 3, 4]
```

Ogni oggetto ha un contatore che indica quante variabili sono collegate al singolo oggetto. Quando le variabili collegate all'oggetto diventano zero, l'oggetto viene "cancellato" (algoritmo garbage collector). Si può usare lo statement `del`, ma questo non cancella l'oggetto, cancella il riferimento ad esso. In conseguenza di questo, se l'oggetto rimane senza riferimenti, anche l'oggetto viene cancellato.

Identità, Uguaglianze e Aliases

Identità e uguaglianza sono concetti diversi. Per verificare che due alias siano riferiti allo stesso oggetto, si usa `is`. Per verificare che due oggetti diversi siano uguali, si usa `==`.

```
charles = {'name': 'Charles L. Dodgson', 'born': 1832}
alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}
alex == charles  # True
alex is charles  # False
```

Quindi l'operatore `is` compara l'intero risultato in output della funzione `id()`. D'altra parte `==` chiama la funzione `__eq__`. Molte classi hanno una implementazione propria di `__eq__` ma il metodo `__eq__` di default, ereditato da `object`, compara `id()`. Quindi a meno che l'oggetto comparato non implementi un metodo `__eq__`, `is` e `==` sono equivalenti.

Deep copy vs shallow copy

Il metodo più semplice di copiare una lista è quello di fare una *shallow copy*. Per farlo può usare `list(l1)` oppure `l1[:]`. Se però `l1` contiene degli oggetti mutabili, allora solo il riferimento allo stesso oggetto viene copiato. Se quindi agisco sull'oggetto mutabile, le modifiche saranno visibili sia in `l1` che nella sua copia:

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)
l1.append(100)
l1[1].remove(55)
print('l1:', l1)  # l1: [3, [66, 44], (7, 8, 9), 100]
print('l2:', l2)  # l2: [3, [66, 44], (7, 8, 9)]
```

Si può utilizzare il modulo `copy` che permette di usare il metodo `deepcopy`.

Oggetti mutabili in funzioni e classi

L'utilizzo di argomenti o attributi mutabili in funzioni e classi è tipicamente fonte di bug. Ogni volta che si passa una lista come argomento o attributo, tutti i metodi e funzioni modificano la lista originale.

Funzioni

```
def f(a, b):
    a += b
    return a

a = [1, 2]
b = [3, 4]
```

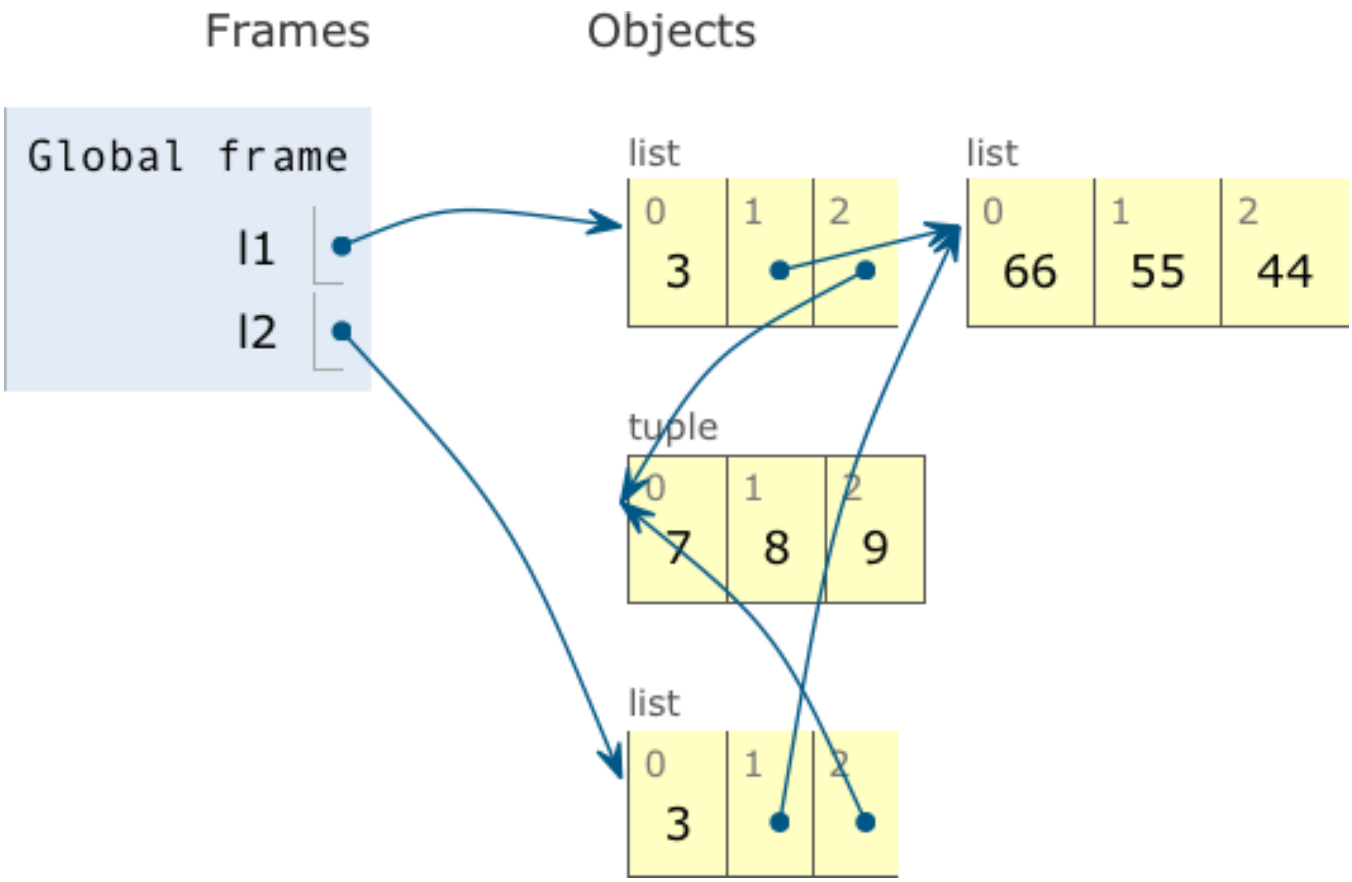

Figure 2: Comportamento in caso di *shallow copy*

```
f(a, b)
# [1, 2, 3, 4]
a, b
# ([1, 2, 3, 4], [3, 4])
```

`f` modifica direttamente la lista di input `a`.

Classi

```
class HauntedBus:
    """A bus model haunted by ghost passengers"""

    def __init__(self, passengers=[]):
        self.passengers = passengers

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)


bus1 = HauntedBus()
bus1.pick('Carrie')
bus1.passengers
# ['Carrie']
bus2 = HauntedBus()
bus2.passengers
# ['Carrie']
bus2.pick('Dave')
bus1.passengers
# ['Carrie', 'Dave']
bus2.passengers is bus3.passengers
# True
```

In questo caso la lista vuota viene definita una volta sola, in corrispondenza della definizione della classe. Quando viene utilizzato il costruttore `HauntedBus()` senza argomenti, viene utilizzato il riferimento alla stessa lista ogni volta. Per evitare questi problemi bisogna: 1. non utilizzare come argomenti di default argomenti mutabili 2. fare almeno una *shallow copy* quando l'input atteso è una lista

```
class Bus:
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)
```

Funzioni come *first-class objects*

In un linguaggio di programmazione gli oggetti *first-class* hanno le seguenti caratteristiche: 1. possono essere create durante il *runtime* 2. possono essere assegnate a una variabile/attributo 3. possono essere passate come argomento ad una funzione 4. possono essere usate come risultato di una funzione

Esempio:

```
def factorial(n):
     """returns n!"""
     return 1 if n < 2 else n * factorial(n - 1)

fact = factorial
fact
#<function factorial at 0x...>
fact(5)
#120
map(factorial, range(11))
# <map object at 0x...>
list(map(factorial, range(11)))
# [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Una funzione come `map`, che prende in input altre funzioni, è detta *high-order function*.

Funzioni anonime

La *keyword* `lambda` permette di creare funzioni anonime. Queste devono essere espressioni (non contenere `while`, `try`, `=`,...) Tipicamente le funzioni `lambda` vengono usate nelle *high-order* function. Al di fuori di queste casistiche non si dovrebbero usare per ragioni di leggibilità.

Callable definiti dall'utente

Si può implementare la funzione `__call__` nelle classi per rendere un oggetto *callable* con `()`.

```
import random

class BingoCage:

    def __init__(self, items):
        self._items = list(items)
        random.shuffle(self._items)

    def pick(self):
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self):
        return self.pick()

bingo = BingoCage(range(3))
bingo.pick()
# 1
bingo()
# 0
callable(bingo)
# True
```

Programmazione funzionale

Modulo *operator*

```
from functools import reduce
from operator import mul

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

In questo esempio usiamo: 1. `mul` per evitare di scrivere la funzione triviale `lambda a,b: a*b`. 2. `reduce`, prende un elemento della *collection* (in questo caso `range(1, n+1)`, il risultato precedente della funzione (in questo caso `mul`) e li passa come argomenti alla funzione stessa.

Altre funzioni utili di `operator` sono: `itemgetter` e `attrgetter` (molto utilizzati come argomento `key` in `sorted`).

Modulo `functools`

Si può usare `partial` per fissare un parametro di una funzione più generica.

Types

Python è una lingua *dynamically typed*. Si può utilizzare però `mypy` come tool per rendere python *gradually typed*, ovvero in cui:

1. Assegnare *types* è opzionale
2. Non vengono segnalati errori relativi ai *types* durante il *runtime*
3. I *types* non migliorano le performance

```
from typing import Optional

def show_count(count: int, singular: str, plural: Optional[str] = None) -> str:
    ...
```

Cos'è un *type*

Un *type* è definito dalle operazioni che l'oggetto supporta.

Ciò è definito *duck typing*: l'unica cosa che importa è se durante il *runtime* l'operazione che si cerca di eseguire è definita sull'oggetto.

> *"If I can invoke birdie.quack(), then birdie is a duck in this context."*

Con `mypy`, c'è maggior rigidità: la cosa importante è il *nominal typing*

```
class Bird:
    pass

class Duck(Bird):
    def quack(self):
        print('Quack!')

def alert(birdie):
    birdie.quack()

def alert_duck(birdie: Duck) -> None:
    birdie.quack()

def alert_bird(birdie: Bird) -> None:
    birdie.quack()
```

L'ultima funzione evidenzierà un errore quando controllata da `mypy` dato che `quack` non è definito da `Bird`. Se però definissi `daffy = Duck()`, sia `alert_duck(daffy)` che `alert_bird(daffy)` funzioneranno durante il *runtime*.

Quali *types* sono supportati?

`typing.Any`

Questo è il tipo più generale. `mypy` considererà che gli oggetti annotati con `Any` supportino qualsiasi operazione (cosa chiaramente impossibile da garantire). Si scoprirà durante il runtime se tutto funziona o meno.

Un concetto importante è quello di `consistent-with`. Dato T1 e un sottotipo T2, allora T2 è *consistent-with* T1 (*Liskov substitution*).

> *Nella programmazione orientata agli oggetti il principio di sostituzione di Liskov (Liskov substitution principle) è una particolare definizione di sottotipo, introdotta da Barbara Liskov e Jeannette Wing nel 1993. La formulazione sintetica del principio è la seguentez:*
>
> > *"Se $q(x)$ è una proprietà che si può dimostrare essere valida per oggetti x di tipo T, allora $q(y)$ deve essere valida per oggetti y di tipo S dove S è un sottotipo di T."*

Questa nozione di sottotipo è quindi basata sulla nozione di sostituibilità secondo cui, se S è un sottotipo di T, allora oggetti dichiarati in un programma di tipo T possono essere sostituiti con oggetti di tipo S senza alterare la correttezza dei risultati del programma.

Ogni tipo è *consistent-with* `Any` e `Any` è *consistent-with* ogni tipo.

`Union` L'unione è più utile con tipi che non sono coerenti tra loro. Ad esempio: `Union[int, float]` è ridondante perché `int` è coerente con `float`. Se si usa solo `float` per annotare il parametro, accetterà anche valori `int`.

Return types

Tipicamente si vuole evitare di creare una funzione che possa dare in output più di un tipo. Infatti questo costringe l'utente a capire quale sia il tipo.

Se la funzione non ritorna nulla, si può usare `NoReturn`. Queste funzioni tipicamente sono quelle che elevano una eccezione interrompendo il runtime.

Numeri

Sebbene la struttura gerarchica dei tipi numerici in Python, definita nel modulo `numbers` e descritta in PEP 3141, funzioni bene per il controllo dei tipi a runtime, non è supportata per il controllo statico dei tipi. Questo significa che i controllori di tipo statico come `Mypy` non utilizzano le classi astratte del modulo `numbers` (come `Number`, `Complex`, ecc.) per verificare i tipi numericamente.

Il problema principale è che PEP 484 ha stabilito una struttura parallela per il controllo statico dei tipi, raccomandando l'uso di `Union` types o protocolli come `SupportsFloat` invece delle classi astratte del modulo `numbers`. Ciò significa che se si desidera annotare gli argomenti numerici per un controllo statico efficace, bisogna usare tipi concreti come `int`, `float`, o combinazioni di questi tramite `Union` types.

Le opzioni da utilizzare sono, nell'ordine:

1. tipi concreti come `int`, `float`, ...
2. unioni
3. usare i controlli di protocolli come `SupportsFloat`

`tuple`

A differenza delle liste, le tuple permettono un più accurato *type hinting*. Ad esempio con `tuple[float, float]`, ci si aspetta una tupla di due elementi di tipo `float`. Analogamente si possono definire oggetti da `NamedTuple`.

```
...
class Coordinate(NamedTuple):
    lat: float
    lon: float

def geohash(lat_lon: Coordinate) -> str:
    return gh.encode(*lat_lon, PRECISION)
```

Bisogna notare che `Coordinate` è consistente con `tuple[float, float]` (ma il contrario non è vero, `Coordinate` potrebbe avere altri metodi definiti).

In ultimo, se la tupla ha lunghezza non specificata, si può usare `tuple[int, ...]`.

Astrazione

Per un mapping generico si può usare `Mapping[KeyType, ValueType]` (Python $\geq$ 3.9).

Questo è un esempio di utilizzo di classi astratte che permettono di essere molto 'liberali' in ciò che si accetta. Per annotare gli output però, è meglio essere precisi ed evitare l'utilizzo di classi astratte.

Per gli iterabili, analogamente al mapping, si usa `Iterable[ItemType]` (si richiede l'implementazione di `__iter__`). Oppure si può usare `Sequence` (si richiede l'implementazione di `__getitem__` e `__len__`).

Talvolta per definire i tipi all'interno di dizionari e liste, si può usare una sintassi del tipo:

```
from typing import TypeAlias

FromTo: TypeAlias = tuple[str, str]

def zip_replace(text: str, changes: Iterable[FromTo]) -> str:
    ...
```

Parametrizzazione del tipo

```
from collections.abc import Sequence
from random import shuffle
from typing import TypeVar

T = TypeVar('T')

def sample(population: Sequence[T], size: int) -> list[T]:
    if size < 1:
        raise ValueError('size must be >= 1')
    result = list(population)
    shuffle(result)
    return result[:size]
```

Si può definire un tipo generico (nell'esempio `T`) di modo che si mantenga la coerenza tra input e output (sia la sequenza di input che la lista di output sono contenuti dallo stesso tipo).

La parametrizzazione può essere fatta anche restringendo i tipi accettati ma continuando a indicare la coerenza tra tipi quando il parametro del tipo ricompare.

```
from collections.abc import Iterable
from decimal import Decimal
from fractions import Fraction
from typing import TypeVar

NumberT = TypeVar('NumberT', float, Decimal, Fraction)

def mode(data: Iterable[NumberT]) -> NumberT:
```

Se si vogliono includere tutti i sottotipi di una classe astratta si può usare il parametro `bound`: ad esempio `HashableT = TypeVar('HashableT', bound=Hashable)`

Static Protocols

Nel contesto dei suggerimenti di tipo, un protocollo è una sottoclasse di `typing.Protocol` che definisce un'interfaccia che un controllore di tipo può verificare. Un protocollo è definito specificando uno o più metodi che il type checker deve verificare.

Un protocollo è una sotto-classe di `typing.Protocol` nel cui corpo ci sono una o più definizioni con `...`.

```
from typing import Protocol, Any

class SupportsLessThan(Protocol):
    def __lt__(self, other: Any) -> bool: ...
```

Callable

Per annotare *higher-order functions* si può usare una annotazione del tipo `Callable[[ParamType1, ParamType2], ReturnType]`.

Decoratori

Un decoratore è un oggetto chiamabile che prende un altra funzione come argomento. Ovvero:

```
@decorate
def target():
    print('running target()')
```

è equivalente a

```
def target():
    print('running target()')

target = decorate(target)
```

Una importante caratteristica dei decoratori è che vengono eseguiti durante l'import e non al runtime.

Variable Scopes

Se una variabile viene definita al di fuori di una funzione (variabile globale), questa può essere utilizzata anche all'interno della funzione.

Se però un'altra variabile con lo stesso nome viene definita all'interno della funzione, allora quella variabile è locale.

```
>>> b = 6
>>> def f2(a):
...     print(a)
...     print(b)
...     b = 9
...
>>> f2(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f2
UnboundLocalError: local variable 'b' referenced before assignment
```

Questo è molto importante in codici complessi perché evita che vengano sovrascritte variabili globali e/o che queste vengano utilizzate erroneamente.

Closures

Una *closure* è una funzione con uno scope esteso, in cui le variabili ivi definite non sono né globali né locali ma sono 'variabili libere'. Queste variabili sono variabili locali della funzione che contiene la *closure*.

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total / len(series)

    return averager

avg = make_averager()
avg(10)
# 10.0
avg(11)
# 10.5
```

```
avg(15)
# 12.0
```

nonlocal

Con variabili immutabili, diventa necessario utilizzare la *keyword* `nonlocal`:

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

Quindi:

1. Se c'è una dichiarazione globale di `x`, `x` proviene dalla variabile globale del modulo e le viene assegnata.
2. Se c'è una dichiarazione non locale di `x`, `x` proviene dalla variabile locale della funzione circostante più vicina dove `x` è definita, e le viene assegnata.
3. Se `x` è un parametro o se le viene assegnato un valore nel corpo della funzione, allora `x` è la variabile locale.
4. Se si fa riferimento a `x`, ma non le viene assegnato alcun valore e non è un parametro:
    a. `x` verrà cercata negli ambiti locali dei corpi delle funzioni circostanti (ambiti non locali).
    b. Se non viene trovata negli ambiti circostanti, verrà letta dall'ambito globale del modulo.
    c. Se non viene trovata nell'ambito globale, verrà letta da `__builtins__.__dict__`.

Decoratori importanti

functools.wraps

Quando si utilizza un decorator, di fatto si passa la funzione a un *closure*. Così facendo la funzione closure sovrascrive il nome e perde le docstring. Per questa ragione si può utilizzare `@functools.wraps(func)`.

```
import time
import functools


def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_lst = [repr(arg) for arg in args]
        arg_lst.extend(f'{k}={v!r}' for k, v in kwargs.items())
        arg_str = ', '.join(arg_lst)
        print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
        return result
    return clocked
```

Memoizzazione

La memoizzazione è una tecnica che consiste nel salvare i valori restituiti da una funzione in modo da averli a disposizione e non doverli ricalcolare in un utilizzo successivo.

In python 3.9+, si può usare il decoratore `cache`.

```
import functools


@functools.cache
def ...():
    ...
```

Tutti gli argomenti della funzione decorata devono essere *hashable* dato che viene usato un dizionario per memorizzare i risultati.

Tipicamente si usa `cache` per le funzioni ricorsive e per call ad APIs.

`cache` non è altro che un wrapper attorno a `lru_cache`, compatibile con versioni di python precedenti alla 3.9 e che permette anche di passare dei parametri (come `maxsize` per definire quante entry devono essere memorizzate).

singledispatch/multipledispatch

Potremmo volere diversi comportamenti di una funzione a seconda della tipologia di input. Possiamo usare `singledispatch` se importa solo il primo argomento, `multipledispatch` se si controllano i tipi di altri argomenti.

```
from functools import singledispatch
from collections import abc
import fractions
import decimal
import html
import numbers

@singledispatch
def htmlize(obj: object) -> str:
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'

@htmlize.register
def _(text: str) -> str:
    content = html.escape(text).replace('\n', '<br/>\n')
    return f'<p>{content}</p>'

@htmlize.register
def _(seq: abc.Sequence) -> str:
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'

@htmlize.register
def _(n: numbers.Integral) -> str:
    return f'<pre>{n} (0x{n:x})</pre>'

@htmlize.register(fractions.Fraction)
def _(x) -> str:
    frac = fractions.Fraction(x)
    return f'<pre>{frac.numerator}/{frac.denominator}</pre>'

@htmlize.register(decimal.Decimal)
@htmlize.register(float)
def _(x) -> str:
    frac = fractions.Fraction(x).limit_denominator()
    return f'<pre>{x} ({frac.numerator}/{frac.denominator})</pre>'
```

Anche in questo caso, quando possibile, sarebbe opportuno registrare tipi astratti.

Design pattern

Strategy design pattern

Il Strategy Design Pattern è un pattern comportamentale che permette di definire una famiglia di algoritmi, incapsularli in classi separate e renderli intercambiabili. In altre parole, consente di cambiare il comportamento di un oggetto senza modificare il codice che lo utilizza.

Nel caso classico si definisce una classe base per la famiglia di algoritmi e i dettagli di implementazione sono definite nelle varie sottoclassi.

```
from abc import ABC, abstractmethod
from collections.abc import Sequence
from decimal import Decimal
from typing import NamedTuple, Optional


class Customer(NamedTuple):
    name: str
    fidelity: int


class LineItem(NamedTuple):
    product: str
    quantity: int
    price: Decimal

    def total(self) -> Decimal:
        return self.price * self.quantity


class Order(NamedTuple):  # the Context
    customer: Customer
    cart: Sequence[LineItem]
    promotion: Optional['Promotion'] = None

    def total(self) -> Decimal:
        totals = (item.total() for item in self.cart)
        return sum(totals, start=Decimal(0))

    def due(self) -> Decimal:
        if self.promotion is None:
            discount = Decimal(0)
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'


class Promotion(ABC):  # the Strategy: an abstract base class
    @abstractmethod
    def discount(self, order: Order) -> Decimal:
        """Return discount as a positive dollar amount"""


class FidelityPromo(Promotion):  # first Concrete Strategy
```

```
    """5% discount for customers with 1000 or more fidelity points"""

    def discount(self, order: Order) -> Decimal:
        rate = Decimal('0.05')
        if order.customer.fidelity >= 1000:
            return order.total() * rate
        return Decimal(0)


class BulkItemPromo(Promotion):  # second Concrete Strategy
    """10% discount for each LineItem with 20 or more units"""

    def discount(self, order: Order) -> Decimal:
        discount = Decimal(0)
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * Decimal('0.1')
        return discount


class LargeOrderPromo(Promotion):  # third Concrete Strategy
    """7% discount for orders with 10 or more distinct items"""

    def discount(self, order: Order) -> Decimal:
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * Decimal('0.07')
        return Decimal(0)
```

In realtà, in un linguaggio di programmazione dove le funzioni sono oggetti di prima classe, si può utilizzare una implementazione ben più compatta:

```
from collections.abc import Sequence
from dataclasses import dataclass
from decimal import Decimal
from typing import Optional, Callable, NamedTuple


class Customer(NamedTuple):
    name: str
    fidelity: int


class LineItem(NamedTuple):
    product: str
    quantity: int
    price: Decimal

    def total(self):
        return self.price * self.quantity

@dataclass(frozen=True)
class Order:  # the Context
    customer: Customer
    cart: Sequence[LineItem]
    promotion: Optional[Callable[['Order'], Decimal]] = None  1

    def total(self) -> Decimal:
```

```
        totals = (item.total() for item in self.cart)
        return sum(totals, start=Decimal(0))

    def due(self) -> Decimal:
        if self.promotion is None:
            discount = Decimal(0)
        else:
            discount = self.promotion(self)  2
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'


3


def fidelity_promo(order: Order) -> Decimal:  4
    """5% discount for customers with 1000 or more fidelity points"""
    if order.customer.fidelity >= 1000:
        return order.total() * Decimal('0.05')
    return Decimal(0)


def bulk_item_promo(order: Order) -> Decimal:
    """10% discount for each LineItem with 20 or more units"""
    discount = Decimal(0)
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * Decimal('0.1')
    return discount


def large_order_promo(order: Order) -> Decimal:
    """7% discount for orders with 10 or more distinct items"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * Decimal('0.07')
    return Decimal(0)
```

Command design pattern

Il Command Design Pattern è un pattern comportamentale che trasforma una richiesta (un'azione) in un oggetto, che incapsula tutte le informazioni necessarie per eseguire l'azione. Questo pattern permette di separare il richiedente dell'azione (il client) dal ricevente che la esegue. La forza del Command Pattern sta nel fatto che permette di eseguire, annullare o registrare azioni in modo decoupled (indipendente) dal chiamante.

```
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self):
        pass


# Azioni
class Light:
    def on(self):
```

```
        print("Luce accesa")

    def off(self):
        print("Luce spenta")

# Comandi
class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.off()

# Invoker dei comandi
class RemoteControl:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def press_button(self):
        if self.command:
            self.command.execute()

# Client
if __name__ == "__main__":
    # Crea il ricevente
    light = Light()

    # Crea i comandi concreti
    light_on = LightOnCommand(light)
    light_off = LightOffCommand(light)

    # Crea l'invocatore (telecomando)
    remote = RemoteControl()

    # Usa il telecomando per accendere la luce
    remote.set_command(light_on)
    remote.press_button()  # "Luce accesa"

    # Usa il telecomando per spegnere la luce
    remote.set_command(light_off)
    remote.press_button()  # "Luce spenta"
```

Spiegazione:

1. **Command**: È un'interfaccia astratta che definisce il metodo `execute()`.
2. **Light**: È il ricevente che esegue le azioni vere e proprie (accendere e spegnere la luce).
3. **LightOnCommand** e **LightOffCommand**: Sono i comandi concreti che invocano le azioni sul ricevente

`Light`.
4. **RemoteControl**: È l'invocatore che tiene traccia del comando e lo esegue.
5. **Client**: È la parte in cui si creano tutti gli oggetti e si eseguono i comandi.

Oggetti Pythonici

Rappresentazione degli oggetti

Ci sono due metodi per rappresentare oggetti: `repr` e `str`.

1. `repr` è destinato a dare una rappresentazione univoca dell'oggetto di modo che sia ricreabile. Idealmente tale che `x==eval(repr(x))` sia vero.
2. `str` è destinato a essere una stringa facilmente leggibile dall'utente

Oggetto pythonico

```
from array import array
import math


class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.x = float(x)     # Conversion to the 'right' format for early error-catching
        self.y = float(y)

    def __iter__(self):
        return (i for i in (self.x, self.y))

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)

    def __str__(self):
        return str(tuple(self))  # Sfruttiamo __iter__ per la conversione a tuple

    def __bytes__(self): # esportare a bytes
        return (bytes([ord(self.typecode)]) +
                bytes(array(self.typecode, self)))

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    @classmethod
    def frombytes(cls, octets): # leggere da bytes
        typecode = chr(octets[0]) # typecode dal primo byte
        memv = memoryview(octets[1:]).cast(typecode)  # leggere il resto dei byte in memoria
        return cls(*memv) # crare la classe puntando alla memoria
```

`classmethod` vs `staticmethod`

Il decoratore `classmethod` opera sulla classe e non su una istanza. La cosa più tipica è proprio quella dell'esempio, dove si vuole creare un'istanza dai byte. `staticmethod` d'altra parte è una semplice funzione che esiste all'interno del contesto della classe, senza ricevere alcun argomento.

Formattazione stringhe

Tutti i metodi di formattazione (f-strings, `format()`, e `.format()`) usano il loro metodo `.__format__(specifier)`. Se la classe non ha un metodo**format**, viene usato**str**‘ (in questo caso nessuno specifier è supportato.

```
>>> brl = 1 / 4.82  # BRL to USD currency conversion rate
>>> brl
0.20746887966804978
>>> format(brl, '0.4f')
'0.2075'
>>> '1 BRL = {rate:0.2f} USD'.format(rate=brl)
'1 BRL = 0.21 USD'
>>> f'1 USD = {1 / brl:0.2f} BRL'
'1 USD = 4.82 BRL'

>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.3f')
Traceback (most recent call last):
  ...
TypeError: non-empty format string passed to object.__format__
```

Un’altra proprietà necessaria per l’utilizzo con f-strings è `__match_args__`. Ad esempio:

```
class Vector2d:
    __match_args__ = ('x', 'y')

    # etc...
```

hash

Per rendere un oggetto *hashable*, questo deve diventare immutabile.

Una possibilità è di rendere le sue proprietà *read-only* usando `@property` (se non si facesse così non avrebbe senso verificare alcun hash). Inoltre si deve poi implementare la funzione `__hash__`.

Attributi privati

In python non c’è un modo per settare come privati gli attributi. Si può usare un `_` davanti al nome dell’attributo, e in questo caso è noto per convenzione che quella variabile non dovrebbe essere toccata. Un’altra possibilità è quella di usare un `__` (doppio) davanti al nome. Quello che python fa è detta *name mangling*: la variabile `__foo` della classe `Bar` viene registrata come `_Bar__foo`. Quindi per andare a modificarla l’utente deve davvero volere alterare l’attributo.

`__slots__` e salvare memoria

Gli attributi di una classe vengono salvati in un dizionario con un overhead significativo. Questo permette di aggiungere attributi anche dopo la definizione e inizializzazione della classe. Se gli attributi della classe vengono dichiarati durante la definizione della classe, si può usare `__slots__`:

```
>>> class Pixel:
...     __slots__ = ('x', 'y')  # tupla, deve essere immutabile!
...                             # Lista sarebbe supportata ma tupla è concettualmente appropriata
...
>>> p = Pixel()
>>> p.__dict__
Traceback (most recent call last):
  ...
AttributeError: 'Pixel' object has no attribute '__dict__'
```

```
>>> p.x = 10
>>> p.y = 20
>>> p.color = 'red'
Traceback (most recent call last):
  ...
AttributeError: 'Pixel' object has no attribute 'color'
```

Quando si usa `__slots__`, la classe non avrà più un dizionario che salva le variabili e non se ne possono aggiungere di ulteriori. Questo ovviamente non vale per le sottoclassi:

```
>>> class OpenPixel(Pixel):
...     pass
...
>>> op = OpenPixel()
>>> op.__dict__
{}
>>> op.x = 8
>>> op.__dict__
{}
>>> op.x
8
>>> op.color = 'green'
>>> op.__dict__
{'color': 'green'}
```

D'altra parte, se si vuole preservare il risparmio di memoria, si deve definire `__slots__` anche nella sottoclasse.

Metodi speciali per sequenze

Un esempio di implementazione di una classe basata su sequenze:

```
from array import array
import reprlib
import math


class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        return iter(self._components)

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[components.find('['):-1]
        return f'Vector({components})'

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components))

    def __abs__(self):
        return math.hypot(*self)

    def __bool__(self):
        return bool(abs(self))

    def __len__(self):
        return len(self._components)

    def __getitem__(self, key):
        if isinstance(key, slice):
            cls = type(self)
            return cls(self._components[key])
        index = operator.index(key)
        return self._components[index]

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv)

    def __eq__(self, other):
        if len(self) != len(other):
            return False
        for a, b in zip(self, other):
            if a != b:
                return False
```

```
        return True

    def __hash__(self):
        hashes = map(hash, self._components)
        return functools.reduce(operator.xor, hashes)

    __match_args__ = ('x', 'y', 'z', 't')

    def __getattr__(self, name):
        cls = type(self)
        try:
            pos = cls.__match_args__.index(name)
        except ValueError:
            pos = -1
        if 0 <= pos < len(self._components):
            return self._components[pos]
        msg = f'{cls.__name__!r} object has no attribute {name!r}'
        raise AttributeError(msg)

    def angle(self, n):
        r = math.hypot(*self[n:])
        a = math.atan2(r, self[n-1])
        if (n == len(self) - 1) and (self[-1] < 0):
            return math.pi * 2 - a
        else:
            return a

    def angles(self):
        return (self.angle(n) for n in range(1, len(self)))

    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('h'):  # hyperspherical coordinates
            fmt_spec = fmt_spec[:-1]
            coords = itertools.chain([abs(self)],
                                     self.angles())
            outer_fmt = '<{}>'
        else:
            coords = self
            outer_fmt = '({})'
        components = (format(c, fmt_spec) for c in coords)
        return outer_fmt.format(', '.join(components))
```

1. I componenti del vettore vengono assegnati ad un attributo protetto e memorizzati in un array (che ha molto meno overhead di una lista).
2. Si definisce `__iter__` per permettere di iterare sui componenti
3. `reprlib.repr()` per limitare la lunghezza dell'output (troncandolo oltre un certo punto e aggiungendo `...`)
4. `math.hypot` per calcolare il modulo, equivale a `math.sqrt(sum(x * x for x in self))`
5. `__len__` e `__getitem__` permettono a `Vector` di comportarsi proprio come una sequenza
6. Implementazione di `__eq__` efficiente (non devo confrontare l'intera lista sempre, appena c'è qualcosa di diverso mi fermo)
7. `__hash__` viene fatto calcolando `hash()` di ogni componente e vengono combinati tra loro utilizzando `reduce(xor, hashes)`

Interfaccia e protocolli

Protocolli Dinamici e Statici

I protocolli sono essenzialmente interfacce implicite o esplicite che una classe può implementare.

Protocolli Dinamici

Essi sono definiti per convenzione e descritti nella documentazione. Si basano sul "duck typing": se un oggetto ha i metodi giusti, Python lo tratterà come se implementasse un certo protocollo.

In questo caso, il vantaggio è che un oggetto può implementare solo *una parte* di un protocollo. Lo svantaggio è che non possono essere verificati dai type checker statici (come `MyPy`).

Ad esempio, il protocollo di sequenza prevede l'implementazione di `__getitem__(self, i)` per permettere l'accesso agli elementi tramite indice. Ma idealmente, una sequenza dovrebbe anche implementare `__len__(self)`.

```python
class Vowels:
    def __getitem__(self, i):
        return 'AEIOU'[i]

v = Vowels()
print(v[0])  # Output: A
print('E' in v) # Output: True
```

Python è abbastanza flessibile da sfruttare `__getitem__` anche per le funzioni mancanti dell'interfaccia di una sequenza. Ad esempio `__iter__` non esiste ma quando deve iterare, python automaticamente usa un range che parte da 0 e lo passa a `__getitem__`.

Fail-fast L'utilizzo di protocolli dinamici rende importante il concetto di *fail-fast*: dato che molti bug possono essere individuati solo durante il runtime, si vuole alzare le eccezioni il prima possibile. Quindi ad esempio se l'argomento della funzione è una sequenza:

```python
    def __init__(self, iterable):
        self._balls = list(iterable)
```

Protocolli Statici

Sono definiti esplicitamente creando sottoclassi di `typing.Protocol`. Richiedono che una classe implementi *tutti* i metodi dichiarati nel protocollo. Per questo possono essere verificati dai type checker statici.

Abstract Base Classes (ABC)

- Sono un altro modo per definire interfacce esplicite in Python.
- Un'ABC definisce uno o più metodi astratti che le sottoclassi devono implementare.
- Le ABC possono essere utilizzate per la type checking a runtime (tramite "goose typing") e per i type hints per i type checker statici.

Differenze chiave tra protocolli dinamici e statici:

Caratteristica	Protocollo Dinamico	Protocollo Statico
Definizione	Implicita, basata su convenzioni	Esplicita, tramite `typing.Protocol`
Implementazione	Implementazione parziale può essere sufficiente	Richiede l'implementazione di tutti i metodi definiti
Type checking	Non verificabile staticamente	Verificabile staticamente

Goose Typing

Il fatto che esista un metodo con un certo nome in una classe, di per sé non garantisce che quella classe implementi un metodo che esegue quello che ci si aspetta. Ad esempio il metodo `draw` può essere adatto sia alla classe `Artist` che

alla classe `Lottery`, ma chiaramente fa due cose molto diverse. Se si adotta un metodo duck typing, si dovrebbero accettare entrambe le classi, senza fare alcun check del tipo `isinstance(obj, cls)`. Nel goose typing, questo check è accettato fintanto che `cls` è una classe astratta di base (la metaclasse di `cls` deve essere `abc.ABCMeta`).

ABC di python ha un grande vantaggio, una classe non deve essere registrata come sottoclasse perchè ABC la riconosca come tale. Talvolta è sufficiente avere alcuni metodi speciali implementati:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
```

ABC nella libreria standard

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		`__contains__`	
Hashable		`__hash__`	
Iterable		`__iter__`	
Iterator	Iterable	`__next__`	`__iter__`
Reversible	Iterable	`__reversed__`	
Generator	Iterator	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized		`__len__`	
Callable		`__call__`	
Collection	Sized, Iterable, Container		`__contains__`, `__iter__`, `__len__`
Sequence	Reversible, Collection	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	Sequence	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited Sequence methods and `append`, `clear`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
ByteString	Sequence	`__getitem__`, `__len__`	Inherited Sequence methods
Set	Collection		`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__rsub__`, `__xor__`, `__rxor__` and `isdisjoint`
MutableSet	Set	`add`, `discard`	Inherited Set methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
Mapping	Collection	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
MutableMapping	Mapping	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited Mapping methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
MappingView	Sized	`__init__`, `__len__` and `__repr__`	
ItemsView	MappingView, Set		`__contains__`, `__iter__`
KeysView	MappingView, Set		`__contains__`, `__iter__`
ValuesView	MappingView, Collection		`__contains__`, `__iter__`
Awaitable		`__await__`	
Coroutine	Awaitable	`send`, `throw`	`close`
AsyncIterable		`__aiter__`	
AsyncIterator	AsyncIterable	`__anext__`	`__aiter__`
AsyncGenerator	AsyncIterator	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`
Buffer		`__buffer__`	

Creare sottoclasse di ABC

È importante definire tutti i metodi astratti richiesti dalla ABC. Alcune implementazioni concrete sono invece ereditate dalla classe base. Talvolta si vuole sovrascriverli se la classe ha delle proprietà speciali: ad esempio se la classe è una sequenza di elementi ordinati, si vuole sovrascrivere `__contains__` di `abc.Sequence` con un binary search al posto che una ricerca sequenziale che è implementata di default.

Creare una ABC

1. Dichiarare la classe come sottoclasse di `abc.ABC` o di una qualsiasi altra ABC.
2. Definire implementazioni concrete o metodi astratti con `@abc.abstractmethod` come decoratore.

```
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```

`@abc.abstractmethod` deve sempre essere il decoratore più interno.

ABC hanno anche il metodo `register` che permette di dichiarare che una classe è sottoclasse di un'altra senza che alcun check venga fatto: python si fida sulla parola.

```
@AbstractClass.register
class ConcreteClass():
   ...
```

Ereditarietà

`super`

La funzione `super` risulta fondamentale per costruire le sottoclassi. Spesso la sottoclasse che sovrascrive un metodo della classe base, vuole comunque chiamare il metodo della classe base che sovrascrive. In realtà ci sarebbero due opzioni; prendiamo ad esempio una classe la cui base class è `OrderedDict`:

1. Super: `super().__setitem__(key, value)`
2. Classe base: `OrderedDict.__setitem__(key, value)`

Bisogna sempre preferire la prima opzione, sia perché `super` implementa una logica per gestire una gerarchia con una classe con molteplici eredità, sia per non hard-code `OrderedDict` dentro la classe.

Sottoclassi con tipi *built-in*

Le sottoclassi dei tipi *built-in* in Python presentano un comportamento particolare: i metodi sovrascritti non vengono sempre considerati dagli altri metodi *built-in*. Questo contraddice il principio della *late binding*, secondo cui la ricerca di un metodo dovrebbe partire sempre dalla classe dell'oggetto ricevente (*self*) e determinarsi a runtime.

Sottoclassi con molte ereditarietà

L'ordine di risoluzione dei metodi (MRO) è la sequenza con cui Python cerca i metodi e gli attributi nelle classi ereditarie. Python utilizza l'algoritmo C3 linearization (o algoritmo di MRO) per determinare l'ordine di ricerca.

Il Diamond Problem Il diamond problem si verifica quando una classe eredita da due classi che a loro volta ereditano da una stessa superclasse. Questo può causare ambiguità nella risoluzione dei metodi.

```python
class A:
    def metodo(self):
        print("Metodo di A")

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

d = D()
d.metodo()  # Quale metodo viene chiamato?
```

In questo caso, l'MRO di D segue l'ordine D → B → C → A, evitando chiamate duplicate e risolvendo il conflitto in modo deterministico. Python gestisce il diamond problem grazie all'algoritmo C3, garantendo che ogni classe venga visitata una sola volta nel corretto ordine.

Mixin Classes

Le Mixin Classes sono classi progettate per fornire funzionalità aggiuntive ad altre classi senza essere usate come classi base indipendenti. Favoriscono la composizione rispetto alla gerarchia profonda → Aiutano a mantenere il codice più leggibile rispetto a una catena complessa di ereditarietà.

```python
class LoggerMixin:
    def log(self, message):
        print(f"[LOG]: {message}")

class FileHandler:
    def open_file(self, filename):
        print(f"Apertura del file {filename}")
```

```python
class FileLogger(FileHandler, LoggerMixin):  # FileLogger eredita sia da FileHandler che da LoggerMixin
    pass

file_logger = FileLogger()
file_logger.open_file("documento.txt")  # Metodo della classe FileHandler
file_logger.log("File aperto con successo")  # Metodo ereditato da LoggerMixin
```

Le mixin sono molto usate sia in `abc` per definire delle implementazioni concrete, sia nei framework come Django, ad esempio per aggiungere autenticazione o permessi alle classi delle viste.

Best practices

1. Usare *object-composition* al posto che *class inheritance*: rende l'implementazione più flessibile.
2. Se una classe è destinata a definire un'interfaccia, dovrebbe essere dichiarata esplicitamente come una Abstract Base Class (ABC) o una sottoclasse di typing.Protocol. Questo rende chiaro che la classe fornisce un'interfaccia che altre classi devono implementare. Se vuoi un'interfaccia rigida → usa ABCs (abc.ABC). Se vuoi più flessibilità e duck typing → usa Protocol (typing.Protocol).
3. Usare classi aggregate: una Aggregate Class è una classe che viene creata principalmente ereditando da più Mixin Classes, senza aggiungere nuove strutture o comportamenti propri. Il suo scopo è combinare funzionalità già esistenti in una singola interfaccia comoda per l'utente.
4. Eredita solo da classi progettate per essere sottoclassate. Quando si crea una sottoclasse in Python, è importante ereditare solo da classi esplicitamente progettate per essere estese. Questo evita problemi di compatibilità, comportamenti inaspettati e rende il codice più mantenibile.

Typing avanzato

Overload

Esistono delle funzioni che accettano diverse combinazioni di argomenti. Per questa ragione si può usare `@typing.overload` per annotare queste diverse combinazioni.

```
@overload
def add(a: int, b: int) -> int: ...

@overload
def add(a: float, b: float) -> float: ...

@overload
def add(a: str, b: str) -> str: ...

def add(a: Union[int, float, str], b: Union[int, float, str]) -> Union[int, float, str]:
    return a + b
```

L'utilizzo di `...` quando si fa l'overload è necessario per soddisfare la sintassi richiesta da python.

`TypedDict`

Quando si ha un dizionario con molteplici chiavi e valori di tipi diversi, non esiste in python un buon modo per annotare il tipo. Quello che si può fare è utilizzare `TypedDict` in una classe:

```
from typing import TypedDict

# per annotare questo dizionario:
# {"isbn": "0134757599",
#  "title": "Refactoring, 2e",
#  "authors": ["Martin Fowler", "Kent Beck"],
#  "pagecount": 478}
# posso definire una classe con `TypedDict`

class BookDict(TypedDict):
    isbn: str
    title: str
    authors: list[str]
    pagecount: int
```

Durante il runtime, creare una sottoclasse di `TypedDict` è equivalente a definire un dizionario ma con il vantaggio di una annotazione più semplice. Ovviamente non si può scrivere un initializer con valori di defult per i vari campi, non si possono definire metodi e non vengono definiti degli attributi, si deve interrogare l'oggetto come un dizionario normale.

Type casting

Il *typecasting* in Python, attraverso `cast()`, consente di forzare il tipo di una variabile quando il type checker non è in grado di dedurlo correttamente. Sebbene abbia delle controindicazioni, come il rischio di mascherare errori di tipo, può essere utile rispetto ad alternative peggiori come `# type: ignore`, che è meno informativo, o l'uso eccessivo di `Any`, che può propagare imprecisioni nell'inferenza dei tipi.

Type hints sono letti durante il runtime

Python memorizza gli hint di tipo nei metadati delle funzioni, classi e moduli nell'attributo `__annotations__`. Questo può creare dei problemi:

1. Maggiore consumo di CPU e memoria all'importazione dei moduli con molti hint di tipo. Per risolvere questo problema si può richiedere a python di non valutare le annotazioni e lasciare come stringhe.

Questo comportamento non è ancora il predefinito poiché romperebbe alcune librerie come FastAPI e pydantic che si basano sugli hint di tipo a runtime.

2. Forward references: per riferirsi a classi non ancora definite, bisogna usare stringhe invece dei tipi reali.

```
class Rectangle:
   def stretch(self, factor: float) -> 'Rectangle':
      return Rectangle(width=self.width * factor)
```

La soluzione a questo è l'utilizzo di `inspect.get_annotations()` (Python 3.10+).

```
from inspect import get_annotations

hints = get_annotations(Rectangle.stretch)
print(hints)  # {'factor': <class 'float'>, 'return': <class '__main__.Rectangle'>}
```

Ci sono continue evoluzioni su questo argomento dato che i possibili margini di miglioramento di performance sono molti, ma molte soluzioni rischiano di rompere classi che si basano su type hinting a runtime.

Implementazione di una classe con tipo generico

```
import random

from collections.abc import Iterable
from typing import TypeVar, Generic

from tombola import Tombola

T = TypeVar('T')

class LottoBlower(Tombola, Generic[T]):

    def __init__(self, items: Iterable[T]) -> None:
        self._balls = list[T](items)

    def load(self, items: Iterable[T]) -> None:
        self._balls.extend(items)

    def inspect(self) -> tuple[T, ...]:
        return tuple(self._balls)
```

1. Si usa `Generic[T]` il parametro di tipo formale ($\leftrightarrow$ parametro di tipo effettivo che si definisce con il tipo specifico passato quando si crea un istanza della classe).
2. I metodi successivi sono tutti vincolati dal parametro formale.

Variance

La **variance** descrive come i tipi generici si comportano rispetto alla gerarchia delle sottoclassi. Esistono tre casi principali: **invariance, covariance e contravariance**.

1. Invariance

Un tipo generico è **invariante** se due istanze con tipi diversi non sono compatibili tra loro, anche se i tipi appartengono alla stessa gerarchia.

Ad esempio, supponiamo di avere un distributore di bevande generico:

```
from typing import TypeVar, Generic

T = TypeVar('T')
```

```
class BeverageDispenser(Generic[T]):
    def __init__(self, beverage: T) -> None:
        self.beverage = beverage

def install(dispenser: BeverageDispenser["Juice"]) -> None:
    pass
```

Se proviamo a passare `BeverageDispenser[OrangeJuice]` a `install()`, otteniamo un errore, anche se `OrangeJuice` è una sottoclasse di `Juice`. Questo perché `BeverageDispenser[T]` è invariante per impostazione predefinita.

Python applica l'invarianza alle collezioni mutabili (come `list` o `set`) per evitare problemi di sicurezza sui tipi.

2. Covariance

Un tipo generico è **covariante** quando segue la gerarchia dei sottotipi: se `B` è una sottoclasse di `A`, allora `Container[B]` può essere usato dove è richiesto `Container[A]`.

```
T_co = TypeVar('T_co', covariant=True)

class BeverageDispenser(Generic[T_co]):
    def __init__(self, beverage: T_co) -> None:
        self.beverage = beverage

def install(dispenser: BeverageDispenser["Juice"]) -> None:
    pass
```

Ora possiamo passare un `BeverageDispenser[OrangeJuice]` a `install()`, perché segue la stessa direzione della gerarchia dei tipi.

Le collezioni immutabili, come `frozenset`, sono covarianti per natura.

3. Contravariance

Un tipo generico è **contravariante** quando la relazione tra i tipi si inverte: se `B` è una sottoclasse di `A`, allora `Container[A]` può essere usato dove è richiesto `Container[B]`.

```
from typing import TypeVar, Generic

class Refuse:
    """Any refuse."""

class Biodegradable(Refuse):
    """Biodegradable refuse."""

class Compostable(Biodegradable):
    """Compostable refuse."""

T_contra = TypeVar('T_contra', contravariant=True)

class TrashCan(Generic[T_contra]):
    def put(self, item: T_contra) -> None:
        pass

def deploy(can: TrashCan["Biodegradable"]) -> None:
    pass
```

Possiamo passare un `TrashCan[Refuse]` a `deploy()`, perché può contenere qualsiasi tipo di rifiuto, ma **non** possiamo passare un `TrashCan[Compostable]`, perché sarebbe troppo specifico.

Le strutture di sola scrittura, come i callback, sono spesso contravarianti.

Regole Generali

- Se un tipo è **solo in uscita** (output), può essere **covariante**.
- Se un tipo è **solo in ingresso** (input), può essere **contravariante**.
- Se un tipo è **sia in ingresso che in uscita**, deve essere **invariante**.

Questi principi garantiscono sicurezza e coerenza nei tipi generici in Python.

Protocollo Generico Statico

In Python 3.10, esistono protocolli generici statici, come `SupportsAbs`, che permettono di definire interfacce con metodi specifici senza dover ereditare esplicitamente da una classe base. Questo è diverso dall'implementazione delle classi generiche, che definiscono un comportamento per una struttura dati con parametri di tipo.

```
from typing import Protocol, runtime_checkable, TypeVar
from abc import abstractmethod

T_co = TypeVar('T_co', covariant=True)

@runtime_checkable
class SupportsAbs(Protocol[T_co]):
    @abstractmethod
    def __abs__(self) -> T_co:
        pass
```

1. È un protocollo, quindi non richiede un'implementazione diretta.
2. È covariante nel tipo di ritorno (`T_co`).
3. Supporta il controllo a runtime grazie a `@runtime_checkable`.

Con questa implementazione:

1. Qualsiasi classe con `__abs__` che restituisce un `float` è compatibile con `SupportsAbs[float]`.
2. Anche `int` è valido, perché il suo `__abs__` restituisce un `int`, che è compatibile con `float`.

Overload di Operatori

L'overloading di operatori permette a oggetti definiti dall'utente di essere compatibili con operatori infissi (`+`, `-`, `|`, ...). Questa operazione viene mal-vista in quanto porta ad abusi, confusione per chi legge il codice e molto spesso bugs. Per questo in python:

1. Non possiamo cambiare il significato degli operatori per i tipi predefiniti.
2. Non possiamo creare nuovi operatori, ma solo fare overload di quelli esistenti.
3. Alcuni operatori non possono essere sovraccaricati: `is`, `and`, `or`, `not` (ma i bitwise `&`, `|`, `~` possono).

Unary operators

1. Negazione (`-`) → Implementato da `__neg__`, inverte il segno di un numero.
2. Mantenimento del segno (`+`) → Implementato da `__pos__`, solitamente restituisce il valore invariato.
3. Complemento bitwise (`~`) → Implementato da `__invert__`, calcolato come `~x == -(x+1)`.

Per supportare questi operatori in una classe personalizzata, bisogna definire i rispettivi metodi speciali, restituendo sempre un nuovo oggetto senza modificare quello esistente.

Overload degli operatori infissi

Per fare overloading dell'operatore addizione, bisogna implementare `__add__`. Quando ci sono due tipi diversi che vengono sommati, python cerca l'operatore `__add__` nel termine di sinistra, se non lo trova come fallback cerca l'operatore `__radd__` in quello di destra e se non trova nemmeno quello viene alzata `TypeError`.

```
    # inside the Vector class

    def __add__(self, other):
        pairs = itertools.zip_longest(self, other, fillvalue=0.0)
        return Vector(a + b for a, b in pairs)

    def __radd__(self, other):
        return self + other```
```

Bisogna anche implementare gli operatori che agiscono in-place. Mentre `__add__` chiama il costruttore per definire un nuovo oggetto, `__iadd__` ritorna `self` dopo averlo modificato.

Lo stesso approccio può essere utilizzato con gli altri operatori.

Operator	Forward	Reverse	In-place	Description
`+`	`__add__`	`__radd__`	`__iadd__`	Addition or concatenation
`-`	`__sub__`	`__rsub__`	`__isub__`	Subtraction
`*`	`__mul__`	`__rmul__`	`__imul__`	Multiplication or repetition
`/`	`__truediv__`	`__rtruediv__`	`__itruediv__`	True division
`//`	`__floordiv__`	`__rfloordiv__`	`__ifloordiv__`	Floor division
`%`	`__mod__`	`__rmod__`	`__imod__`	Modulo
`divmod()`	`__divmod__`	`__rdivmod__`	`__idivmod__`	Returns tuple of floor division quotient and modulo
`**`, `pow()`	`__pow__`	`__rpow__`	`__ipow__`	Exponentiation
`@`	`__matmul__`	`__rmatmul__`	`__imatmul__`	Matrix multiplication
`&`	`__and__`	`__rand__`	`__iand__`	Bitwise and
`|`	`__or__`	`__ror__`	`__ior__`	Bitwise or
`^`	`__xor__`	`__rxor__`	`__ixor__`	Bitwise xor
`<<`	`__lshift__`	`__rlshift__`	`__ilshift__`	Bitwise shift left
`>>`	`__rshift__`	`__rrshift__`	`__irshift__`	Bitwise shift right

Overload degli operatori di confronto

In questo caso viene utilizzato lo stesso metodo anche come reverse operator. L'altra eccezione è che con `==` e `!=`, se non sono implementati i metodi, non viene alzato un `TypeError` ma si confronta l'id degli oggetti.

Group	Infix Operator	Forward Method Call	Reverse Method Call	Fallback
Equality	`a == b`	`a.__eq__(b)`	`b.__eq__(a)`	`Return id(a) == id(b)`
	`a != b`	`a.__ne__(b)`	`b.__ne__(a)`	`Return not (a == b)`
Ordering	`a > b`	`a.__gt__(b)`	`b.__lt__(a)`	`Raise TypeError`
	`a < b`	`a.__lt__(b)`	`b.__gt__(a)`	`Raise TypeError`
	`a >= b`	`a.__ge__(b)`	`b.__le__(a)`	`Raise TypeError`
	`a <= b`	`a.__le__(b)`	`b.__ge__(a)`	`Raise TypeError`

Iteratori, Generatori e Coroutines

Iteratori

Ogni collezione standard python è iterabile, supporta quindi:

1. `for` loops
2. list/dict/set comprehension
3. unpacking

Quando python deve iterare su un oggetto, chiama la funzione `iter`. La funzione controlla se l'oggetto implementa un metodo `__iter__`, se non lo implementa ma ha `__getitem__` allora `iter` genera un iteratore che prende in sequenza gli elementi con un indice che parte da zero. Se anche questo non funziona, allora python alza un `TypeError`. Questa implementazione rende tutte le sequenze come iterabili. Questo è un caso estremo di duck typing.

Si può anche usare `iter` con un oggetto *callable*: il primo argomento è l'oggetto mentre il secondo è una sentinella, un valore per il quale l'iteratore si ferma.

Esempio:

```
def d6():
    return randint(1, 6)

d6_iter = iter(d6, 1)
```

A questo punto l'iterazione proseguirà fino a che non sarà estratto `1`.

```
>>> for roll in d6_iter:
...     print(roll)
...
4
3
6
3
```

Quando l'iteratore è esaurito, esso alza `StopIteration`.

Caratteristica	Iterable	Iterator
Definizione	Un oggetto che può essere iterato (es. liste, tuple, stringhe, dizionari, insiemi).	Un oggetto che mantiene il suo stato e restituisce elementi uno alla volta.
Metodo richiesto	Deve implementare `__iter__()` che restituisce un iteratore.	Deve implementare `__iter__()` e `__next__()`.
Esempi	`list`, `tuple`, `str`, `set`, `dict`	Oggetti ottenuti con `iter(iterable)`, generatori.
Riutilizzabile?	Sì, si può iterare più volte.	No, una volta esaurito, l'iterazione finisce.
Funzionamento	Fornisce un iteratore con `iter(obj)`.	Restituisce il prossimo elemento con `next(obj)`, fino a `StopIteration`.

Gli *iterator* sono anche *iterable* ma non viceversa. E quando si definisce un iterabile, non lo si deve rendere iteratore.

Generatori

Ogni funzione che contiene `yield` nel corpo è una funzione generatrice.

```
>>> def gen_123():
...     yield 1
...     yield 2
...     yield 3
...
>>> gen_123  # doctest: +ELLIPSIS
<function gen_123 at 0x...>
```

```
>>> gen_123()   # doctest: +ELLIPSIS
<generator object gen_123 at 0x...>
>>> for i in gen_123():
...     print(i)
1
2
3
```

Lazyness

Una implementazione il più pigra possibile è vista come positiva. Ciò significa che il valore deve essere prodotto il più tardi possibile dato che salva memoria e potenzialmente dei cicli di CPU. Un modulo estremamente utile per definire dei generatori è `itertools`.

Generatori di iterazione infinita

1. **`itertools.count(start=0, step=1)`**
   - Genera numeri infiniti a partire da `start`, incrementando di `step`.
   - `count(10, 2)` → `10, 12, 14, ...`
2. **`itertools.cycle(iterable)`**
   - Itera all'infinito sugli elementi di un iterabile.
   - `cycle("AB")` → `A, B, A, B, ...`
3. **`itertools.repeat(item, times=None)`**
   - Ripete `item` indefinitamente o `times` volte.
   - `repeat(5, 3)` → `5, 5, 5`

Generatori combinatori

4. **`itertools.permutations(iterable, r=None)`**
   - Genera tutte le permutazioni possibili di `r` elementi.
   - `permutations("ABC", 2)` → `AB, AC, BA, BC, CA, CB`
5. **`itertools.combinations(iterable, r)`**
   - Genera combinazioni di `r` elementi senza ripetizione.
   - `combinations("ABC", 2)` → `AB, AC, BC`
6. **`itertools.combinations_with_replacement(iterable, r)`**
   - Simile a `combinations`, ma permette ripetizioni.
   - `combinations_with_replacement("AB", 2)` → `AA, AB, BB`
7. **`itertools.product(*iterables, repeat=1)`**
   - Genera il prodotto cartesiano tra più iterabili.
   - `product("AB", "12")` → `A1, A2, B1, B2`

Generatori per filtrare e trasformare

8. **`itertools.compress(data, selectors)`**
   - Filtra `data`, includendo solo gli elementi corrispondenti ai `True` in `selectors`.
   - `compress('ABC', [1, 0, 1])` → `A, C`
9. **`itertools.dropwhile(predicate, iterable)`**
   - Scarta elementi finché il `predicate` è vero, poi restituisce il resto.
   - `dropwhile(lambda x: x < 3, [1, 2, 3, 4])` → `3, 4`
10. **`itertools.takewhile(predicate, iterable)`**
    - Opposto di `dropwhile`: restituisce elementi finché il `predicate` è vero.
    - `takewhile(lambda x: x < 3, [1, 2, 3, 4])` → `1, 2`
11. **`itertools.filterfalse(predicate, iterable)`**
    - Come `filter()`, ma restituisce solo gli elementi per cui `predicate` è `False`.
    - `filterfalse(lambda x: x % 2, range(5))` → `0, 2, 4`
12. **`itertools.starmap(function, iterable)`**
    - Simile a `map()`, ma "spacchetta" gli elementi dell'iterabile come argomenti della funzione.
    - `starmap(pow, [(2, 5), (3, 2)])` → `32, 9`

Generatori per operazioni sugli iterabili

13. `itertools.accumulate(iterable, func=operator.add)`
    - Restituisce somme cumulative (o altri calcoli cumulativi con `func`).
    - `accumulate([1, 2, 3, 4])` → `1, 3, 6, 10`
14. `itertools.chain(*iterables)`
    - Concatena più iterabili in un unico iteratore.
    - `chain("ABC", "DEF")` → `A, B, C, D, E, F`
15. `itertools.tee(iterable, n=2)`
    - Duplica un iterabile in `n` iteratori indipendenti.
    - `tee(range(3), 2)` → Due iteratori: `(0, 1, 2), (0, 1, 2)`
16. `itertools.groupby(iterable, key=None)`
    - Raggruppa elementi consecutivi con la stessa chiave
    - ```
      for key, group in groupby("aaabbc"):
          print(key, list(group))
      ```
      Output: `a ['a', 'a', 'a'], b ['b', 'b'],` 'c ['c']

Generatori e Coroutines classiche

Una coroutine è una funzione speciale che può sospendersi e riprendere l'esecuzione, permettendo una gestione più efficiente del flusso di controllo. A differenza dei generatori, utili per generare dati per iterare, le coroutines sono consumatrici di dati.

Tuttavia, la documentazione ufficiale di Python utilizza terminologie incoerenti, portando alla distinzione tra:

1. "Classic coroutines" → generatori usati come coroutine (PEP 342)

   ```
   def coroutine_example():
       print("Avviata coroutine")
       x = yield  # Attende un valore
       print(f"Ricevuto: {x}")

   co = coroutine_example()
   next(co)  # Avvia la coroutine fino al primo `yield`
   co.send(42)  # Invia 42 alla coroutine e riprende l'esecuzione
   ```

2. "Native coroutines" → introdotte con async def in Python 3.5

Caratteristica	**Generatore**	**Coroutine**
Scopo	Produce dati per l'iterazione	Riceve dati ed esegue operazioni
Riceve dati?	No	Sì (`.send(x)`)
Sospende e riprende?	Sì (con `yield`)	Sì (con `yield` e `.send()`)
Usato per	Iterazione (es. leggere file, processare dati)	Flussi di controllo, simulazioni, gestione eventi
Esempio di uso	`for x in gen(): ...`	`co.send(value)`

Le coroutines sono garbage collected quando non ci sono più riferimenti ad esse nel codice. Per ottenere un output da una coroutine si può leggerla dalla `StopIteration` instance.

```
def averager() -> Generator[None, Optional[float], Tuple[int, float]]:
    total = 0.0
    count = 0
    average = 0.0
    while True:
        term = yield
        if isinstance(term, None):
            break
        total += term
        count += 1
```
```

```
        average = total / count
    return count, average

coro_avg = averager()
next(coro_avg)
coro_avg.send(10)
coro_avg.send(30)
coro_avg.send(6.5)
try:
   coro_avg.send(None)
except StopIteration as exc:
   result = exc.value

result
# (3, 15.5)
```

Context Manager

In Python, un context manager è un oggetto che gestisce l'allocazione e il rilascio di risorse in modo automatico, garantendo che le operazioni di pulizia vengano eseguite correttamente. Il modo più comune per usare un context manager è con l'istruzione `with`. Normalmente si usa con i file ma si può estendere a qualsiasi oggetto potenzialmente. Affinché un oggetto sia gestibile tramite context manager esso deve definire i metodi `__enter__` (che ritorna l'oggetto che viene gestito nel contesto) e `__exit__`.

```
class MioContextManager:
    def __enter__(self):
        print("Entrato nel contesto")
        return self  # L'oggetto stesso viene restituito

    def __exit__(self, exc_type, exc_value, traceback):
        print("Uscito dal contesto")

# Usarlo con 'with'
with MioContextManager():
    print("Dentro il blocco with")
```

Con gli argomenti di `__exit__` che gestiscono una possibile eccezione che causa l'uscita dal context manager.

Si può anche sfruttare `contextlib` che contiene moltissimi decoratori e classi.

```
from contextlib import contextmanager

@contextmanager
def mio_context():
    print("Entrato nel contesto")
    yield  # Qui si esegue il codice nel blocco `with`
    print("Uscito dal contesto")

with mio_context():
    print("Dentro il blocco with")
```

Il problema di questa implementazione è che non viene gestita alcuna eccezione all'interno del contesto. Per gestirla si può inserire `yield` in una logica `try/except/finally` (prova a eseguire il blocco `with`, gestisci l'eccezione se necessario, infine fai la pulizia).

Infine le funzioni generatore decorate con `@contextmanager` possono essere usate a loro volte come decoratori.

Clausola `else`

La clausola `else` può essere utilizzata non solo con `if`, ma anche con `for`, `while` e `try`. La logica è la seguente:

- **for/else**: Eseguito solo se il ciclo `for` termina senza un `break`.

```
for item in my_list:
  if item.flavor == 'banana':
    break
else:
  raise ValueError('No banana flavor found!')
```

- **while/else**: Eseguito solo se il ciclo `while` termina perché la condizione è falsa, non a causa di un `break`.

```
i = 0
while i < 3:
  print(i)
  i += 1
else:
  print("Ciclo completato senza interruzioni.")
```

- **try/else**: Eseguito solo se nessuna eccezione è sollevata nel blocco `try`.

```
try:
  dangerous_call()
except OSError:
  log('OSError...')
else:
  after_call()
```

In tutti i casi, l'`else` viene saltato se ci sono eccezioni o comandi come `return`, `break` o `continue`.