Indice

- 1. File
  - 1.1. Nomi file e percorsi
  - 1.2. F-string
  - 1.3. Serializzazione e deserializzazione di file
  - 1.4. Attraversamento di directory
- 2. Classi
  - 2.1. Definizioni di classe
  - 2.2. Copia di oggetti
    - 2.2.1. Copia superficiale vs copia profonda
  - 2.3. Funzioni pure
- 3. Metodi
  - 3.1. Definizioni di metodi
  - 3.2. Metodi di istanza
  - 3.3. Metodi statici
  - 3.4. Metodi speciali
    - 3.4.1. Inizializzazione di oggetti
    - 3.4.2. Stampa di oggetti
    - 3.4.3. Overloading di operatori
- 4. Ereditarietà
  - 4.1. Variabili di classe
  - 4.2. Insieme totalmente ordinato di oggetti
  - 4.3. Delegazione
  - 4.4. Genitori e figli
    - 4.4.1. Polimorfismo
    - 4.4.2. Principio di sostituzione di Liskov
- 5. Extra
  - 5.1. Set
  - 5.2. Counter
  - 5.3. Defaultdict
  - 5.4. Espressioni condizionali
  - 5.5. List comprehension

- Glossario
- Bibliografia
- Licenze

1. File

1.1. Nomi file e percorsi

I file sono organizzati in directory.

Ogni programma in esecuzione ha una directory di lavoro corrente, che è la directory predefinita per la maggior parte delle operazioni.

```
>>> import os
>>> os.getcwd()
'/home/fglmtt/github.com/admin'

Nota che

    os fornisce funzioni per lavorare con file e directory

    getcwd restituisce la directory di lavoro corrente

    /home/fglmtt/github.com/admin è un percorso

Percorsi assoluti vs relativi

    /home/fglmtt/github.com/admin è un percorso assoluto

    data/words.txt è un percorso relativo

Funzione    Descrizione
listdir Elenca il contenuto di una directory
path.exists Verifica se un file o directory esiste
path.isdir  Verifica se un percorso si riferisce a una directory
path.isfile Verifica se un percorso si riferisce a un file
path.join   Unisce directory e nomi file in un percorso

path è un sottomodulo di os.
python

>>> import os
>>> os.path.exists('code')
True
>>> os.path.isfile('code')
False
>>> os.path.isdir('code')
True
>>> os.listdir('code')
['text_inspector.py', 'spelling_bee.py', 'uses_any.py', 'data']
>>> os.path.join(os.getcwd(), 'code', 'data', 'words.txt')
'/home/fglmtt/github.com/admin/code/data/words.txt'

1.2. F-string

Stringhe che:

    Hanno la lettera f prima delle virgolette di apertura

    Contengono una o più espressioni tra parentesi graffe

    Le espressioni sono automaticamente convertite in str
```

```
python

>>> d = {'one': 1}
>>> l = [1, 2, 3]
>>> f'dict: {d}, list: {l}, sum list: {sum(l)}'
"dict: {'one': 1}, list: [1, 2, 3], sum list: 6"

>>> d = {'one': 1}
>>> l = [1, 2, 3]
>>> writer = open('deleteme.txt', 'w')
>>> writer.write(f'dict: {d}, list: {l}\n')
34
>>> writer.write(f'sum list: {sum(l)}\n')
12
>>> writer.close()
>>> print(open('deleteme.txt').read())
dict: {'one': 1}, list: [1, 2, 3]
sum list: 6

>>> d = {'one': 1}
>>> l = [1, 2, 3]
>>> writer = open('deleteme.txt', 'w')
>>> writer.write("dict: " + d + ",list: " + l + "\n")
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "dict") to str

>>> writer.write("dict: " + str(d) + ",list: " + str(l) + "\n")
34

1.3. Serializzazione e deserializzazione di file

I programmi leggono e scrivono file per memorizzare dati di configurazione.

I dati di configurazione:

    Specificano cosa un programma dovrebbe fare e come

    Sono tipicamente scritti in YAML o JSON

yaml e json forniscono funzioni per (de)serializzare i dati di conseguenza.
python

>>> writer = open('config.json', 'w')
>>> config = {'config_1': 1, 'config_2': 2}
>>> writer.write(json.dumps(config))
30
>>> writer.close()
>>> json.loads(open('config.json').read())
{'config_1': 1, 'config_2': 2}

>>> writer = open('config.json', 'w')
>>> config = {'config_1': 1, 'config_2': 2}
```

```
>>> writer.write(str(config))
30
>>> writer.close()
>>> json.loads(open('config.json').read())
Traceback (most recent call last):
...
json.decoder.JSONDecodeError: Expecting property name enclosed in double
quotes: line 1 column 2 (char 1)

>>> print(open('config.json').read())
{'config_1': 1, 'config_2': 2}

Dato che:

    str(config) non serializza config correttamente

    json.loads si aspetta " ma riceve '

json.loads solleva json.decoder.JSONDecodeError.
python

>>> writer = open('config.json', 'w')
>>> config = {'config_1': 1, 'config_2': 2}
>>> writer.write(json.dumps(config))
30
>>> writer.close()
>>> print(open('config.json').read())
{"config_1": 1, "config_2": 2}

1.4. Attraversamento di directory

walk.py (vedi qui):
python

def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)
        if os.path.isfile(path):
            print(path)
        elif os.path.isdir(path):
            walk(path)

L'output dovrebbe essere equivalente a:
bash

$ ls -R

2. Classi
2.1. Definizioni di classe

Una classe è un tipo definito dal programmatore.
python

class Time:
```

```
    """Rappresenta un orario del giorno"""

    L'header indica che la nuova classe si chiama Time

    Il body è solo una docstring

Una definizione di classe crea un oggetto classe, che è come una fabbrica
per creare oggetti.
python

>>> class Time:
...     """Rappresenta un orario del giorno"""
...
>>> lunch = Time()
>>> type(lunch)
<class '__main__.Time'>
>>> print(lunch)
<__main__.Time object at 0x104f78650>

lunch è un'istanza della classe Time.
2.2. Copia di oggetti

Gli oggetti sono passati alle funzioni come alias.

Il modulo copy fornisce la funzione copy che può duplicare qualsiasi
oggetto.
python

>>> import copy
>>> class Time:
...     """Rappresenta un orario del giorno"""
...
>>> lunch = Time()
>>> dinner = copy.copy(lunch)
>>> lunch is dinner
False

>>> import copy
>>> class Time:
...     """Rappresenta un orario del giorno"""
...
>>> lunch = Time()
>>> dinner = copy.copy(lunch)
>>> lunch is dinner
False
>>> lunch == dinner
False

Per le classi definite dal programmatore, il comportamento predefinito di
== è lo stesso di is, cioè == verifica l'identità, non l'equivalenza.
2.2.1. Copia superficiale vs copia profonda

Il modulo copy fornisce due funzioni per copiare oggetti.
Funzione    Descrizione
```

```
copy    Copia l'oggetto ma non gli oggetti che contiene (detta copia
superficiale)
deepcopy    Copia l'oggetto, gli oggetti a cui si riferisce, gli oggetti a
cui essi si riferiscono, e così via (detta copia profonda)
2.3. Funzioni pure
Funzioni

    I cui valori di ritorno sono identici per argomenti identici

    Non hanno effetti collaterali (es. modifica di argomenti riferimento
mutabili)

Tutto ciò che può essere fatto con funzioni impure può anche essere fatto
con funzioni pure.

Come regola generale:

    Scrivi funzioni pure quando è ragionevole

    Ricorri a funzioni impure solo se c'è un vantaggio

Il seguente codice non è idiomatico e serve solo come esempio.

custom_time_v1.py (vedi qui):
python

import copy

class Time:
    """Rappresenta un orario del giorno"""

def print_time(time):
    s = (
        f"{time.hour:02d}:"
        f"{time.minute:02d}:"
        f"{time.second:02d}"
    )
    print(s)

def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time

def increment_time(time, hours, minutes, seconds):
    time.hour += hours
    time.minute += minutes
    time.second += seconds

    carry, time.second = divmod(time.second, 60)
    carry, time.minute = divmod(
        time.minute + carry, 60
```

```
    )
    carry, time.hour = divmod(time.hour + carry, 60)

def add_time(time, hours, minutes, seconds):
    total = copy.copy(time)
    increment_time(
        total,
        hours,
        minutes,
        seconds
    )
    return total

python

>>> import custom_time_v1 as custom_time
>>> start = custom_time.make_time(9, 20, 0)
>>> custom_time.print_time(start)
09:20:00
>>> custom_time.increment_time(start, 1, 0, 0)
>>> custom_time.print_time(start)
10:20:00

increment_time è una funzione impura.
python

>>> import custom_time_v1 as custom_time
>>> start = custom_time.make_time(9, 20, 0)
>>> end = custom_time.add_time(start, 1, 0, 0)
>>> custom_time.print_time(start)
09:20:00
>>> custom_time.print_time(end)
10:20:00
>>> start is end
False

add_time è una funzione pura.
Funzione built-in   Descrizione
type(obj)   Restituisce il tipo di obj
isinstance(obj, class)  Verifica se obj è istanza di class
hasattr(obj, str)   Verifica se str è un attributo di obj
vars(obj)   Restituisce gli attributi di obj
python

>>> import custom_time_v1 as custom_time
>>> type(custom_time.Time)
<class 'type'>
>>> start = custom_time.make_time(9, 20, 0)
>>> type(start)
<class 'custom_time_v1.Time'>
>>> isinstance(start, custom_time.Time)
True
>>> hasattr(start, 'duration')
False
```

```
>>> vars(start)
{'hour': 9, 'minute': 20, 'second': 0}

3. Metodi
3.1. Definizioni di metodi

Un metodo è una funzione che è definita all'interno di una definizione di
classe.
python

class Time:
    """Rappresenta un orario del giorno"""
    def print_time(self):
        s = (
            f"{self.hour:02d}:"
            f"{self.minute:02d}:"
            f"{self.second:02d}"
        )
        print(s)

def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time

    print_time è un metodo di istanza

    make_time è una funzione

3.2. Metodi di istanza

Metodi che devono essere invocati con un oggetto come ricevitore.

Due modi per chiamare print_time:

Supponiamo che start sia un'istanza di Time.

Sintassi di funzione (meno comune):
python

Time.print_time(start)

    start è il ricevitore e viene passato come parametro

Sintassi di metodo (idiomatica):
python

start.print_time()

    start è l'oggetto su cui viene invocato il metodo (ricevitore)

    Il ricevitore è assegnato al primo parametro
```

```
        self si riferisce a start all'interno del metodo print_time

3.3. Metodi statici

Metodi che possono essere invocati senza un oggetto come ricevitore.
python

class Time:
    """Rappresenta un orario del giorno"""
    def print_time(self):
        s = (
            f"{self.hour:02d}:"
            f"{self.minute:02d}:"
            f"{self.second:02d}"
        )
        print(s)

    def int_to_time(seconds):
        minute, second = divmod(seconds, 60)
        hour, minute = divmod(minute, 60)
        return make_time(hour, minute, second)

def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time

    print_time è un metodo di istanza

    int_to_time è un metodo statico

    make_time è una funzione

I metodi statici non hanno self come parametro.

I metodi statici sono invocati sull'oggetto classe.
python

start = Time.int_to_time(34800)

3.4. Metodi speciali

Cambiano il modo in cui gli operatori e alcune funzioni lavorano con un
oggetto.
Metodo speciale Descrizione
__init__    Inizializza gli attributi di un oggetto. Gli argomenti sono
quelli passati all'oggetto classe
__str__ Restituisce una rappresentazione stampabile di un oggetto. Il
valore di ritorno deve essere una str. Chiamato da str() e print()
__add__ Implementa l'operatore +
__eq__  Implementa l'operatore ==
```

```
3.4.1. Inizializzazione di oggetti

custom_time_v2.py (vedi qui, questo è codice idiomatico):
python

class Time:
    """Rappresenta un orario del giorno"""
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

python

>>> import custom_time_v2 as custom_time
>>> start = custom_time.Time(9, 20, 0)
>>> start
<custom_time_v2.Time object at 0x100bc7110>
>>> start = custom_time.Time()
>>> start
<custom_time_v2.Time object at 0x100bf43e0>

3.4.2. Stampa di oggetti
python

class Time:
    """Rappresenta un orario del giorno"""
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return (
            f"{self.hour:02d}:"
            f"{self.minute:02d}:"
            f"{self.second:02d}"
        )

python

>>> import custom_time_v2 as custom_time
>>> start = custom_time.Time(9, 20, 0)
>>> custom_time.Time.__str__(start)
'09:20:00'
>>> start.__str__()
'09:20:00'

>>> import custom_time_v2 as custom_time
>>> start = custom_time.Time(9, 20, 0)
>>> str(start)
'09:20:00'
>>> print(start)
09:20:00
```

3.4.3. Overloading di operatori

L'overloading di operatori è il processo di utilizzare metodi speciali per cambiare il modo in cui gli operatori lavorano con tipi definiti dal programmatore.
python

```
class Time:
    """Rappresenta un orario del giorno"""

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return (
            f"{self.hour:02d}:"
            f"{self.minute:02d}:"
            f"{self.second:02d}"
        )

    def int_to_time(seconds):
        minute, second = divmod(seconds, 60)
        hour, minute = divmod(minute, 60)
        return Time(hour, minute, second)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def __add__(self, other):
        seconds = (
            self.time_to_int() +
            other.time_to_int()
        )
        return Time.int_to_time(seconds)

    def __eq__(self, other):
        return (
            self.hour == other.hour
            and self.minute == other.minute
            and self.second == other.second
        )
```

python

```
>>> import custom_time_v2 as custom_time
>>> start = custom_time.Time(hour=1)
>>> end = custom_time.Time(hour=1, minute=30)
>>> str(end)
'01:30:00'
```

```
>>> import custom_time_v2 as custom_time
>>> start = custom_time.Time(hour=1)
>>> end = custom_time.Time(hour=1, minute=30)
>>> t = custom_time.Time(1, 0, 0)
>>> start == t
True
>>> start is t
False

4. Ereditarietà
4.1. Variabili di classe

Variabili definite all'interno di una definizione di classe, ma non
all'interno di alcun metodo.

poker.py (vedi qui):
python

class Card:
    """Rappresenta una carta da gioco standard"""
    suits = [
        'Clubs', 'Diamonds', 'Hearts', 'Spades'
    ]
    ranks = [
        None, 'Ace', '2', '3', '4', '5',
        '6', '7','8', '9', '10',
        'Jack', 'Queen', 'King', 'Ace'
    ]

Le variabili di classe sono associate alla classe, piuttosto che alle
istanze.
python

>>> import poker
>>> poker.Card.suits
['Clubs', 'Diamonds', 'Hearts', 'Spades']
>>> poker.Card.ranks[11:]
['Jack', 'Queen', 'King', 'Ace']

python

class Card:
    """Rappresenta una carta da gioco standard"""
    suits = [
        'Clubs', 'Diamonds', 'Hearts', 'Spades'
    ]
    ranks = [
        None, 'Ace', '2', '3', '4', '5',
        '6', '7','8', '9', '10',
        'Jack', 'Queen', 'King', 'Ace'
    ]

    def __init__(self, suit, rank):
```

```
        self.suit = suit
        self.rank = rank

    suits e ranks sono variabili di classe

    suit e rank sono attributi di istanza

python

>>> import poker
>>> queen = poker.Card(1, 12)
>>> queen.suit, queen.rank
(1, 12)
>>> queen.suits
['Clubs', 'Diamonds', 'Hearts', 'Spades']

È anche legale usare l'istanza per accedere alle variabili di classe.
4.2. Insieme totalmente ordinato di oggetti

Un insieme di oggetti è totalmente ordinato se:

    Due elementi qualsiasi possono essere confrontati

    I risultati sono consistenti

Metodo speciale Descrizione
__eq__  Implementa l'operatore ==
__ne__  Implementa l'operatore !=. Se __ne__ non esiste, l'interprete
invoca __eq__ e inverte il risultato
__lt__  Implementa l'operatore <
__gt__  Implementa l'operatore >. Se __gt__ non esiste, l'interprete invoca
__lt__ e inverte il risultato
__le__  Implementa l'operatore <=
__ge__  Implementa l'operatore >=. Se __ge__ non esiste, l'interprete
invoca __le__ e inverte il risultato

I seguenti metodi rendono gli oggetti Card totalmente ordinati.
python

class Card:
    """Rappresenta una carta da gioco standard"""
    suits = [
        'Clubs', 'Diamonds', 'Hearts', 'Spades'
    ]
    ranks = [
        None, 'Ace', '2', '3', '4', '5',
        '6', '7','8', '9', '10',
        'Jack', 'Queen', 'King', 'Ace'
    ]

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
```

```
    def __eq__(self, other):
        return (
            self.suit == other.suit and
            self.rank == other.rank
        )

    def to_tuple(self):
        return (self.suit, self.rank)

    def __lt__(self, other):
        # Il seme è più importante del rango
        # Se i semi sono uguali, confronta i ranghi
        return self.to_tuple() < other.to_tuple()

    def __le__(self, other):
        return self.to_tuple() <= other.to_tuple()
```

4.3. Delegazione

Un metodo che passa la responsabilità a un altro.

poker.py (vedi qui):
python

```
import random

class Deck:
    """Rappresenta un mazzo di carte"""
    def __init__(self, cards):
        self.cards = cards

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

    def make_cards():
        cards = []
        for suit in range(4):
            # Gli assi hanno rango superiore ai re
            for rank in range(2, 15):
                card = Card(suit, rank)
                cards.append(card)
        return cards

    def shuffle(self):
        random.shuffle(self.cards)

    def sort(self):
        self.cards.sort()

    def take_card(self):
        return self.cards.pop()
```

```
    def put_card(self, card):
        self.cards.append(card)

    def move_cards(self, other, num):
        for _ in range(num):
            card = self.take_card()
            other.put_card(card)
```

python

```
>>> import poker
>>> cards = poker.Deck.make_cards()
>>> deck = poker.Deck(cards)
>>> type(deck)
<class 'poker.Deck'>
>>> type(deck.cards)
<class 'list'>
>>> len(deck.cards)
52

>>> import poker
>>> cards = poker.Deck.make_cards()
>>> deck = poker.Deck(cards)
>>> deck.shuffle()
>>> for card in deck.cards[:3]:
...     print(card)
...
Ace of Hearts
King of Diamonds
8 of Hearts
>>> deck.sort()
>>> for card in deck.cards[:3]:
...     print(card)
...
2 of Clubs
3 of Clubs
4 of Clubs
```

Deck.sort non fa alcun lavoro eccetto passare la responsabilità

list.sort usa il metodo __lt__ per ordinare gli oggetti Card

4.4. Genitori e figli

L'ereditarietà è la capacità di definire una nuova classe che è una versione modificata di una classe precedentemente definita.

poker.py (vedi qui):
python

```
class Hand(Deck):
    """Rappresenta una mano di un giocatore"""
```

```
    def __init__(self, label=''):
        self.label = label
        self.cards = []

Hand eredita da Deck:

    Hand è una classe figlia di Deck, che è la classe genitore

    Gli oggetti Hand possono accedere ai metodi definiti in Deck

Hand sovrascrive Deck.__init__:

    L'interprete invoca Hand.__init__ per gli oggetti Hand

python

>>> import poker
>>> cards = poker.Deck.make_cards()
>>> deck = poker.Deck(cards)
>>> card = deck.take_card()
>>> hand = poker.Hand('player 1')
>>> hand.put_card(card)
>>> print(hand)
Ace of Spades

4.4.1. Polimorfismo

Un metodo o operatore che lavora con multiple tipologie di oggetti.
python

>>> import poker
>>> cards = poker.Deck.make_cards()
>>> deck = poker.Deck(cards)
>>> hand = poker.Hand('player 1')
>>> deck.move_cards(hand, 2)
>>> print(hand)
Ace of Spades
King of Spades

Deck.move_card è polimorfico poiché funziona con Deck e Hand.
4.4.2. Principio di sostituzione di Liskov

    Le istanze della classe figlia dovrebbero avere tutti gli attributi
della classe genitore, ma possono averne di aggiuntivi

    La classe figlia dovrebbe avere tutti i metodi della classe genitore,
ma può averne di aggiuntivi

    Se una classe figlia sovrascrive un metodo della classe genitore, il
nuovo dovrebbe prendere gli stessi parametri e restituire un risultato
compatibile

-- Barbara Liskov
```

```
Se segui queste regole, qualsiasi metodo progettato per lavorare con
istanze di una classe genitore funzionerà anche con istanze figlie.
5. Extra
5.1. Set

Collezioni di elementi unici.
python

>>> s = set()
>>> type(s)
<class 'set'>
>>> s = set('hello')
>>> s
{'h', 'e', 'l', 'o'}

5.2. Counter

Come i set, eccetto che un Counter tiene traccia di quante volte gli
elementi appaiono.
python

>>> import collections
>>> c = collections.Counter('hello')
>>> c
Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1})
>>> c['l']
2

Le chiavi devono essere hashable.
5.3. Defaultdict

Come i dizionari, eccetto che un defaultdict genera automaticamente chiavi
che non esistono.
python

>>> import collections
>>> d = collections.defaultdict(list)
>>> d
defaultdict(<class 'list'>, {})
>>> d['zero']
[]
>>> d
defaultdict(<class 'list'>, {'zero': []})

list funziona come una factory.
5.4. Espressioni condizionali

Espressioni che usano condizionali per selezionare uno di due valori.

factorial implementato con un'istruzione condizionale:
python

def factorial(n):
    if n == 0:
```

```
        return 1
    else:
        return n * factorial(n-1)
```

factorial implementato con un'espressione condizionale:
python

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

In generale, le istruzioni condizionali possono essere sostituite con espressioni condizionali se entrambi i rami contengono:

Una singola espressione

Nessuna istruzione

5.5. List comprehension

Modi concisi per iterare attraverso una sequenza e creare una lista.

Loop tradizionale:
python

```
def capitalize_title(title):
    t = []
    for word in title.split():
        t.append(word.capitalize())
    return ' '.join(t)
```

Con una list comprehension:
python

```
def capitalize_title(title):
    t = [
        word.capitalize() for word in title.split()
    ]
    return ' '.join(t)
```

Le parentesi quadre indicano che il risultato è una lista

L'espressione all'interno delle parentesi specifica gli elementi della lista

for indica quale sequenza iterare

Glossario
Termine Significato
Percorso assoluto Un percorso che non dipende dalla directory corrente
Classe figlia Una classe che eredita da un'altra classe
Classe Un tipo definito dal programmatore. Una definizione di classe crea un nuovo oggetto classe
Oggetto classe Un oggetto che rappresenta una classe. Un oggetto classe è il risultato di una definizione di classe

```
Variabile di classe Una variabile definita all'interno di una definizione
di classe, ma non all'interno di alcun metodo
Dati di configurazione  Dati, spesso memorizzati in un file, che
specificano cosa un programma dovrebbe fare e come
Espressione condizionale    Un'espressione che usa un condizionale per
selezionare uno di due valori
Directory di lavoro corrente    La directory predefinita usata da un
programma a meno che non sia specificata un'altra directory
Copia profonda  Un'operazione di copia che copia anche oggetti annidati
Delegazione Quando un metodo passa la responsabilità a un altro metodo per
fare la maggior parte o tutto il lavoro
Deserializzazione   Convertire una stringa in un oggetto
Directory   Una collezione di file e altre directory
Funzione factory    Una funzione usata per creare oggetti, spesso passata
come parametro a una funzione
F-string    Una stringa che ha la lettera f prima delle virgolette di
apertura e contiene una o più espressioni tra parentesi graffe
Ereditarietà    La capacità di definire una nuova classe che è una versione
modificata di una classe precedentemente definita
Istanza Un oggetto che appartiene a una classe
Metodo di istanza   Un metodo che deve essere invocato con un oggetto come
ricevitore
List comprehension  Un modo conciso per iterare attraverso una sequenza e
creare una lista
Metodo  Una funzione che è definita all'interno di una definizione di
classe
Overloading di operatori    Il processo di utilizzare metodi speciali per
cambiare il modo in cui gli operatori lavorano con tipi definiti dal
programmatore
Classe genitore Una classe da cui si eredita
Percorso    Una stringa che specifica una sequenza di directory, spesso che
porta a un file
Polimorfismo    La capacità di un metodo o operatore di lavorare con
multiple tipologie di oggetti
Funzione pura   Una funzione (i) i cui valori di ritorno sono identici per
argomenti identici e (ii) non ha effetti collaterali
Ricevitore  L'oggetto su cui viene invocato un metodo
Percorso relativo   Un percorso che inizia dalla directory di lavoro
corrente, o da qualche altra directory specificata
Serializzazione Convertire un oggetto in una stringa
Copia superficiale  Un'operazione di copia che non copia oggetti annidati
Metodo speciale Un metodo che cambia il modo in cui gli operatori e alcune
funzioni lavorano con un oggetto
Metodo statico  Un metodo che può essere invocato senza un oggetto come
ricevitore
Totalmente ordinato Un insieme di oggetti è totalmente ordinato se due
elementi qualsiasi possono essere confrontati e i risultati sono
consistenti
```