Complementi di programmazione

Appunti redatti

Iacopo Ruzzier

Ultimo aggiornamento: 17 settembre 2024

Indice

Parte I
introduzione ai linguaggi dinamici

1 cenni storici

gli sviluppi nei linguaggi rispecchiano esigenze ed epoca storica **sistemi mainframe** → applicazioni per calcolo scientifico, interfacce testuali, amministrazione in forma di script per automatizzare compiti C → prestazioni elevate, ma meno portabile Assembly → veloce, scarsamente portabile, ma interfacciamento diretto con hw shell → più lento ma molto portabile, ideale per manutenzione e scripting l'avanzamento HW e la complessità in aumento fanno emergere il concetto di *"linguaggio ad alto livello general purpose"* → Python, Ruby, Java

semplicità, oop, portabilità

1991: **WWW** → emergono linguaggi per Web App (PHP, JavaScript), se ne adattano altri esistenti (C, Perl)

2 linguaggi statici vs dinamici

linguaggi statici

tipicamente compilati

tipi delle variabili verificati a tempo di **compilazione**

linguaggi dinamici

tipicamente interpretati

tipi verificati a tempo di **esecuzione**

pro

flessibilità

maggior facilità di debug, sviluppo più rapido

(tipicamente) sintassi più semplice

contro

prestazioni inferiori (overhead significativo per l'allocazione di memoria e verifica tipi a runtime)

errori a runtime → debug problematico in applicazioni critiche

maggiore difficoltà di manutenzione (codice meno auto-documentante)

maggiore probabilità di bug

può avere una fase di compilazione in formati intermedi indipendenti dall'architettura, che poi vengono interpretati

l'interprete

si serve di funzioni interne per gestire memoria ed errori in automatico

tipizzazione dinamica

Metaprogramming → capacità di auto-analisi e modifica del codice

librerie esterne facilmente utilizzabili

Parte II
tipizzazione

un programma manipola **dati** attraverso **istruzioni**

istruzioni → indicano l'operazione da compiere su dati di un certo tipo

tipo → indica cosa rappresenta il dato (quindi le operazioni permesse)

nel programma, un dato viene identificato da un nome, ma il programmatore deve anche conoscerne

tipo → per non indicare operazioni sbagliate

dimensione → per non saturare la memoria

scope e tempo di vita → per sapere quando usare l'entità

la tipizzazione permette di

scoprire codice **illecito/senza senso**

ottimizzare l'**esecuzione**

aiuta nell'**astrazione** (tralascio i dettagli di memorizzazione del dato a basso livello)

aiuta a definire **interfacce**

3 type check nei linguaggi dinamici

permette maggiore flessibilità, ma implica la possibilità di molti errori e comportamenti non previsti

4 tipizzazione forte

sono imposte rigide regole sulla **conversione** dei tipi di dato e sulla **compatibilità tra tipi**
obbligo di dichiarare i tipi
conversioni esplicite

5 tipizzazione debole

consente conversioni implicite e operazioni tra tipi incongruenti
esempio:

```
a = 3
b = 58
a + b = ?
```

in C, si passa all'aritmetica dei puntatori → `op. tra indirizzi`
in Java, i numeri sono convertiti in stringhe → `358`
in Perl, le stringhe numeriche sono convertite in numeri → `61`
in Python otteniamo un errore!
non c'è obbligo di dichiarazione di tipo

6 tipizzazione safe

quando il linguaggio impedisce che un operazione di casting implicito causi un crash
Java, Perl

7 tipizzazione unsafe

quando il linguaggio **non** impedisce il crash → C (se il puntatore è fuori range), **Python**

8 tipizzazione in Python

python è completamente object oriented → ogni variabile è oggetto, **anche i tipi primitivi**
ogni oggetto ha a disposizione metodi e attributi della sua classe + **quelli ereditati**
tramite ereditarietà si possono realizzare comportamenti diversi per uno stesso metodo - è necessario **capire quale metodo chiamare** (a tempo di compilazione/runtime), controllando che la classe abbia il metodo, oppure passare alle superclassi

```
class Moto(Veicolo):
  def getRoute():
    return 2

class Auto(Veicolo):
  def getRoute():
    return 3

Veicolo v = getVeicolo() # ritorno Auto o Moto
v.getRoute() # risultato??
```

è dunque necessario conoscere il tipo dell'oggetto...?

8.1 duck typing

in Python, **non controllo il tipo!** ma solo se l'oggetto possiede il metodo

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

```
def calcola(a,b,c):
  return (a+b)*c

e1 = calcola(1,2,3)
# 9
e2 = calcola([1,2,3],[4,5,6],2)
# [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
e3 = calcola('mele ', 'e arance ',3)
# 'mele e arance mele e arance mele e arance '
```

*è sufficiente che riceva parametri che supportano le operazioni + e *!*

Parte III
Python

alto livello, dinamico, interpretato
compilatore+interprete, Bytecode, PVM (Python Virtual Machine)
shebang: `#!/usr/bin/python`
documentazione integrata con `help(something)`

9 tipi di base

numerics: interi, float, complessi (`z = 10 + 20j / z.real / z.imag`), booleani
operazioni aritmetiche std, divisione intera, resto intero, valore assoluto, complesso coniugato, potenza, arrotondamenti vari
sequenze e **stringhe**: intesi come sequenza ordinata di elementi
stringhe (immutabili), liste `[]`, tuple `()` (immutabili a differenza delle liste)
accesso ad un elemento con bracket notation, concatenazione, ripetizione, lunghezza con `len()`
operazioni su stringhe: `lower, upper, count(substr), find(substr), replace(sub1,sub2),join,split,...`
operazioni su liste: `append, insert, pop, sort, sorted, len, in, remove, del...`
slicing mediante bracket notation (per entrambi i tipi)
deep e shallow copy di liste: copiando tramite `b = a` si copia il riferimento allo stesso oggetto, e dunque si modifica lo stesso oggetto → per una deep copy si usa un modulo esterno, o lo slicing `b = a[:]`
operazioni su tuple: tutti quelli delle liste **tranne quelli di modifica**

9.1 list comprehension

```
[expr for var in sequence]
# esempio
squares = [x**2 for x in range(10)]
```

set: insieme non ordinato (no accesso ad indice) di oggetti non duplicati
creato con `set()` o `\{elem,elem\}`
`set([list])` per crearlo da una lista
comodi per eliminare i duplicati es. da una lista, o per test di appartenenza (più efficiente che operare su liste)
operazioni: `len(S),el in/not in S,S.isdisjoint(S2),`
`S1 |/&/- S2 / S1.union/intersection/difference(S2),...`
dictionary: simile a hashmap (coppie key-value)

i valori possono essere qualsiasi, le chiavi **devono essere oggetti immutabili** es. numerics, stringhe, tuple

chiavi uniche nel dizionario

accesso ai valori con bracket notation → si può usare per **aggiungere/modificare valori**

metodi: `.keys()`, `.values()`, `.items()` → lista con le chiavi, lista con i valori, lista di tuple; `del dict[key]`, `.pop(key)`, `key in dict`, `.has_key(key)`

10 costrutti di base

i blocchi sono indentati, non usano graffe

non si usa `;` a fine istruzione, solo per multiple istruzioni su una riga

possibile assegnamento multiplo es. `a,b = 1,2`

`import module` o `from module import something` o `import module as alias`

`# commenti`

```
if cond:
  block
elif cond: # opzionale
  block
else: # opzionale
  block

while cond:
  block
else: # opzionale
  block

for elem in iterable: # qualsiasi iterabile
  block
else: # opzionale
  block
```

`break`, `continue` cambiano il flusso del ciclo; `break` non esegue l'eventuale `else`

a seconda dell'iterabile che scelgo per il for, accadono cose diverse

`range()` genera sequenze di numeri ordinate

liste e tuple: riferimento ad un oggetto ad ogni iterazione

stringhe: 1 carattere per iterazione

set e dictionary: 1 elemento/chiave per iterazione, **non garantito l'ordine!**

`enumerate()` ritorna una tupla `(index, obj)` per ogni elemento della sequenza passata → si può usare `for i,o in enumerate(a):`

si usano `iter()`, `next()` come wrapper più comodi per chiamare i metodi

11 funzioni

non innestate in una classe (≠ metodi)

```
def foo(arg1, arg2):
  block
  return statement
```

parametri opzionali con valore di default se non passati: si mettono per ultimi

passaggio di argomenti: **positional arguments** (senza specificare i parametri, conta l'ordine) oppure **keyword arguments** (ordine indifferente)

operatori * e ** per "spacchettare" liste e dizionari in positional/keyword arguments

possono essere usati anche nella definizione, in questo caso tutti i parametri posizionali sono condensati in una tupla, mentre i keyword arguments in un dizionario

utile se non conosco a priori il numero di parametri necessari!

posso comunque avere parametri espliciti

11.1 ricorsione

possibile come negli altri linguaggi, a patto di un criterio di arresto

```
# esempio - funzione per appiattire una lista di liste
def flatten(l):
  res = []

  if not hasattr(l,'__iter__'):
    return [l]

  for o in l:
    res.extend(flatten(o))
  return res
```

11.2 programmazione di ordine superiore

la usano i linguaggi che permettono di passare funzioni come parametro/ritornarle come risultato di altre funzioni

```
# passaggio come parametro
def print_function_result(func):
  print(f'il risultato \`e {func()}')

# uso come valore di ritorno
def get_help_print_function(lang):
  def eng_help():
    print('help')

  def ita_help():
    print('aiuto')

  if lang == 'eng':
    return eng_help # ASSENZA DI ()
  else:
    return ita_help # ASSENZA DI ()
```

11.2.1 closure

le funzioni innestate **possono accedere alle variabili delle funzioni madri** → sono incapsulate nelle funzioni figlie in modo da poter essere usate anche quando **termina la funzione madre**

```
def print_msg(msg):
  def printer():
    print(msg) # NON FA PARTE DELLA FUNZIONE MA DELLA MADRE!

  return printer
```

11.2.2 lambda

anche **funzione anonima**

```
lambda args: return_value
#esempio per ritornare il quadrato
lambda x:x**2
```

11.2.3 `map()`

funzione che spiega bene il concetto di fz. di ordine superiore
input: **funzione e sequenza di oggetti**
output: **sequenza (lista) sui cui elementi è stata applicata la funzione**
esempio con le lambda:

```
map(lambda x:x**2, [1,2,3]) # [1,4,9]
```

11.3 generatori

in generale, un iterable è qualsiasi oggetto che ha un metodo `__iter__()` che ritorna un iteratore → implica la costruzione di una classe con tale metodo!

un generatore permette di semplificare il processo, generando **un valore alla volta** invece di dover istanziare un oggetto che contenga l'intera sequenza

possono essere espressioni o funzioni

11.3.1 funzioni generatrici

ritornano una serie di valori invece che uno solo

si creano attraverso il comando `yield value`

vengono trattate da python come un generatore, e sono dotate implicitamente di metodi `__iter__` e `__next__`

```
# esempio: range "fai da te"
def my_range(stop):
  i = 0
  while i < stop:
    yield i
    i += 1
```

11.3.2 espressioni generatrici

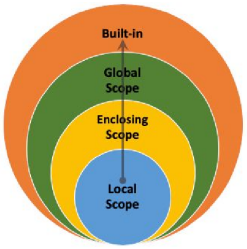
più brevi da scrivere

```
(expr for var in sequence)
# esempio
(x**2 for x in range(10))
```

simili alle list comprehension

11.4 scope

- **built-in**: entità sempre disponibili (es. `print()`)
- **global**: e. definite nel codice fuori da qualsiasi blocco
- **enclosing**: e. incapsulate per effetto della closure
- **local**: e. definite in un blocco



la risoluzione segue **local** → **enclosing** → **global** → **built-in**

esiste lo **shadowing**

per forzare l'uso di entità globali, si usa `global var`

11.4.1 argomenti delle funzioni

gli argomenti delle funzioni sono **local**

viene passato il **riferimento all'oggetto - tutto è un oggetto**

quando riassegnamo ad es. un numero o stringa alla variabile locale, stiamo cambiando l'oggetto a cui punta, dunque non modifichiamo la variabile esterna

quando operiamo ad es. su una lista con un metodo che va a modificarla, **cambia anche quella esterna! → stiamo operando sullo stesso oggetto (il riferimento è lo stesso)**

12 eccezioni

come in Java e altri linguaggi, nell'ambiente di runtime viene definita una classe madre `Exception` → ogni anomalia è associata ad una sua sottoclasse

le eccezioni sono sollevate

automaticamente (in seguito ad anomalie a runtime es. divisione per zero)

manualmente dal programmatore

in caso di eccezione viene eseguito codice di gestione; se non è presente un gestore specifico, verrà stampato lo stack delle chiamate e si uscirà dal programma

```
try:
  codice
except Classe_eccezione as identificativo:
  codice #eseguito se nel try viene sollevata
except Altraclasse_eccezione as identificativo:
  codice #eseguito se nel try viene sollevata
except:
  codice # eseguito se nel try viene sollevata un'eccezione non
                  # catturata esplicitamente prima
else:
  codice # eseguito se non vengono sollevate eccezioni
finally:
  codice # eseguito sempre

# per SOLLEVARE ECCEZIONI:
raise NameException(arg[,arg]...)
```

13 decoratori

usati per modificare le funzionalità di altre funzioni

recap: in python **tutto è un oggetto**, anche le funzioni → una volta definite, possiamo (senza parentesi) **assegnarle a variabili, passarle come parametro ad altre funzioni, usarle come valore di ritorno**, o anche **definirle all'interno di altre funzioni** quando metto `()` **eseguo la funzione**, senza sto semplicemente **passando l'oggetto**

i decoratori permettono di eseguire codice **prima** e **dopo** una funzione

```
def new_decorator(foo): # il decoratore
  def wrapTheFoo():
    print("Before executing foo()")
    foo()
    print("After executing foo()")

  return wrapTheFoo

def dec_foo(): # funzione che necessita di decoratore
  print("I am the foo")

dec_foo()
# I am the foo

dec_foo = new_decorator(dec_foo) # ora la funzione \`e wrappata

dec_foo()
# Before executing foo()
# I am the foo
# After executing foo()
```

per una scrittura più compatta:

```
def new_dec(foo):
  def wrap():
    # qualcosa prima
    foo()
    # qualcosa dopo

  return wrap

@new_dec                # alternativo a dec_foo = new_dec(dec_foo)
def dec_foo():
  # ...
```

per decorare funzioni con argomenti (forma più generica):

```
def another_dec(foo):
  def wrapper(*args, **kwargs):
    # prima
    foo(*args, **kwargs)
    # dopo

  return wrapper

@another_dec
def another_foo(arg1,...):
  # ...
```

14 classi e oggetti

recap OOP
classe → definisce variabili e metodi contenuti
oggetto → istanza di una classe
astrazione → non guardo dettagli implementativi (da utilizzatore)
incapsulamento → protezione attributi interni
ereditarietà → riuso del codice
polimorfismo → adattabilità

```
class NomeClasse: # PascalCase come convenzione
  i = 0 # attributo di classe
  def metodo(self, arg1, arg2): # metodo
    # ...

var = NomeClasse() # istanza

NomeClasse.i # accesso ad attributi di classe e metodi
var.i # uguale

var.i = 1 # creo un attributo di istanza che shadowa quello di classe
```

la classe è **essa stessa un oggetto** → posso usarla per accedere ad attributi di classe e metodi con dot notation
gli attributi di classe appartengono all'oggetto classe
i metodi hanno come primo argomento obbligatorio il **riferimento all'oggetto stesso (per convenzione 'self')**, che permette di accedere ai metodi/attributi dell'oggetto
per chiamare i metodi **non si passa 'self'**

14.1 costruttore

```
class NomeClasse:
  def __init__(self[,args...]):
    self.a = 1 # attributo di istanza
```

chiamato automaticamente quando istanzio l'oggetto
usato per inizializzare variabili **di istanza** (**non di classe!**)

14.2 metodi static e class

chiamati come altri metodi ma **usando l'oggetto class**

```
class NomeClasse:
  @staticmethod
  def static_method(x):
    # ...

  @classmethod
  def class_method(self,x): #self \`e l'OGGETTO CLASSE!
    # ...
```

esempio d'uso:

```
class Date(object):
  @classmethod
  def from_string(self, date_as_string):
    day, month, year = map(int, date_as_string.split('-'))
    date1 = self(day, month, year) # STO ISTANZIANDO LA CLASSE!
    return date1

  @staticmethod
  def is_date_valid(date_as_string):
    day, month, year = map(int, date_as_string.split('-'))
    return day <= 31 and month <= 12 and year <= 3999
```

14.3 ciclo di inizializzazione

quando istanzio un oggetto, l'interprete chiama un metodo per
creare l'oggetto → `__new__`, di solito si usa quello di default della classe madre `object`

```
# esempio di ridefinizione (uso cls dove di solito uso self)
class ClassName:
  def __new__(cls, *args, **kwargs):
    print("Ciao io sono __new__")
    return object.__new__(cls)
```

inizializzare l'oggetto → uso `__init__`

14.4 metodi/attributi privati

in python non esiste privacy! posso comunque accedere a tutti i metodi e attributi in qualche modo
entità che **iniziano con __ e terminano con massimo 1 _** vengono considerati privati
python rinomina le entità private come `_NomeClasse__NomeEntita` per evitare di potervi accedere tramite `.NomeEntita`

14.5 metodi speciali

iniziano e terminano con `__`, hanno una semantica speciale e vengono chiamati in automatico in certe situazioni

```
- __init__                  # costruttore
- __repr__                  # rappresentazione dell'oggetto
- __getattr__          # per emulare l'accesso ad una attributo:
                              # a.x --> a.__getattr__('x')
- __getitem__          # per emulare un tipo contenitore (tipo
```

```
                                # le liste): a[x] -> a.__getitem__('x')
- __add__, __mul__, etc. # per emulare somme, prodotti etc.
- __call__                # l'oggetto pu\`o essere chiamato come una funzione
- __str__                 # converte un oggetto in una stringa
- __del__                 # distrugge la classe ( metodo distruttore )
- __{ eq , gt, ge, lt, le }__ # verifica se un valore \`e
                                # { uguale, ..., ... } ad un altro
- __setitem__          # operatore [] in uscita (assegnazione)
```

i metodi corrispondono ad **operatori python** → quando uso gli operatori in realtà sto andando a chiamare questi metodi!

→ è possibile avere comportamenti specializzati tramite **overloading** dei metodi di default (**polimorfismo**)

`__getattr__` e `__setattr__` (usati per gli operatori . e =)

li sovrascrivo ad es. per **aggiornare un altro campo in conseguenza di un settaggio**, oppure per **proteggere l'accesso ad un attributo**

```
class Prova:
  def __setattr__(self,campo,valore):
    if campo == 'saldo':
      super().__setattr__(campo,valore)
      self.movimenti.append(valore)

  def __getattr__(self,campo):
    if campo == 'saldo':
      raise AttributeError('campo non accessibile')
    else:
      return super().__getattr__(campo)

a = Prova()
a.saldo # AttributeError
```

14.6 getter e setter

anche in python è possibile definire dei metodi specifici per accedere/modificare attributi privati, ma definire i propri metodi o usare i metodi speciali non è la soluzione preferita → si usano dei **decoratori built-in specifici**

`@property`: annota il metodo getter → si accede all'attributo attraverso il nome del metodo, che verrà trattato **come un attributo**

`@<attr_nome>.setter`: annota il metodo setter

```
class Prova:
  def __init__(self,x):
    self.__x_ = x

  @property
  def x(self):
    return self.__x_

  @x.setter
  def x(self,new_x):
    self.__x_ = new_x

a = Prova(3)
a.x # 3
a.x = 2 # non porta ad errori
```

14.7 ereditarietà

permette di far ereditare metodi e attributi di una classe madre alle classi figlie

python permette anche quella **multipla** → quando deve decidere quale implementazione di un metodo utilizzare, se il metodo non è definito nella classe l'interprete passa in rassegna le classi madre **da sinistra a destra**

```
class Automobile:
  def __init__(self):
    self.posizione = 0

  def muoviti(self,l):
    self.posizione += l

class FuoriStrada(Automobile): # eredita da Automobile metodi e attributi
  def ridotte(self):
    self.rapporto = 0.001

  def muoviti(self, l): # sovrascrivo un metodo di Automobile
    l = l * self.rapporto
    super().muoviti(l) # chiamo il metodo della classe madre

class Barca:
  def affonda(self):
    self.affondato = True

  def muoviti(self,l):
    if self.affondato:
      return False

class Anfibio(Barca, Automobile): # eredita da entrambi
  pass # se chiamo muoviti(), verr\`a chiamato quello di Barca
```

verifico il Method Resolution Order (ordine di risoluzione) accedendo all'attributo `__mro__` (in sola lettura - **non cambia una volta inizializzato**)o chiamando `mro()`

`__bases__` indica le classi base e **può essere modificato** → dopo la modifica viene **ricalcolato** `__mro__`

il costruttore chiamato di default (se non definito nella classe) è quello della prima classe madre → per **invocarli entrambi bisogna esplicitarlo**

`isinstance(ist,class)`: per verificare il tipo di un'istanza

`issubclass(x,y)`: per verificare se `x` è sottoclasse di `y`

14.8 accesso agli attributi

un oggetto può avere attributi self / di classe / ereditati / metodi di accesso "fake" agli attributi

come in altri linguaggi, per gli attributi di classe si parte dal livello più basso e si sale via via attraverso le classi madre

attributi di istanza: dipende dai vari `__init__` e da come vengono chiamati

```
class Animale:
  x = 0; y = 0
  def __init__(self):
    self.a = 1
    self.b = 1

class Cane(Animale):
  x = 10 # sovrascrive l'attributo di classe
  def __init__(self):
    self.a = 2 # viene sovrascritto dal costruttore di Animale subito sotto
    super().__init__()
    self.b = 2 # sovrascrive il costruttore appena chiamato!
```

```
class Gatto(Animale):
  def __init__(self):
    self.a = 2
    # se cerco b avr\`o un errore! non sto chiamando il costruttore madre
```

priorità ad attributi di **istanza su quelli di classe**

```
class Prova:
  x = 0; y = 0
  def __init__(self):
    self.x = 1 # x sar\`a di istanza, y di classe
```

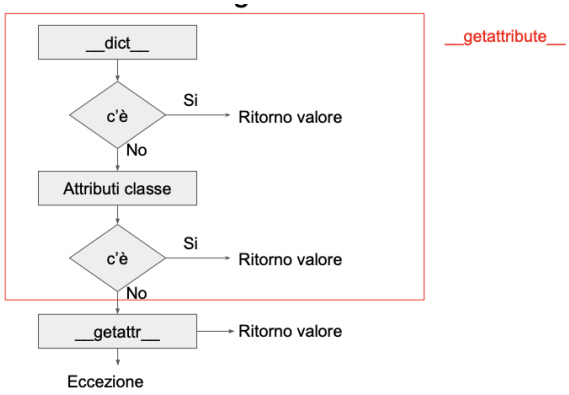
in particolare (dalla documentazione):

> "A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes"

questo namespace è l'attributo `__dict__`: contiene tutti gli attributi di istanza, ed è aggiornato man mano che assegno valori (anche quando lo faccio in `__init__()`)

è il metodo `__getattribute__()` a controllare in `__dict__` e tra gli attributi di classe

se non trovo un attributo in `__dict__` o tra gli attributi di classe, passo a `__getattr__()` (di default solleva un'eccezione)



dovrei dunque modificare `__getattribute__()` e non `__getattr__()` per imporre condizioni di accesso particolari (`__getattr__()` non esegue proprio se il nome dell'attributo viene trovato prima):

```
class Prova:
  def __init__(self):
    self.saldo = 0

  def __getattribute__(self, campo): # al posto di __getattr__()
    if campo == 'saldo':
      raise AttributeError('campo non accessibile')
    else:
      return super().__getattribute__(campo)
```

casi particolari in cui non viene chiamato `__getattribute__()`: quando l'interprete chiama attributi speciali

es.: nei cicli `__iter__` e `__next__`, se usiamo `len()` che chiama `__len__`

```
class MyList(list):
  def __getattribute__(self, item):
    print(f'getattribute {item}')
    return super().__getattribute__(item)
```

```
  def foo(self):
    print('mi hai chiamato')
    pass

l = MyList()
l.foo                   # getattribute foo
l.foo()          # getattribute foo
                        # mi hai chiamato
l.__len__          # getattribute __len__
len(l)                  # NO OUTPUT - NON PASSA PER __getattribute__
```

nota: i metodi sono considerati come attributi → coerente con il concetto che **le funzioni sono anch'esse oggetti**

`__setattr__()`: analogo per settare gli attributi (∄ `__setattribute__()`) → di fatto **modifica/aggiunge entry in** `__dict__`, e viene invocato anche in `__init__()`

ecco perché `instance.attr = x` **non modifica l'attributo di classe**

per evitare limitazioni nell'inizializzazione, in `__init__()` accedo all'attributo da inizializzare tramite bracket notation:

```
class Prova:
  def __init__(self):
    self.__dict__['saldo'] = 0

  def __setattr__(self,campo,valore):
    if campo == 'saldo':
      raise AttributeError('campo non modificabile direttamente')
    else:
      return super().__setattr__(campo,valore)
```

`@property`: è un decoratore → è considerato attributo di istanza/di classe/metodo? a che punto viene controllato?

quello che fa questo decoratore è **ritornare un descrittore**: un attributo il cui valore è un oggetto che implementa uno o più metodi tra `__get__()`, `__set__()`, `__delete__()`

→ l'interprete **chiama il corrispettivo metodo** per accedere/modificare/eliminare l'attributo

da https://amir.rachum.com/descriptors/:

> Define any of these methods and an object is considered a descriptor and can override default behavior upon being looked up as an attribute. [...] (...)

> If an object defines both `__get__()` and `__set__()`, it is considered a data descriptor. Descriptors that only define `__get__()` are called non-data descriptors (they are typically used for methods but other uses are possible). (...)

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance's dictionary. **If an instance's dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If an instance's dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.**

To make a read-only data descriptor, define both `__get__()` and `__set__()` with the `__set__()` raising an AttributeError when called. Defining the `__set__()` method with an exception raising placeholder is enough to make it a data descriptor. [...]

```
class Desc:
  def __get__(self,istanza,owner):
    pass
  # self: istanza di Desc
  # istanza: oggetto sul quale viene richiesto l'accesso all'attributo
  # owner: la classe dell'oggetto sul quale viene richiesto l'accesso

  def __set__(self,istanza,value):
```

```
    pass
  # value: nuovo valore da assegnare

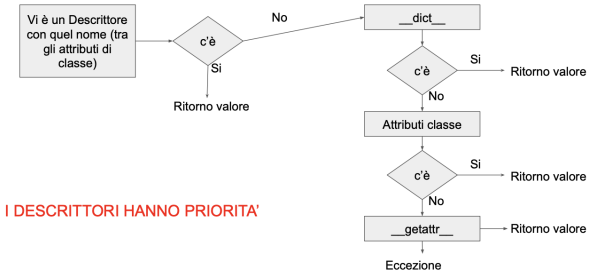
  def __delete__(self,istanza):
    pass

class Prova:
  i = Desc()

p = Prova()
p.i # attributo di classe
```

caso limite di implementazione "a mano" di un descrittore: quando assegno un descrittore ad un attributo di classe, l'oggetto sarà lo stesso per tutte le istanze → per averlo diverso ad ogni istanza, possiamo implementare i metodi del descrittore andando a scrivere/leggere da `__dict__`, ma in questo modo **dal secondo accesso non andrei mai a verificare gli attributi di classe e non vedrei il descrittore (mi fermo sempre nel dizionario)**

→ in python, **i descrittori hanno priorità**



14.9 metaclassi

classi in cui le istanze sono a loro volta classi (reminder: **in python tutto è oggetto, anche una classe**)

metaclasse madre: `type` → tutte le classi sono sue istanze

internamente python crea le classi (le definizioni) istanziando la metaclasse `type`

si può **estendere** e creare una propria metaclasse, ridefinendo `__init__()` e `__new__()` per modificare il comportamento di base delle classi

```
class Prova(metaclass=MyMetaClass): # per specificare la metaclasse
  # ...
```

14.9.1 `type()`

con 1 argomento ritorna il tipo dell'oggetto

con 3 **ritorna una nuova classe**!! (equivalente alla definizione solita)

```
class X(Y):
  a = 1
# equivale a
type('X',(Y),dict(a=1))

class MyMetaClass(type):
  def __new__(cls, classname, super, classdict):
    return super().__new__(cls, classname, super, classdict)

  def __init__(self):
    super().__init__()
```

utili per avere meccanismi simili all'ereditarietà, ma anche per creare classi in modo diverso a runtime (es. classi con attributi di classe che hanno nomi definiti dall'utente)

14.10 `__call__()`

metodo speciale chiamato quando un oggetto viene invocato come una funzione
implica ad es. che **anche una classe può essere un decoratore**

```
class dec:
  def __init__(self, f):
    self.f = f

  def __call__(self):
    pass

@dec
def foo():
  # ...
```

`foo` viene passato ad `__init__`, ed il nuovo oggetto ritornato verrà associato a `foo`
→ quando verrà invocata `foo` verrà chiamata `__call__`

14.11 classi astratte

classe che definisce almeno un metodo astratto, ossia senza implementazione
questi vanno definiti nelle sottoclassi
possibile in python tramite il modulo Abstract Base Classes (ABC) ed il suo decoratore `@abstractmethod`

```
from abc import ABC, abstractmethod

class Astratta(ABC): # deve ereditare solo da ABC e altre classi astratte!
  @abstractmethod
  def metodo_astratto(self):
    pass
```

15 programmazione funzionale

vari paradigmi:
imperativo: programma come serie di istruzioni eseguite una dopo l'altra
ad oggetti: programma come insieme di oggetti che interagiscono tramite metodi
funzionale: in stile matematico, **si dichiara che funzione usare** per ottenere il **risultato**
logico: problema sotto forma di vincoli, si cerca una soluzione che soddisfi i vincoli
il paradigma funzionale è in stile **dichiarativo** → dichiaro che valore deve assumere un dato nome (tipicamente il risultato di una funzione)
usa *"espressioni"* (come in matematica), che vengono valutate per portare al risultato
le funzioni definite in questo paradigma sono definite **Pure**:
stesso input → stesso risultato
non modificano gli argomenti in input
non si usano cicli (ma la ricorsione)
usa la programmazione di ordine superiore (passaggio o ritorno di funzioni)
i nomi (variabili in altri paradigmi) sono immutabili
python permette tutto ciò!
funzioni pure → basta non modificare i parametri, al limite copiarli
possibili la ricorsione e la programmazione di ordine superiore
il programmatore può consapevolmente non modificare una variabile

```
def fattoriale(x):
  return 1 if x <= 1 else return x * fattoriale(x-1) #non modifico argomenti!

fact10 = fattoriale(10) # da qui non devo cambiare fact10!
```

la ricorsione può **saturare lo stack** - python ha un limite di chiamate ricorsive permesse (**modificabile**)

```
import sys
sys.setrecursionlimit(<max_chiamate>)
```

vantaggi:
debug più facile → risultato di una funzione dato solo dagli argomenti!
funzioni tendenzialmente più semplici → assenza di cicli o flussi complessi
struttura più interpretabile da un compilatore → maggiori ottimizzazioni, parallelizzazioni automatiche, **lazy evaluation** (esecuzione solo al bisogno del risultato)
svantaggi:
cambio di mentalità ed abitudini
molto meno utilizzata rispetto a imperativa o OO

15.1 strumenti python per la programmazione funzionale

15.1.1 list comprehension

```
[ expr for value in iterable if condition ]

[ k*k for k in range(1, n+1) ]
[ k for k in range(1, n+1) if n % k == 0 ]
# set
{ k*k for k in range(1, n+1) }
# dizionario
{ k : k*k for k in range(1, n+1) }
```

costrutto sintattico per creare liste/dizionari tramite for innestato
il risultato è una lista/dizionario risultante da tutti i risultati di `expr` calcolati con ogni `value` in `iterable` (se `condition` è vera)
possono anche esserci for multipli, con nomi letti da sinistra verso destra

```
[ x for iter2 in iter1 for x in iter2 ] # ok
[ x for x in iter2 for iter2 in iter1 ] # errore
```

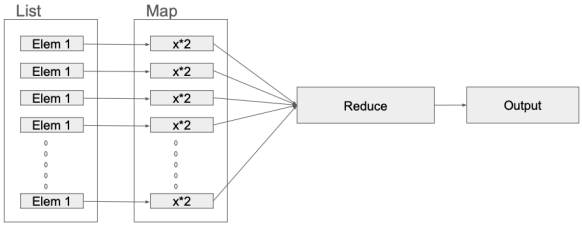
15.1.2 lambda

fondamento della programmazione funzionale, solo argomenti e risultato
numero qualsiasi di argomenti, una sola espressione che viene valutata e restituita
un programma scritto con sole lambda è **Turing-completo**

```
lambda args: expr
# usi di esempio
filter(lambda x: (x % 2 != 0), iter1)
map(lambda x: x*2, iter1)
```

15.1.3 map/reduce

paradigma distribuito per la computazione dei dati, diviso in
mappatura dei dati: filtri ed elaborazioni sui singoli elementi
riduzione: aggregazione dei risultati della prima fase
essendo distribuito non si può assumere la sequenza dei dati
la riduzione può avvenire in più step
utile in db distribuiti in cui i dati sono in nodi diversi e la computazione non può avvenire in un unico modo
`reduce()` (dalla libreria `functools`): esegue in locale ma è comunque utile in ambienti distribuiti → applica ripetutamente una funzione di due argomenti sugli elementi di una sequenza per ridurla ad un unico valore
ad ogni iterazione il primo argomento è il risultato delle passate precedenti e il secondo un elemento (non ancora processato) della lista → per la prima passata si usa il primo elemento della lista come elemento non processato, oppure si specifica che valore usare come primo risultato




```
# sommo tutti gli el
reduce(lambda x,y: x+y, [1,2,3,4])
# 10
# creo un set da una lista
reduce(lambda x,y: x.union(set([y])), [1,2,3,4], set())
# {1, 2, 3, 4}
```

efficace sia quando la computazione può essere distribuita, sia quando è trattata come flusso → **riduce risorse computazionali**

15.1.4 eval

funzione che valuta un'espressione **passata sotto forma di stringa** → l'interprete esegue la stringa **come fosse codice del programma** e ritorna il risultato dell'espressione valutata

posso anche passare variabili dal programma "principale", sia come globali che locali al codice della stringa

```
eval(expr, globals, locals)
```

```
eval('print("ciao")')
# ciao
a = 10: b = 20
eval("print(f'c: {c}, d: {d}')", {'c': a}, {'d': b})
```

può valutare qualsiasi espressione → problemi di sicurezza → è bene non far valutare stringhe generiche, oppure settare accuratamente globals e locals per vietare accessi a funzioni critiche

15.1.5 exec

simile ad `eval`, ma può anche **eseguire istruzioni** (es. `import`, dunque non solo espressioni)

può eseguire **codice compilato** → riferito al formato intermedio (**bytecode**) che usa anche python

15.1.6 compile

compila una stringa o file in bytecode

`mode`: `exec` se contiene istruzioni, `eval` se una singola espressione, `single` se una singola istruzione interattiva

```
compile(source,file,mode) # uso source o file
```

15.1.7 functools

modulo per la programmazione funzionale, fornisce funzioni di ordine superiore per eseguire operazioni comuni

fornisce anche le classi `partial` e `partialmethod` per rappresentare le funzioni

`partial`: rappresenta funzioni in cui alcuni argomenti sono **definiti prima di chiamarla**

```
def somma(x,y):
  return x + y

sommapartial = functools.partial(somma, y=1)
# posso chiamare sommapartial passando soltanto la x
```

`partialmethod`: analogo per metodi di classe → richiama un altro metodo con parametri preimpostati

```
class A:
  def print_str(self,s):
    print(s)

  print_ciao = functools.partialmethod(print_str, s='ciao')
```

`cmp_to_key`: trasforma una **funzione di comparazione** in una **funzione chiave**

> A comparison function is any callable that accepts two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

```
sorted([(1,2),(2,1)], key=lambda x: x[1]) # esempio di key function

# caso complesso
class Data:
  def __init__(self,anno,mese,giorno):
    # ...

# scrivo una funzione di confronto
def cmp_date(x,y):
  if x.anno < y.anno:
    return -1
  elif x.anno > y.anno:
    return 1
  # ...

d1 = Data(2024,12,31)
d2 = Data(2023,10,10)

# quando mi serve una funzione chiave, uso cmp_to_key(comp_foo)
sorted([d1,d2], key=functools.cmp_to_key(cmp_date))
```

`@total_ordering`: decoratore per classi che fornisce i metodi di confronto per quella classe → basandosi su `__eq__` e uno tra gli altri metodi, **inferisce gli altri → mi basta implementare questi due**

`@LRU_cache`: decoratore che **evita la chiamata a funzione se già invocata con gli stessi argomenti** e ritorna il risultato già computato

`@LRU_cache(maxsize=x)` per settare il numero massimo di memorizzazioni

15.1.8 itertools

modulo che fornisce funzioni per generare sequenze di dati iterabili

```
from itertools import *

# iteratori infiniti
count(10)                                          # -> 10, 11, 12, ...
cycle([1,2,3])                                     # -> 1, 2, 3, 1, 2, 3, ...
repeat('a')                                        # -> a, a, a, ...

# iteratori che modificano la sequenza
accumulate([1,2,3,4])                              # -> 1 3 6 10
chain('abc', 'def')                                # -> a b c d e f
zip_longest('ABC', 'xyz')                          # -> Ax By Cz
starmap(pow, [(1,2), (3,4)])                       # -> applica pow(*x) -> 1 81
```

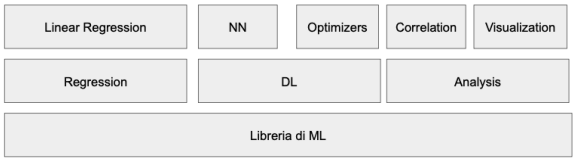
```
# iteratori combinatori
permutation([1,2,3])                    # -> tutte le permutazioni possibili
combination([1,2,3])                    # -> tutte le combinazioni possibili
```

16 struttura progetto

è bene organizzare un progetto software in più moduli e file → leggibilità e navigabilità, in alcuni casi anche ottimizzazione della compilazione (moduli non modificati non ricompilati), riutilizzo dei moduli

modulo: insieme di entità correlate con un fine preciso

package: raccolta di moduli correlati, inerenti alla stessa area del programma



i moduli corrispondono ai sorgenti, che contengono funzioni, costanti e classi inerenti al modulo

→ un package è una directory di più file sorgenti → **per far si che una dir sia vista come package è necessario inserirvi un file `__init__.py` (anche vuoto)**

16.1 moduli

```
# sintassi importazione |------------| uso nel codice
# |-----------------------------------------------|
import <modulo>                                                 <modulo>.<entit\`a>
import <modulo>        as m                                  m.<entit\`a>
from <modulo> import <entit\`a>                 <entit\`a>
from <modulo> import *                                <entit\`a>
import <modulo>.<entit\`a>                          <entit\`a>

# sintassi per importare moduli/entit\`a contenuti in package/subpackage
import <package>[.<subpackage>[.<modulo>[.<entit\`a>]]]
```

quando si importa il modulo, viene **eseguito il sorgente** → avvengono le definizioni delle entità ed eventualmente delle **inizializzazioni**

16.2 package

posso importare anche i package, e la loro inizializzazione può essere messa in `__init__.py`

se importo un package, avviene prima l'init del package che dei moduli

quando importo i moduli nella forma `from <pkg> import *` **non vengono importati tutti i moduli (motivi di performance)** → di per se non viene importato **nulla**

→ specifico i moduli da importare in `__init__.py` nella lista `__all__`

16.3 librerie

quando l'insieme di package del nostro programma è pensato per il riutilizzo in altri progetti

libreria standard e un enorme catalogo a disposizione tramite Pypi (`pip`)

altre librerie molto comuni: `string, datetime, math, ...`

`pip`: package manager per python, def cerca su Pypi ma volendo accetta anche altri repo esterni

`sys.path` ci mostra la lista delle dir in cui vengono cercati i package (e le librerie) → in ordine (solitamente):

dir corrente

path presenti nella variabile di ambiente `PYTHONPATH`

`/usr/lib/python.x.x`

'sys.path' si può modificare, ad es. per progetti che a runtime vogliono delle versioni specifiche di certi package

16.4 ambienti

proprio per diversificare i path a seconda del progetto, possiamo installare le versioni in dir differenti e modificare `PYTHONPATH` per ciascuno di essi

oppure creare un virtual environment Python use `venv`

una volta attivato, python installa le librerie soltanto in questo ambiente e l'interprete le cercherà soltanto lì → ogni progetto dovrebbe avere il suo per evitare conflitti!

`deactivate` per disattivarlo

`pip3 freeze > requirements.txt` per esportare la lista delle librerie usate nel venv

`pip3 install -r requirements.txt` per importarle

16.5 moduli e script

quando importo un modulo, ne eseguo il sorgente, **così come accade se eseguo direttamente il file come script...**

per distinguere il comportamento di un sorgente a seconda che sia importato come modulo o eseguito come script, sfrutto la variabile `__name__`, che prende **il nome del modulo quando importato**, e `__main__` quando è **lanciato come script**

buona norma inserire le istruzioni da eseguire come script in un `if __name__ == '__main__':`

es. `venv`: potremmo averne bisogno come modulo, ma solitamente lo usiamo per inizializzare un ambiente virtuale → `python3 -m venv /path/to/venv` **dove '-m' indica l'utilizzo del modulo come script** (equivale a `python3 /path/to/venv.py` ma senza dover conoscere la posizione del sorgente)

16.6 build

per distribuire codice è buona norma **buildarlo** in un pacchetto

si inserisce nella dir il file `pyproject.toml` che contiene (eventualmente) i metadati del progetto (autore, dipendenze, ...)

per Pypi si usa `build`:

```
pip3 install build
python3 -m build path/to/package
```

alla fine nella cartella `dist` trovo il package da poter distribuire

https://packaging.python.org/en/latest/guides/writing-pyproject-toml/#writing-pyproject-toml

17 gestione della memoria

i dati di un programma vengono salvati in memoria:

stack per variabili dichiarate nel codice

heap per allocazioni esplicite

problema di gestione: per quanto tempo mantengo occupato lo spazio in memoria?

approccio manuale:

decide il programmatore, tramite specifiche istruzioni nel codice (es. `malloc, dealloc` in C)

gestione esplicita e molto veloce, ma prona ad errori, memory leaks, segfaults e null pointers

approccio automatico:

decide il compilatore/interprete

seguono alcuni metodi e algoritmi

17.1 reference counting

algoritmo che libera automaticamente memoria quando un oggetto non è più referenziato (usato ad es. in Perl e PHP)

ogni oggetto ha un campo aggiuntivo che indica il suo numero di riferimenti (aggiornato dinamicamente) → una volta a 0, l'oggetto viene distrutto

molto veloce e reattivo, ma:

overhead per aggiornare il contatore

riferimenti circolari → 2 oggetti che si refenziano a vicenda non verranno mai distrutti!

soluzione: marcare come *deboli* i riferimenti ricorsivi (li uso solo se almeno uno degli oggetti è referenziato altrove) (di contro aumento ancora l'overhead)

17.2 garbage collector

entità che ciclicamente controlla lo stato degli oggetti in memoria ed eventualmente li elimina
alcuni algoritmi utilizzati

17.2.1 Tracing

partendo da degli oggetti radice segue tutti gli oggetti referenziati, proseguendo a cascata → tutti gli oggetti **non referenziati** sono **considerati garbage ed eliminati**
gli oggetti radice sono variabili globali, locali, e argomenti della funzione
motivi per cui un oggetto non è più raggiungibile:
sintattico: la sintassi implica che quell'oggetto non sia più raggiungibile → `a = 1: a = 2` → l'oggetto `1` non è più disponibile
facile da trovare, i gc si basano soprattutto su questo
semantico: oggetti irraggiungibili per via del flusso di codice (branch di codice inutilizzati)
difficile da trovare → euristiche
questo tipo di gc implica un overhead per trovare gli oggetti → bisogna stabilire quando chiamarlo
Java: ogni tot
Python: quando la memoria occupata raggiunge un limite

17.2.2 Mark and Sweep

ogni oggetto ha associato un flag (irraggiungibile o raggiungibile, `0`,`1`)
fase di **scansione**: si marchia gli oggetti come `0`/`1`
fase di **deallocazione**: si elimina gli oggetti `0`
molto limitante (devo interrompere il programma e riscansionare tutto)

17.2.3 Tri-color Marking

organizzo gli oggetti in 3 insiemi
White : candidati alla rimozione
Gray : raggiungibili ma i cui oggetti referenziati non sono stati analizzati
Black : raggiungibili che non referenziano nessun oggetto nel white set
unico flusso possibile: W → G → B
prima fase: inizializzazione degli insiemi (oggetti radice nel gray, gli altri nel white, black vuoto)
seconda fase (fino a svuotamento del gray):
scelgo un oggetto da G
identifico tutti gli oggetti che referenzia, in particolare quelli appartenenti a W
li inserisco in G, e inserisco l'oggetto in B
terza fase: libero la memoria degli oggetti in W → non sono raggiungibili **direttamente** (1a fase) né **indirettamente** (2a fase)
algoritmo con il vantaggio che step 1 e 2 possono essere eseguiti durante l'esecuzione senza interromperla

17.2.4 strategie di rilascio memoria

in movimento: copio tutti gli oggetti raggiungibili in una nuova area → meno frammentazione
non in movimento: rilascio le zone degli oggetti irraggiungibili

17.2.5 Generational

ipotesi avvallata da risultati empirici: gli oggetti creati più di recente hanno maggiore probabilità di diventare irraggiungibili nell'immediato futuro → suddivido gli oggetti in insiemi di "vecchiaia", facendo controlli frequenti solo sulle generazioni più giovani e tenendo traccia dei riferimenti tra generazioni
generazioni: eden → survivor 2 (sopravvissuti ad un certo numero di cicli) → survivor 1 (altro range intermedio) → old
ogni volta che un gruppo supera una soglia viene invocato il gc su di esso: gli oggetti raggiungibili sono copiati nel gruppo immediatamente più vecchio, poi la regione viene svuotata
più veloce (agisce su set ridotti), ma meno preciso

17.2.6 approcci ibridi

Minor cycle: frequente e performante, ma poco preciso (es. Generational)
Major cycle: fatto ogni tanto, ma su tutti gli oggetti (es. Mark and Sweep)

17.3 gestione memoria in python

in python ora è ibrido: reference counting e un generational garbage collector
modulo `gc` consente di impostare il garbage collector, attivarlo/disattivarlo, farlo eseguire su una generazione, eccetera
in realtà **dipende dall'interprete**: CPython (il più usato) usa principalmente il reference counting, e poi il gc generazionale a soglia

```
import sys
a = 'aaa'
sys.getrefcount(a) # -> 2 - conto anche il passaggio alla funzione
x = [a]
d = {'a':a}
sys.getrefcount(a) # -> 4 - usciti dalla prima funzione il contatore viene decrementato
```

per profilare il consumo di memoria, python ha diverse librerie tra cui `memory-profiler, tracemalloc`

```
import tracemalloc
def foo():
  f = [ x for x in range(0,100000) ]
tracemalloc.start()
foo()
current,peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(f"Istantanea {current}, picco {peak}")
```

la gestione della memoria a basso livello dipende dall'interprete e dal linguaggio in cui è scritto → in CPython (scritto in C) esiste un **allocatore di oggetti** che si occupa di gestire lo spazio ed allocare la memoria necessaria quando un oggetto viene creato
per evitare tante piccole allocazioni, alloca meno volte zone più grandi (di fatto gestisce un suo heap privato)
quando si crea un oggetto l'interprete lo alloca nello heap privato, e l'allocatore gli assegna una zona di memoria (eventualmente chiamando una `malloc()` se non basta), e viceversa per la rimozione
per oggetti $>$ 512 byte viene chiamata direttamente la `malloc()`
se l'oggetto ha un allocatore specifico viene chiamato questo
le zone di memoria gestite sono divise in Arene, a loro volta divise in Pool → i nuovi oggetti sono allocati nell'Arena con più pool pieni, perché più probabile che si svuotino (così posso liberarle)
un Pool ha la dimensione di una pagina di memoria (stabilita da SO, blocchi contigui), e contiene blocchi di dimensione prefissata e uguale
quando si alloca un oggetto si cerca il pool con blocchi di dimensione richiesta, e se non esistono si crea un nuovo pool
ogni blocco contiene un solo oggetto, e viene segnato come libero quando dealloco

18 debug

python offre un modulo specifico (a differenza di altri linguaggi): `pdb`
alcune funzioni a disposizione:

- `run(statement[, globals[, locals]])`: esegue `statement` (in forma di stringa) sotto il controllo del debugger → il prompt del debugger compare prima dell'esecuzione del codice, consentendo di impostare breakpoint, esaminare variabili e controllare il flusso del programma in maniera interattiva
- `globals` e `locals` specificano l'ambiente nel quale il codice viene eseguito (def usa il dizionario del modulo `__main__`)

- `runeval(expr[, globals[, locals]])`: valuta `expr` (stringa) attraverso il debugger, e restituisce il valore dell'espressione (per il resto simile a `run()`)
- `runcall(foo[, argument, ...])`: esegue `foo` (non stringa ma funzione o metodo) con gli argomenti forniti, e restituisce il valore di ritorno di `foo`

lanciato come script: `python3 -m pdb <programma>`
in modalità interattiva si possono lanciare vari comandi, la cui maggioranza ha una versione abbreviata da 1 o 2 lettere: si separano gli argomenti con spazi o tab

- `w(here)`: stampa la traccia dello stack, con il frame corrente indicato da una freccia
- `d(own)/u(p)`: sposta il frame corrente in basso/alto di un livello nella stack trace (verso un frame più recente/vecchio)
- `b(reak) [[filename:]lineno | function[, condition]]`: imposta un breakpoint su una riga o funzione, eventualmente con `condition` (lista bp se senza argomenti)
  - `filename:lineno`: specifico il file e la riga (file cercato in `sys.path`)
  - `function`: breakpoint all'inizio della funzione specificata
  - `condition`: espressione che deve essere vera perché il breakpoint sia rispettato
- `tbreak [[filename:]lineno | function[, condition]]`: imposta un breakpoint temporaneo che viene cancellato dopo il primo raggiungimento
- `cl(ear) [bpnumber [bpnumber ...]]`: cancella uno o più breakpoint, se nessun argomento è fornito **li cancella tutti ma chiedendo conferma**
- `disable [bpnumber [bpnumber ...]]`: disabilita i breakpoint forniti senza cancellarli, (riattivabili successivamente)
- `enable [bpnumber [bpnumber ...]]`: riabilita uno o più breakpoint disabilitati
- `ignore bpnumber [count]`: imposta quante volte ignorare un breakpoint (ignorato `count` volte, def 0) → a 0 diventa attivo
- `condition bpnumber [condition]`: aggiunge o rimuove `condition` (espressione che deve essere vera affinché il bp sia rispettato) per il breakpoint specificato
- `s(tep)`: esegue la riga corrente e si ferma alla prima occasione (in una funzione chiamata o alla prossima riga)
- `n(ext)`: continua l'esecuzione fino alla prossima riga nella funzione corrente, evitando di fermarsi all'interno di eventuali chiamate di funzione
- `r(eturn)`: continua l'esecuzione fino al termine della funzione corrente
- `c(ont(inue))`: riprende l'esecuzione fino al prossimo breakpoint
- `j(ump) lineno`: salta a `lineno` come prossima riga da eseguire
- `l(ist) [first[, last]]`: mostra le righe di sorgente attorno a quella corrente o in un intervallo specificato (def 11 righe)
- `a(rgs)`: stampa la lista degli argomenti della funzione corrente
- `p expr`: valuta e stampa il valore dell'espressione python `expr` nel contesto corrente.
- `pp expression`: come `p`, ma stampa il risultato formattato usando il modulo `pprint`
- `alias [name [command]]`: crea un alias per un comando di debug (senza argomenti elenca tutti gli alias attivi)
  - indico parametri sostitutivi con `\%1`, `\%2`,... mentre `\%*` viene sostituito da tutti i parametri

- possono essere annidati, e contenere qualsiasi cosa digitabile al prompt pdb; shadowing sui comandi interni; controllati ricorsivamente sulla prima parola della riga (tutte le altre vengono lasciate invariate)
- ```
  # 2 esempi utili
  #Visualizza le variabili d'istanza (utilizzo: "pi classInst")
  alias pi for k in \%1.__dict__.keys(): print "\%1.",k,"=",\%1.__dict__[k]
  #Visualizza le variabili d'istanza in self
  alias ps pi self
  ```

- `unalias name`
- `! statement`: esegue un'istruzione Python **monoriga** nel contesto dello stack frame corrente (`!` può essere omesso se la prima parola non è un alias o comando interno del debugger)

  ```
  # esempio: impostare globals - devo usare ;
  # per avere 2 istruzioni sulla stessa riga
  global gloval_var; global_var = something
  ```

- `q(uit)`

posso definire delle porzioni di codice da eseguire in debug con `if __debug__:` (`__debug__` sempre definita, tranne quando uso `-O` che indica all'interprete di ottimizzare il codice per la produzione)

modo più rapido: uso `assert` (?)

19 unit test

primo livello di test del software, in cui vengono testate le parti più piccole di un software → usato per convalidare che ogni unità del software funzioni come previsto

test case: insieme di condizioni usate per determinare se un sistema sottoposto a test funziona correttamente

test suite: raccolta di test case usati per testare un programma software e per dimostrare che ha una serie specifica di comportamenti eseguendo insieme i test aggregati

test runner: componente che imposta l'esecuzione dei test e fornisce il risultato all'utente

in python si usa il modulo **unittest**

```
import unittest
value = True # False
class Foo(unittest.TestCase):
  def test(self):
    self.assertTrue(value)

unittest.main()
```

un test case è una classe che eredita da `TestCase` → i test contenuti al suo interno sono funzioni con nome che inizia con `test`

per effettuare la valutazione si usano le assert messe a disposizione dal modulo → se non va a buon fine, il test viene considerato **NON superato**

tipi di risultati possibili:

`OK`: tutti i test superati

`FAIL`: test non passato, sollevata `AssertionError`

`ERROR`: sollevata un'eccezione diversa da `AssertionError`

metodi per preparare i test ed eseguire operazioni dopo l'esecuzione (es. connessione a db, chiusura file):
```

```
setUp()                  # eseguito prima di ogni test
tearDown()                  # eseguito dopo ogni test
setUpClass()          # eseguito prima di tutti i test della classe
tearDownClass() # eseguito dopo tutti i test della classe
```

20 file I/O

preambolo su filesystem e cose varie

20.1 apertura file

open('pathname',mode)‘

`mode`: `r`,`w`,`a`, `b` (quest'ultimo da aggiungere agli altri per aprire il file in modalità binaria)
ritorna il descrittore del file (da usare per individuare il file aperto)

```
# LETTURA
fd = open('path','r')
fd.read() # legge intero file in una stringa
fd.readline()
for line in fd: # legge una riga alla volta
  # ...
fd.close()

# SCRITTURA
fd = open('path','w')
fd.write(s) # scrivo una stringa
print('hello', file=fd) #solo in python3
fd.close()

# SCRITTURA IN MODALIT\`A BINARIA
fd = open('prova.bin', 'wb')
fd.write(bytes([1, 2, 3]))
fd.close()
fd2 = open('prova.bin', 'rb')
a = fd2.read()
a = list(a)

# SINTASSI ALTERNATIVA
with open('path','r') as fd:
  # blocco eseguito solo se non avvengono errori nell'apertura
  # il file viene chiuso automaticamente in ogni caso
```

20.2 Pickle

modulo che permette di serializzare su file oggetti arbitrari

```
pickle.dump(obj, file)         # -> salva l'oggetto su file
pickle.dumps(obj)                 # -> restituisce la serializzazione come stringa
pickle.load(file)                 # -> legge il file e restituisce l'oggetto
pickle.loads(data)                 # -> legge data (stringa) e restituisce l'oggetto
```

posso salvare builtin, tipi numerici, tipi stringa, iterabili, funzioni (no lambda), classi, istanze con metodo `__getstate__` salvabile

funzioni e classi vengono salvate usando **solo il nome** (per sicurezza) → quando vengono deserializzate è necessario importarle preventivamente

può essere pericoloso → posso deserializzare oggetti con codice malevolo

21 performance: C vs Python

profilazione dei tempi di esecuzione con `python3 -m cProfile foo.py`
possibile profilare parti specifiche di codice importando `cProfile`:

```
import cProfile, pstats, io
from pstats import SortKey

pr = cProfile.Profile()
pr.enable()
# ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.pint_stats()
print(s.getvalue())
```

spesso si scrive un programma in python e poi si importano funzioni C per le parti più onerose → codice semplice per la maggior parte del programma, e velocità nelle parti onerose
scrivo la/le funzioni C e le compilo: `cc -fPIC -shared -o my_functions.so my_functions.c`
carico la libreria C nel programma python, importo il file compilato e uso le funzioni

```
from ctypes import *

so_file = "/path/to/my_functions.so"
my_functions = CDLL(so_file)
print(my_functions.foo())
```

22 parsing argomenti

```
python3 script.py arg1 arg2 ...
```

nel programma li trovo (importando `sys`) in `sys.argv` (lista) → implica parsare manualmente gli argomenti (tenendo conto di posizione, tipo, assegnazioni eccetera)
il modulo `argparse` gestisce tutto in automatico: definisco cosa mi aspetto come argomenti, poi il modulo genera un help e parsa gli argomenti in modo corretto

```
import argparse
parser = argparse.ArgumentParser(description="Process some integers.")

# fase di definizione
parser.add_argument(dest='integers',
                    metavar='N',
                    type=int,
                    nargs='*',
                    help='an integer for the accumulator')

# fase di parsing - ritorna un ogegetto Namespace con le variabili popolate
arguments = parser.parse_args()
```

`add_argument('argname')`: specifico il nome dell'argomento, considerato posizionale ed obbligatorio (errore se non lo inserisco)
`add_argument('-a','--arg')`: specifica il nome dell'argomento in versione corta e lunga → parametri considerati opzionali
altri parametri opzionali della funzione:

- `dest`: specifico la variabile di destinazione (def nome argomento)
- `type`: tipo di dato (def stringa) (`argparse.FileType('r')` per file)

- `default`: valori def per i parametri opzionali
- `nargs`: numero di valori da considerare per quell'argomento
- `action`: azione da compiere sui valori
- `store`: default
- `store_true/false`: salva `True`/`False` a seconda che il parametro sia stato specificato (o viceversa) → per controllare argomenti flag
- `save_const`: salva il valore definito in `const`, `False` se non definito
- `append`: crea una lista se il parametro è specificato più volte
- `count`: conta il numero di volte che il parametro è specificato (`-vvv -> v=3`)
- `help`: stringa da mostrare per l'argomento
- `metavar`: valore di esempio da mostrare nell'help