Classi

Definizione di Tipi di Dato

Python è un linguaggio multi-paradigma

...Ma è considerato soprattutto un linguaggio orientato agli oggetti

- La caratteristica dei linguaggi orientati agli oggetti (object-oriented)
- ...È di permettere la definizione di nuovi tipi di dato

Cosa possiamo fare con i tipi di dato primitivi?

E.g. interi, floating point...

- Costruire nuove istanze (e.g. il numero 3, il numero 1.2)
- Eseguire operazioni su di essi

Python ci permette di fare lo stesso con tipi di dato definiti dall'utente

Per esempio, possiamo definire un tipo "numero complesso"

Classi

Il costrutto che permette di definire nuovi tipi si chiama classe

- Le classi hanno una sintassi piuttosto complessa
- ...Quindi la introdurremo per gradi

Una classe rappresenta l'archetipo (il "progetto") per un nuovo tipo di dato

La classe deve specificare:

- Il nome del nuovo tipo di dato
- ...Come esso è rappresentato
- ...Come fare a costruirne un'istanza
- ...Quali operazioni sono disponibili su di esso

Classi

Per introdurre una nuova classe, si usa l'istruzione `class`

```
class <nome classe>:
    ...
```

- La parola chiave `class` indica l'inizio dell'istruzione
- `<nome classe>` è un identificatore (con le solite regole)

Vediamo un esempio (numero complesso)

In [1]:
```
class Complex:
    pass
```

- Per convenzione, si fanno iniziare i nomi di classe con una maiuscola
- `pass` è un'istruzione "nulla" (la specifica della classe è assente)

Costruttore di Default

Per costruire una istanza di una classe (i.e. un oggetto)

...Ne si "chiama" il nome come se fosse una funzione:

In [2]:
```
a = Complex()
print(a)
```

```
<__main__.Complex object at 0x108964bd0>
```

Tale funzione si chiama costruttore

- Quando viene eseguita, Python alloca memoria per il nuovo oggetto
- ...E restituisce l'oggetto creato (i.e. un dato)

Una classe con specifica vuota ha un costruttore di default

...Che non va niente se non allocare memoria


Campi

È possibile definire variabili nell'istanza creata

Per farlo, occorre tenere presente che:

- Una classe definisce un namespace
- ...Ed ogni istanza della classe ha una sua area di memoria

Al momento della definizione vanno quindi specificate classe ed istanza

In Python si forniscono entrambe usando la notazione puntata

In [3]:
```
a = Complex()
a.real = 1 # variabile real, nell'istanza appena creata
a.img = -1 # variabile img, nell'istanza appena creata
```

- Prima si indica l'istanza (e di conseguenza la classe) in cui creare la variabile
- ...E dopo il punto si specifica il nome della variabile da creare


Campi

Le variabili specificate in questo modo si dicono campi (o attributi)

- A parte l'uso della notazione puntata
- ...Si comportano come normali variabili

Vediamo qualche esempio:

In [4]:
```
a = Complex()
a.real = 1
a.img = -1
print(f'Il numero è: {a.real} + {a.img}i')
```

```
Il numero è: 1 + -1i
```

- In particolare possiamo accederne al contenuto
- ...E modificarlo, se necessario


Definizione di Costruttori

Tipicamente, è naturale che tutte le istanze di una certa classe

...Condividano determinati campi

- E.g. tutti i numeri complessi hanno una parte reale ed un immaginaria

Possiamo assegnarli a mano, e.g.:

In [5]:
```
a, b = Complex(), Complex()
a.real, a.img = 1, -1
b.real, b.img = 2, 0.5
```

...Ma è molto scomodo e soggetto ad errori

Sarebbe molto meglio definire una funzione per costruire nuovi oggetti


Definizione del Costruttore

Python lo consente permettendo di definire il costruttore di una classe

```
class <nome classe>:
    def __init__(self, ...):
        ...
```

- Come nome si usa sempre `__init__`
  - Nella chiamata useremo il nome della classe (come prima)
- Il primo parametro contiene sempre l'oggetto che stiamo costruendo
  - Per convenzione, lo si chiama `self`
- Un costrutture può avere altri parametri, a seconda della necessità


Definizione del Costruttore

Vediamo un esempio per i numeri complessi

In [6]:
```
class Complex:
    def __init__(self, real, img):
        self.real = real
        self.img = img
```

- Il nostro costruttore richiede due parametri (`real` ed `img`)
- Assegna alla variabile `self.real` il valore contenuto in `real`
- Assegna alla variabile `self.img` il valore contenuto in `img`

Si noti che `self.real` e `real` sono variabili distinte

- `self.real` è una variabile nell'istanza/dato che stiamo costruendo
- `real` è un (normalissimo) parametro formale


Definizione del Costruttore

Al momento della chiamata è necessario passare i parametri

In [7]:
```
a = Complex(1, -1)
b = Complex(2, 0.5)
print(f'a contiene: {a.real} + {a.img}i')
print(f'b contiene: {b.real} + {b.img}i')
```

```
a contiene: 1 + -1i
b contiene: 2 + 0.5i
```

- Visto che abbiamo definito un costruttore
- ...Questo viene eseguito al momento della costruzione di un nuovo dato
- ...Ed inizializza i campi `real` e `img`, come abbiamo indicato


Funzioni per Operare su Oggetti

Di solito, ad un tipo di dato sono associate determinate operazioni

- Per esempio, nel caso dei numeri complessi
- ...Potremmo avere la somma, prodotto, coniugato...

Possiamo pensare di implementarle usando normali funzioni:

In [8]:
```
def add_complex(x, y):
    return Complex(x.real + y.real, x.img + y.img)

a, b = Complex(1, -1), Complex(2, 0.5)
c = add_complex(a, b)
print(f'c = {c.real} + {c.img}i')
```

```
c = 3 + -0.5i
```

- L'apprioccio è corretto ed utilizzabile
- ...Ma sarebbe più comodo associare la funzioni direttamente alla classe!

Metodi

Una funzione associata ad una classe si chiama metodo

La sintassi è la solita, ma compare dentro la definizione di classe:

```
class <nome classe>:
    def <nome metodo>(self, ...):
        ...
```

Un metodo viene invocato su un oggetto

- Tale oggetto viene passato al metodo come primo argomento
- Per questa ragione, il primo argomento si chiama per convenzione `self`


Metodi

Vediamo un esempio:

In [9]:
```python
class Complex:
    def __init__(self, real, img):
        self.real, self.img = real, img

    def add(self, x):
        return Complex(self.real + x.real, self.img + x.img)

a, b = Complex(1, -1), Complex(2, 0.5)
c = a.add(b) # <-- Chiamata del metodo
print(f'c = {c.real} + {c.img}i')
```

```
c = 3 + -0.5i
```

- Per invocare un metodo si usa la notazione puntata
- In questo modo si specifica l'oggetto da passare come `self`
- ...E di conseguenza la classe da usare come namespace


Ridefinizione di Operazioni

Utilizzando metodi con nomi specifici...

...È integrare la classe con le operazioni base di Python

- Si può supportare l'operatore `+` definendo un metodo `__add__`
- Si può supportare l'operatore `*` definendo un metodo `__mul__`
- Si può definire come un oggetto si convertito in stringa definendo `__repr__`

Potete trovare informazioni dettagliate sulla documentazione ufficiale

Si tratta di una funzionalità molto comoda

- Può rendere il codice più compatto e leggibile
- ...Ed è sfruttata da molti dei pacchetti Python più utilizzati

Ridefinizione di Operazioni

Vediamo un esempio per la nostra classe

In [10]:
```python
class Complex:
    def __init__(self, real, img):
        self.real, self.img = real, img

    def __add__(self, x):
        return Complex(self.real + x.real, self.img + x.img)

    def __repr__(self):
        return f'{self.real} + {self.img}i'

a, b = Complex(1, -1), Complex(2, 0.5)
c = a + b # <-- viene utilizzato __add__
print(f'c = {c}') # <-- viene utilizzato __repr__
```

```
c = 3 + -0.5i
```

- `a + b` corrisponde alla chiamata `a.__add__(b)`
- Quando un `Complex` è convertito in stringa viene chiamato `__repr__`


Metodi delle Liste

Molti costrutti che abbiamo già utilizzato sono implementati come classi

...E di conseguenza hanno diversi metodi!

- Vediamo qualche metodo delle liste:

In [11]:
```
l = ['a', 'b', 'c', 'd']

l.append('e') # Aggiunge un elemento in coda
print(l)

print(l.pop(2)) # Rimuove un elemento dalla lista e lo restituisce
print(l)

print(l.count('a')) # Conta il numero di occorrenze di un valore
```

```
['a', 'b', 'c', 'd', 'e']
c
['a', 'b', 'd', 'e']
1
```


Metodi delle Liste

Molti costrutti che abbiamo già utilizzato sono implementati come classi

...E di conseguenza hanno diversi metodi!

- Vediamo qualche metodo delle liste:

In [12]:
```
l = ['a', 'b', 'c', 'd']

print(l.index('b')) # Restituisce l'indice di un elemento nella lista

l2 = l.copy() # copia la lista
print(l2)
```

```
1
['a', 'b', 'c', 'd']
```

- Inoltre le liste utilizzano la ridefinizione degli operatori
- Per maggiori dettagli, potete consultare questa pagina


Metodi delle Tuple

Per le tuple sono definiti solo un paio di metodi:

In [13]:
```
t = ('a', 'b', 'c', 'a', 'b')

print(t.count('a')) # Conta il numero di occorrenze di un elemento

print(t.index('b')) # Indice di un elemento in una tupla
```

```
2
1
```

- Per maggiori dettagli, potete consultare questa pagina

Metodi degli Insiemi

Per gli insiemi abbiamo:

In [14]:
```
s = {'a', 'b', 'c'}

s.add('e') # Aggiunge un elemento
print(s)

s.pop() # Rimuove un elemento (non possiamo scegliere quale)
print(s)


print(s.copy()) # Restituisce una copia dell'insieme
```

```
{'e', 'b', 'c', 'a'}
{'b', 'c', 'a'}
{'b', 'c', 'a'}
```

- Per maggiori dettagli, potete consultare questa pagina


Metodi dei Dizionari

Per i dizionari abbiamo:

In [15]:
```
d = {'nome': 'Mario', 'cognome': 'Rossi'}

for v in d.values(): # Generatore per iterare sui _valori_
    print(v)

for k, v in d.items(): # Generatore per iterare su coppie (chiave, valore)
    print(f'{k}: {v}')


print(d.copy()) # Restituisce una copia
```

```
Mario
Rossi
nome: Mario
cognome: Rossi
{'nome': 'Mario', 'cognome': 'Rossi'}
```

- Per maggiori dettagli, potete consultare questa pagina

Metodi delle Stringhe

Per le stringhe sono disponibili molti metodi

Qui ne vediamo solo alcuni:

In [16]:
```
s = 'Ciao, mondo!'

print(s.find('mondo')) # Indice di una stringa all'interno di un'altra

print(s.count('o')) # Numero di occorrenze di un carattere


print(s.split(',')) # Divide in più stringhe, sulla base di un separatore
```

```
6
3
['Ciao', ' mondo!']
```

- Potete documentarvi riguardo agli altri su questa pagina


Operatore `==` ed Operatore `is`

Supponiamo di volere confrontare dei numeri complessi

- Allo scopo, possiamo ridefinire l'operatore di uguaglianza `==`
- Il metodo da fornire si chiama in questo caso `__eq__`

In [17]:
```
class Complex:
    def __init__(self, real, img):
        self.real, self.img = real, img

    def __add__(self, x):
        return Complex(self.real + x.real, self.img + x.img)

    def __eq__(self, x): # Operatore di uguaglianza
        return self.real == x.real and self.img == x.img

    def __repr__(self):
        return f'{self.real} + {self.img}i'
```


Operatore `==` ed Operatore `is`

Questo approccio ci permette di effettuare confronti

In [18]:
```
a = Complex(1, -1)
b = Complex(2, 0.5)
c = Complex(1, -1)
print(f'a == a: {a == a}')
print(f'a == b: {a == b}')
print(f'a == c: {a == c}')
```

```
a == a: True
a == b: False
a == c: True
```

- Come da noi specificato, l'operatore `==` restituisce `True`
- ...Se i due oggetti confrontati hanno gli stessi valori

Quando si lavora con le classi, però, è possibile un altro tipo di confronto


Operatore `==` ed Operatore `is`

In particolare, ci può interessare se due espressioni...

...Denotano lo stesso oggetto come la stessa area di memoria

- Per questo scopo Python fornisce l'operatore `is`:

In [19]:
```
a = Complex(1, -1)
b = Complex(2, 0.5)
c = Complex(1, -1)
print(f'a is a: {a is a}')
print(f'a is b: {a is b}')
print(f'a is c: {a is c}')
```

```
a is a: True
a is b: False
a is c: False
```

- `a is a` denota `True`, perché si tratta dello stesso oggetto
- `a is c` denota `False`, anche se parte reale ed immaginaria sono uguali

Operatore `==` ed Operatore `is`

Per negare `is` si può usare `not` come al solito:

In [20]:
```
a = Complex(1, -1)
print(f'not (a is a): {not (a is a)}')
```

```
not (a is a): False
```

...Ma anche la sintassi `is not` (più leggibile):

In [21]:
```
print(f'a is not a: {a is not a}')
```

```
a is not a: False
```


Operatore `==` ed Operatore `is`

L'operatore `is` is usa soprattutto per i confronti con `None`

In particolare, si usano:

- `<espr.> is None` per sapere se l'espressione denota `None`
- `<espr.> is not None` per sapere se denota qualcosa di diverso da `None`

In [22]:
```
a = None
b = 3
print(f'a is None: {a is None}')
print(f'b is None: {b is None}')
print(f'a is not None: {a is not None}')
print(f'b is not None: {b is not None}')
```

```
a is None: True
b is None: False
a is not None: False
b is not None: True
```


Classi e Funzioni

In Python, le funzioni sono oggetti cioè istanze di una classe

- In particolare, ogni oggetto può essere considerato una funzione
- ...Purché implementi il metodo `__call__(self, ...)`

In effetti, quando usiamo l'operatore di esecuzione, i.e. `()`

...Inneschiamo una chiamata al metodo `__call__`

```
In [23]: print('ciao')
         print.__call__('ciao')

         ciao
         ciao
```

- `print` è a tutti gli effetti una variabile
- ...Contiene un oggetto che implementa il metodo `__call__`
- ...Possiamo chiamarlo esplicitamente, oppure in modo più facile con `()`


Classi e Funzioni

Possiamo usare questa caratteristica per creare "funzioni configurabili"

In [24]:
```python
class Saluta:

    def __init__(self, name):
        self.name = name

    def __call__(self):
        print(f"Ciao, {self.name}!")
```

- La classe `Saluta` contiene il campo `name`
- ...Che viene inizializzato in fase di costruzione
- La classe definisce anche il metodo `__call__`
- ...Quindi può essere chiamata (è una funzione!)


Classi e Funzioni

Possiamo usare questa caratteristica per creare "funzioni configurabili"

```
In [25]: f = Saluta('Mario')
         g = Saluta('Luigi')
```

- `f` è una istanza di `Saluta`, in cui `name` vale "Mario"
- `g` è una istanza di `Saluta`, in cui `name` vale "Luigi"

```
In [26]: f()
         g()
```

```
Ciao, Mario!
Ciao, Luigi!
```

- Sia `f` che `g` possono essere chiamate
- ...Ma il loro comportamento è diverso
- ...Perché `name` ha un valore diverso nelle due istanze!

Altri Argomenti

Le classi hanno un ruolo fondamentale anche in altri meccanismi

...Che però non copriremo nel corso:

- Si possono definire classi che ereditano da altre classi
  - Si dice che una classe derivata eredita lo schema di una classe base
  - La classe derivata acquisisce tutti i metodi ed i campi di quella base
  - Potete trovare maggiori dettagli sulla documentazione ufficiale
- La classi hanno un ruolo importante nella gestione degli errori
  - Si utilizza il meccanismo delle eccezioni
  - ...In cui speciali classi rappresentano tipologie di errore
  - Potete trovare maggiori dettagli sulla documentazione ufficiale