



Python

Programmazione ad oggetti in Python

Argomenti della lezione

- Funzioni
- Moduli
- Programmazione ad oggetti
- Classi
- Ereditarietà
- Overloading degli operatori
- Gestione degli errori
- Test e debugging (debugger IDLE)
- Esercitazioni

Funzioni

Oltre alle funzioni che mette a disposizione Python, è possibile definire nuove funzioni utilizzando la parola chiave ***def***. L'idea base è dare un nome a un blocco di codice, in modo da poterlo richiamare tutte le volte che serve. Inoltre, una funzione può ricevere dati di ingresso e fornire un valore in uscita.

Una volta definita una funzione, è possibile eseguirla (operazione cui ci si riferisce spesso con la locuzione *chiamata di funzione*), passando argomenti diversi a seconda delle necessità. Questo ci permette di rendere il codice più ordinato ed evitare ripetizioni.

La funzione ha due componenti: **intestazione** (o *firma*) e **corpo**. L'intestazione è formata, a sua volta, da 3 parti:

1. la parola chiave **def** (define, definisci);
2. il nome della funzione (che segue le stesse regole dei nomi degli identificatori);
3. l'elenco dei parametri, cioè delle variabili che assumono i valori (o argomenti) passati alla funzione.

```
def                                somma(a,                                b) :  
    """Ritorna la somma di a e b"""  
    return a + b
```

Il corpo della funzione viene eseguito fino all'ultima istruzione oppure fino all'istruzione **return**. Se non viene esplicitamente restituito un valore, e si arriva all'ultima istruzione della funzione, Python ritorna automaticamente **None**, un valore che significa che la funzione non ritorna nulla di significativo.

Nell'esempio precedente troviamo la stringa di documentazione (**docstring**) subito dopo l'intestazione: un commento multilinea utile a descrivere il compito della funzione. Come abbiamo visto, IDLE fornisce un suggerimento sull'uso di una funzione mentre la digiti, visualizzando la sua docstring come call tip. La descrizione completa si ottiene con la funzione `help([object])`:

```
>>> help(somma)
Help on function somma in module __main__:
somma(a, b)
    Ritorna la somma di a e b
```

Esercizio:

Proviamo a definire la funzione `primo(n)` che stabilisce se un numero è primo oppure no. Un numero intero positivo maggiore di 1 si dice primo se è divisibile solo per 1 e per se stesso.

L'esempio di funzione nella slide precedente accetta 2 parametri in input che abbiamo passato secondo una modalità “*posizionale*”, ma in Python esiste più di una modalità per passare i parametri ad una funzione.

In riferimento alla funzione *somma* creata precedentemente che ha la seguente firma:

```
def somma(a, b):
```

possiamo passare i parametri **a** e **b** nelle seguenti modalità:

- **>>> *somma*(3, 4)**

modalità posizionale, in questo caso il corpo della funzione sostituirà **a** con il valore 3 e **b** con il valore 4;

- **>>> *somma*(b=10, a=5)**

per nome: in questa modalità la posizione dei parametri è ininfluente. Nel corpo della funzione, naturalmente, **b** avrà valore 10 mentre **a** 5

- `>>> somma(5, b=14)`

è possibile utilizzare un mix delle due precedenti modalità a patto che gli argomenti passati per posizione precedano quelli passati per nome.

- `>>> addendi = (3, 5)`

```
>>> somma( *addendi )
```

utilizzo di una *sequenza*, in questo caso una tupla: utilizzando il carattere * prima del nome del parametro i valori della sequenza verranno passati, con corrispondenza posizionale, alla funzione.

- `>>> addendi = { 'b': 3, 'a': 5 }`

```
>>> somma( **addendi )
```

utilizzo di un *dizionario*, utilizzando la notazione ** prima del nome del parametro i valori del dizionario verranno passati, seguendo la corrispondenza `nome_parametro = chiave_dizionario`, alla funzione.

Esistono, inoltre, differenti modalità per definire i parametri passati ad una funzione:

- **>>> def somma ()**

è naturalmente possibile creare una funzione che non prende in input alcun valore. In questo caso, presumibilmente, la funzione opererà su un set di informazioni prestabilite e un eventuale valore restituito sicuramente non sarà “influenzabile” da valori esterni;

- **>>> def somma (a, b)**

la firma precedente prevede che la funzione venga chiamata con 2 parametri in ingresso. Se provassimo ad invocare questa funzione con un numero inferiore (o superiore) di argomenti riceveremmo un *TypeError*, perché il numero di argomenti passati deve corrispondere ad i parametri definiti nella firma della funzione;

- **>>> def somma(a, b=10)**

in questo caso, invece, abbiamo aggiunto un valore di default per il parametro **b**: in questo modo abbiamo reso il parametro opzionale. Se **b** non viene passato alla funzione verrà utilizzato il valore assegnato di default (10).

Possiamo rendere tutti i parametri, o parte di essi, opzionali con un valore di default l'importante è che siano tutti preceduti da eventuali parametri senza default;

- **>>> def somma(a, *, b)**

possiamo fare in modo che l'invocazione di una funzione sia possibile solo con parametri passati per nome (tutti o in parte). Utilizzando la notazione *****, tutti i parametri che seguono il simbolo dovranno necessariamente essere passati per nome;

- **>>> def somma(a, b, /)**

è possibile anche fare in modo che tutti i parametri, o in parte, debbano essere passati in modalità posizionale

- `>>> def somma(*addendi)`

è possibile permettere alla funzione di accettare un numero variabile di argomenti utilizzando la notazione `*nome_parametro`, dopo l'invocazione della funzione il parametro `addendi` verrà valorizzato con una tupla che conterrà la lista di valori passati posizionalmente, ad esempio invocando la funzione con:

```
somma(1, 4, 5, 7)
```

all'interno del corpo della funzione `addendi` conterrà la tupla:

```
(1, 4, 5, 7)
```

- `>>> def somma(**addendi)`

utilizzando la notazione `**nome_parametro`, invece, facciamo in modo che la funzione accetti un numero variabile di argomenti passati per nome che saranno salvati all'interno di un dizionario (nel nostro esempio `addendi`)

- **>>> def somma(*addendi)**

è possibile permettere alla funzione di accettare un numero variabile di argomenti utilizzando la notazione ***nome_parametro**, dopo l'invocazione della funzione il parametro **addendi** verrà valorizzato con un tupla che conterrà la lista di valori passati posizionalmente, ad esempio invocando la funzione con:

somma(1, 4, 5, 7)

all'interno del corpo della funziona addendi conterrà la tupla:

(1, 4, 5, 7)

- **>>> def somma(**addendi)**

utilizzando la notazione ****nome_parametro**, invece, facciamo in modo che la funzione accetti un numero variabile di argomenti passati per nome che saranno salvati all'interno di un dizionario (nel nostro esempio addendi)

Una funzione può terminare principalmente per due motivi:

- quando l'esecuzione termina il blocco di istruzioni in essa contenuto
- quando viene eseguita l'istruzione **return**.

Nel primo caso la funzione restituirà automaticamente al chiamante il valore **None** ad indicare che la funzione non restituisce un valore significativo.

L'utilizzo della parola chiave **return** può avere un duplice scopo:

- Terminare l'esecuzione della funzione prematuramente. Magari se si verificano alcune condizioni si potrebbe decidere di non proseguire con l'intera esecuzione della funzione o non si è proprio in grado di farlo. In questo caso, generalmente, viene utilizzata la sola parola chiave **return** e la funzione restituisce al chiamante il valore **None**;
- Restituire uno o più valori al chiamante. In questo caso, come nell'esempio della funzione `somma` vista precedentemente, la keyword **return** è seguita da uno o più valori da restituire separati da una virgola. Quando i valori restituiti sono più di uno al chiamante verrà restituita una tupla composta dai valori di ritorno della funzione.

In Python è possibile utilizzare l'operazione di *unpacking* per assegnare diversi valori a più variabili:

```
>>> x, y = calcolo_coordinate( ... )
>>> a, b = ( 10, 15 )
```

Moduli

I **moduli**, anche conosciuti come librerie in altri linguaggi, sono dei file usati per raggruppare costanti, funzioni e classi, che ci consentono di suddividere e organizzare meglio i nostri progetti. Python include già una lista estensiva di moduli standard (la *standard library*), ma è anche possibile scaricarne o definirne di nuovi.

Per poter utilizzare un modulo all'interno del nostro codice dobbiamo prima importarlo (come fatto per il modulo **turtle**). Ad esempio la seguente riga di codice importa il modulo **math** dalla libreria standard:

```
>>> import math
```

dopo l'esecuzione dell'import Python ha creato la variabile **math** nello scope locale che punta al relativo modulo. Utilizzando il metodo **dir()** possiamo visualizzare gli identificatori presenti nel local scope. Se a **dir(object)** passiamo un parametro è possibile vedere la lista di attributi validi per quell'oggetto.

una volta visualizzato il contenuto della libreria con

```
>>> dir(math)
```

possiamo accedere ad uno di questi attributi con la notazione ***nome_modulo.nome_attributo***, ad esempio:

```
>>> math.pi
```

nel caso di funzioni possiamo utilizzare la funzione built-in di Python per avere informazioni sull'utilizzo; ad esempio per vedere una descrizione della funzione `math.sqrt` possiamo usare il comando:

```
>>> help(math.sqrt)
```

La modalità di import vista ci consente di utilizzare tutto ciò che è presente all'interno del modulo ma, purtroppo, dobbiamo sempre utilizzare la notazione ***nome_modulo.nome_attributo***. In alternativa possiamo importare solo i nomi che ci interessano utilizzando la sintassi

```
>>> from math import pi, sqrt
```

Lo statement precedente aggiungerà allo scope locale `pi` ed `sqrt` dal modulo `math`. In questo modo sarà possibile richiamarli direttamente senza dover anteporre il nome del modulo.

E' anche possibile importare i moduli o gli elementi di un modulo con un identificatore differente da quello originale. Ad esempio, se creiamo o scarichiamo un nuovo modulo che ha lo stesso nome di un altro che stiamo già utilizzando, possiamo utilizzare il costrutto `import modulo as nuovonome` e `from modulo import nome as nuovonome`. In questo modo potremo riferirci al modulo o al suo elemento con l'alias assegnato dopo la keyword `as`

```
>>> import turtle as tartaruga
>>> t = tartaruga.Turtle()
```

Altra possibilità messa a disposizione dal linguaggio è l'import massivo di tutto ciò che è presente all'interno di un modulo con `from modulo import *` ma se ne sconsiglia l'utilizzo.

Python include già dozzine di moduli che coprono la maggior parte delle operazioni più comuni. Tutti questi moduli sono già presenti in Python e possono essere importati direttamente senza dover scaricare nulla. Alcuni tra i moduli più comunemente usati, divisi per aree, sono:

- *elaborazione di testo*: **re** (che fornisce supporto per le espressioni regolari);
- *tipi di dato*: **datetime** (per rappresentare date e ore), **collections** (diversi tipi di oggetti contenitori), **enum** (per enumerazioni);
- *moduli numerici e matematici*: **math** (funzioni matematiche), **decimal** (supporto per aritmetica a virgola mobile), **random** (generazione di numeri pseudo-casuali), **statistics** (funzioni statistiche);
- *persistenza di dati*: **sqlite3** (interfaccia a database SQLite);
- *accesso a file e directory*: **pathlib** (oggetti per la rappresentazione e manipolazione di file e directory), **shutil** (operazioni di alto livello sui file)

- *compressione e archiviazione di dati*: **zipfile** (supporto per archivi ZIP), **tarfile** (supporto per archivi tar), **gzip** (supporto per file gzip);
- *formati di file*: **csv** (supporto per file CSV), **configparser** (supporto per file CFG/INI);
- *servizi generici del sistema operativo*: **os** (diverse interfacce del sistema operativo), **io** (strumenti per lavorare con file e stream), **time** (funzioni relative al tempo), **argparse** (per passare argomenti ricevuti dalla linea di comando), **logging** (funzioni e classi per il logging);
- *esecuzione concorrente*: **subprocess** (gestione di sotto-processi), **multiprocessing** (parallelismo basato su processi), **threading** (parallelismo basato su thread);
- *comunicazione tra processi e networking*: **asyncio** (I/O asincrono), **socket** (interfaccia di rete di basso livello), **ssl** (wrapper TLS/SSL per i socket);
- *gestione di formati di internet*: **json** (supporto per file JSON), **email** (supporto per email)

- *elaborazione di formati di markup*: il package **html** (strumenti per lavorare con HTML), il package **xml** (strumenti per lavorare con xml);
- *protocolli internet*: il package **urllib** (gestione di URL), **httplib** (strumenti per lavorare con il protocollo HTTP), **ftplib** (strumenti per lavorare con FTP);
- *internazionalizzazione*: **gettext** (strumenti per supportare linguaggi multipli);
- *interfacce grafiche*: il package **tkinter** (interfaccia con Tcl/Tk);
- *tool per lo sviluppo*: **unittest** (strumenti per testare il codice);
- *debugging e profiling*: **pdb** (debugger per Python), **timeit** (strumenti per misurare il tempo di esecuzione di brevi pezzi di codice), **cProfile** (profiler per identificare le parti più lente di un programma);
- *servizi di runtime*: **sys** (funzioni e parametri di sistema), **contextlib** (strumenti per lavorare con i context manager);

Se invece necessitiamo di moduli che non sono inclusi nella libreria standard, possiamo consultare **PyPI**: il **Python Package Index**. **PyPI** è un repository che include decine di migliaia di moduli che possono essere scaricati e installati in modo molto semplice usando un tool chiamato **pip**.

Oltre a poter utilizzare moduli di default o di terze parti (scaricati tramite pip) è naturalmente possibile creare i propri moduli. In Python, non esiste una vera distinzione tra i moduli e il file main.

Proviamo a creare un file chiamato **calcolatrice.py** in cui definiamo 4 funzioni per le principali operazioni matematiche. Se proviamo ad eseguire il file, non otteniamo nessun output ma neanche nessun errore. Questo accade perché il nostro programma non esegue nessuna operazione che produce output, ma semplicemente definisce quattro funzioni senza mai invocarle. Eseguendo la funzione **dir()** dovremmo ritrovarci le funzioni aggiunte al nostro local scope.

Volendo aggiungere del codice al nostro modulo che venga eseguito solo quando il modulo è eseguito direttamente e non quando è importato come libreria esterna possiamo utilizzare la variabile speciale `__name__`

Questa variabile viene valorizzata con il valore `__main__` quando il nostro file viene eseguito direttamente e con l'identificatore del modulo quando il file viene importato. Quindi aggiungendo il seguente `if` alla fine della nostra libreria:

```
if __name__ == '__main__':  
    # il file è stato eseguito direttamente
```

possiamo trasformarlo in un programma che può essere eseguito direttamente.

In Python è possibile raccogliere i moduli in **package** che, in genere, corrisponde alla directory che li contiene.

Prendiamo come esempio la struttura di un programma Python per la gestione di una rubrica:

```
RubricaPython/  
|- main.py  
- rubrica/  
  |- __init__.py  
  |- core.py  
  |- gui/  
    |- __init__.py  
    |- window.py  
  |- dao/  
    |- __init__.py  
    |- ...  
  | ...
```

Il nostro progetto è contenuto all'interno della cartella **RubricaPython** che al suo interno ha il file `main.py` che rappresenta il punto di ingresso all'applicazione ed una cartella denominata **rubrica** che, in questo caso, è il nostro package. Come visibile dalla struttura precedente, a partire dalla root del package ogni cartella contiene un file `__init__.py` (vuoto) che serve ad indicare a Python che quella cartella è parte del package.

Per poter importare il package **rubrica** dobbiamo far sì che la cartella che lo contiene (RubricaPython) sia contenuta all'interno della variabile `sys.path`. Per aggiungerlo possiamo lanciare Python3 direttamente dalla cartella di progetto o aggiungere il percorso completo della cartella alla variabile di ambiente `PYTHONPATH`.

Ora possiamo importare i moduli presenti nel package con la sintassi:

```
import rubrica.utils
from rubrica.dao import sql
```

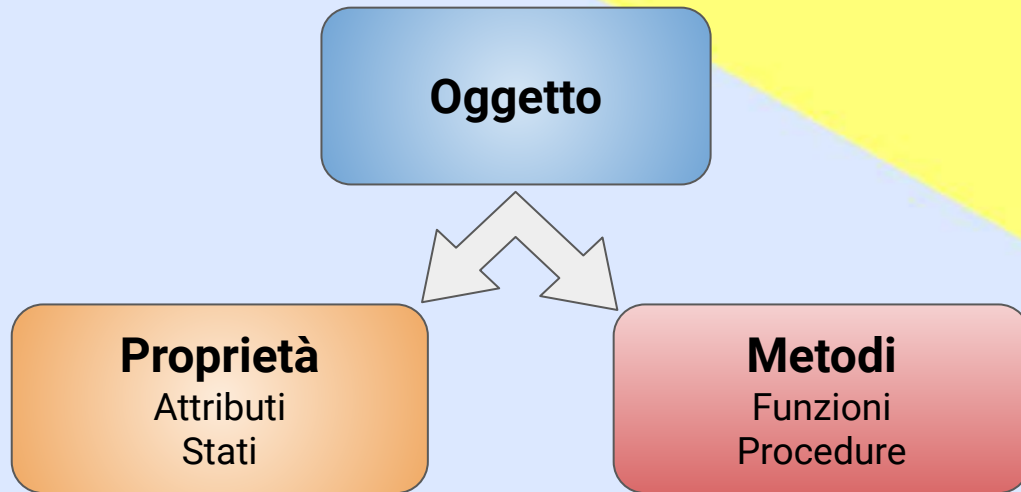
Programmazione ad oggetti

La Programmazione Orientata agli Oggetti (**OOP**) è un paradigma di programmazione che consiste nella definizione di oggetti, in grado di interagire tra di loro attraverso lo scambio di messaggi.

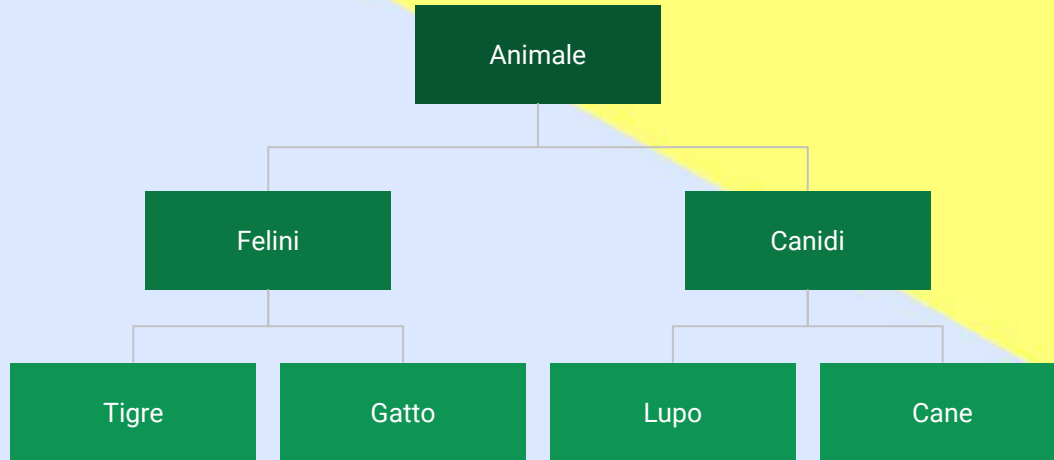
Mentre nella programmazione procedurale il codice viene organizzato, principalmente, per funzioni/procedure nel paradigma **OOP** l'organizzazione prende una direzione più “naturale” in quanto si cerca di astrarre concetti “reali” in entità astratte chiamate “classi” rendendo il codice più leggibile e soprattutto riutilizzabile.

La programmazione orientata agli oggetti prevede, quindi, l'organizzazione di strutture dati e procedure, in contenitori chiamati classi. Una classe è un'astrazione di un modello reale, ed è definita come un tipo di dato astratto. In una classe troviamo **attributi** (dati) e **metodi** (comportamenti).

Un **oggetto** è un'istanza di una **classe**, ovvero un singolo esemplare di una certa categoria. Possono essere presenti più istanze della stessa classe, quindi più oggetti dello stesso tipo.



L'ereditarietà permette di estendere una classe, facendole ereditare le proprietà ed i metodi di un'altra classe ed, eventualmente, ridefinendone altri. Da questo segue che possiamo definire tipi e sottotipi. L'ereditarietà ci permette di vedere le relazioni di parentela tra classi che ereditano dalla stessa superclasse, come un albero radicato.



Rispetto alla programmazione procedurale, la programmazione orientata agli oggetti (OOP) offre vantaggi in termini di:

- **modularità:** le classi sono i moduli del sistema software;
- **coesione dei moduli:** una classe è un componente software ben coeso in quanto rappresentazione di una unica entità;
- **disaccoppiamento dei moduli:** gli oggetti hanno un alto grado di disaccoppiamento in quanto i metodi operano sulla struttura dati interna ad un oggetto; il sistema complessivo viene costruito componendo operazioni sugli oggetti;
- **information hiding:** sia le strutture dati che gli algoritmi possono essere nascosti alla visibilità dall'esterno di un oggetto;
- **riuso:** l'ereditarietà consente di riutilizzare la definizione di una classe nel definire nuove (sotto)classi; inoltre è possibile costruire librerie di classi raggruppate per tipologia di applicazioni;

Classi

Definire una classe in Python è molto semplice, la sintassi ricorda molto quella utilizzata per creare una funzione:

```
>>> class Animale:
    def __init__(self, nome):
        self.nome = nome
    def stampaNome(self):
        print(self.nome)
```

Nel listato precedente è stata creata una classe Animale con l'implementazione dei metodi `__init__` e `stampaNome`.

Il metodo `__init__` è una funzione speciale, messa a disposizione e richiamata da Python, che si occupa di inizializzare lo stato del nostro oggetto (della nostra istanza). Da non confondere con il “costruttore” di altri linguaggi.

`__init__` ha anche un'altra particolarità: gli argomenti passati durante la creazione dell'istanza vengono ricevuti da `__init__`. Questo ci permette di creare automaticamente istanze diverse in base agli argomenti passati.

Il metodo `stampaNome`, invece, è un metodo aggiunto da noi che “opera” sullo stato interno della classe (ne stampa un attributo).

Notate che ogni metodo ha come parametro **`self`**, questo parametro fa riferimento all'istanza corrente e quindi possiamo utilizzarlo all'interno del corpo del metodo per accedere ad attributi ed altri metodi della classe.

Una volta creata una classe è possibile accedere ai suoi metodi ed attributi con la seguente sintassi:

```
>>> cane = Animale('Rex') # Creo una nuova istanza della classe Animale
>>> cane.nome # Accede all'attributo nome
>>> cane.stampaNome() # Invoca il metodo stampa nome
```

Gli attributi di una classe si possono dividere in due categorie:

- **attributi di istanza:** questa categoria di attributi è legata ad una specifica istanza come, ad esempio, l'attributo nome della classe Animale creata precedentemente;
- **attributi di classe:** questa tipologia di attributo è legata all'intera classe di elementi ed può essere acceduto direttamente dalla classe o da una sua istanza.

```
>>> class Persona:
    attributo_classe = 'valore attributo'
    def __init__(self, nome):
        self.nome = nome
```

possiamo accedere ad `attributo_classe` direttamente da `Persona.attributo_classe` e modificandone il valore (da Persona) tutte le eventuali istanze vedrebbero il valore aggiornato.

Attenzione però a non aggiornare il parametro dall'istanza altrimenti otterreste una sovrascrittura del parametro nella relativa istanza!

Come `__init__` Python mette a disposizione altri metodi speciali per le classi come `__str__`, `__repr__`, `__bool__` e `__len__`

Il metodo `__str__` viene invocato automaticamente quando richiamiamo `str(istanza)` su un oggetto o quando chiamiamo una funzione che esegue `str()` come `print()`. Potremmo definirlo in modo che restituisca una rappresentazione in stringa dell'oggetto (ad. es. una classe Persona potrebbe definirlo per restituire la stringa 'Nome Cognome').

anche il metodo `__repr__` restituisce una stringa ma, a differenza di `__str__` che generalmente viene utilizzato per stampare informazioni per l'utente finale, `repr()` è utilizzato principalmente per effettuare operazioni di debug.

Il metodo speciale `__bool__` può essere usato per definire se un oggetto è vero o falso. Se `__bool__` non è definito, Python può usare il risultato di `__len__` per determinare se un oggetto è vero o falso (una lunghezza diversa da 0 è considerata vera). Se anche `__len__` non è definito, l'oggetto è considerato vero.

`__len__`, quindi, viene utilizzato per determinare il numero di elementi di un'istanza. Utile, ad esempio, se la classe gestisce una lista di elementi.

Ereditarietà

L'ereditarietà ci permette di creare una nuova classe a partire da una classe esistente e di estenderla o modificarla.

Ad esempio possiamo creare la classe Animale in questo modo:

```
>>> class Animale:
    def __init__(self, nome):
        self.nome = nome
    def stampaNome(self):
        print(self.nome)
    def stampaVerso(self):
        if hasattr(self, 'verso'):
            print(self.verso)
        else:
            print('---')
```

Con questa prima classe possiamo creare degli oggetti che astraggono un generico animale ma senza poter effettuare delle operazioni proprie di uno specifico animale come il suo verso.

Per ovviare a questo problema possiamo estendere la classe precedente specializzandola

```
>>> class Cane(Animale):  
def __init__(self, nome, verso):  
    super().__init__(nome)  
    self.verso = verso
```

come possiamo vedere dalla subito dopo l'identificatore della classe `Cane` abbiamo dichiarato, tra parentesi tonde, da quale classe vogliamo che erediti attributi e metodi (avremmo potuto anche dichiarare una lista di classi). Inoltre è da notare l'utilizzo del metodo `super()` per ottenere un riferimento della superclasse e sfruttare il suo metodo `__init__`

Overloading degli operatori

Fare l'overloading degli operatori significa definire (o ridefinire) il comportamento di un operatore durante l'interazione con un'istanza di una classe che abbiamo creato in precedenza. Questo ci permette di definire cosa succede quando, ad esempio, utilizziamo una sintassi del tipo `istanza1 + istanza2`.

Per ognuno degli operatori visti precedentemente esiste un corrispondente metodo speciale, che può essere definito per specificare il risultato dell'operazione. Per diversi operatori esistono anche due tipi aggiuntivi di metodi speciali.

Ad esempio, l'operatore `+` ha tre metodi speciali:

- `__add__`: quando eseguiamo *istanza* `+` *valore*, viene in realtà eseguito il metodo `istanza.__add__(valore)`;
- `__radd__`: quando eseguiamo *valore* `+` *istanza*, e il valore non definisce un metodo `__add__` compatibile con la nostra istanza, viene eseguito il metodo `istanza.__radd__(valore)` ;
- `__iadd__`: quando eseguiamo *istanza* `+=` *valore*, viene eseguito `istanza.__iadd__(valore)` , permettendoci di modificare l'istanza in place.

Le tabelle seguenti riassumono gli operatori più comunemente usati e i loro metodi speciali.

Operatori aritmetici

Operatore	Descrizione	Metodi Speciali
+	addizione	<code>__add__</code> , <code>__radd__</code> , <code>__iadd__</code>
-	sottrazione	<code>__sub__</code> , <code>__rsub__</code> , <code>__isub__</code>
*	moltiplicazione	<code>__mul__</code> , <code>__rmul__</code> , <code>__imul__</code>
/	divisione	<code>__truediv__</code> , <code>__rtruediv__</code> , <code>__itruediv__</code>
//	divisione intera	<code>__floordiv__</code> , <code>__rfloordiv__</code> , <code>__ifloordiv__</code>
%	modulo (resto della divisione)	<code>__mod__</code> , <code>__rmod__</code> , <code>__imod__</code>

Operatori di confronto

Operatore	Descrizione	Metodi Speciali
==	uguale a	<code>__eq__</code>
!=	diverso da	<code>__ne__</code>
<	minore di	<code>__lt__</code>
<=	minore o uguale a	<code>__le__</code>
>	maggiore di	<code>__gt__</code>
>=	maggiore o uguale a	<code>__ge__</code>

È importante sapere che esistono alcuni operatori su cui non è possibile effettuare l'overloading, in particolare l'operatore di assegnamento (=), e gli operatori booleani (and, or, not).

Per una lista completa di operatori che possono essere ridefiniti si rimanda alla documentazione ufficiale

<https://docs.python.org/dev/reference/datamodel.html#special-method-names>

Gestione degli errori

Durante l'interazione con IDLE siamo certamente incappati in una serie di errori come quello seguente:

```
>>> ciao
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    ciao
NameError: name 'ciao' is not defined
```

Ogni eccezione ha un tipo (es. **NameError**) e un messaggio che descrive l'errore (es. name 'ciao' is not defined).

Questi tipi sono organizzati in una gerarchia, che include eccezioni più o meno specifiche che vengono utilizzate per situazioni diverse. Per esempio, l'eccezione **ZeroDivisionError** è un caso particolare di **ArithmeticError**, che è un sotto-tipo di **Exception**, che a sua volta è un sotto-tipo di **BaseException**.

Diverse operazioni possono restituire un'eccezione, come quella seguente:

```
>>> n = int('five')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'five'
```

Quando viene sollevata un'eccezione se non è stata gestita dallo sviluppatore il programma mostra un errore e termina.

Per evitare che il nostro programma termini in maniera anomala dobbiamo gestire questi errori tramite il costrutto **try/except**:

```
>>> try:
    n = int('five')
except ValueError:
    print('Invalid number!')
Invalid number!
```


Il funzionamento è semplice: se il codice nel blocco del **try** genera un'eccezione del tipo specificato dall'**except**, allora il blocco dell'**except** viene eseguito per gestirla; se il codice nel blocco del **try** non genera un'eccezione il blocco **except** viene ignorato; in ultimo se il blocco genera un'eccezione di un tipo diverso da quello specificato dall'**except**, allora l'eccezione si propaga e il blocco dell'**except** viene ignorato. È importante notare che l'**except** cattura tutte le eccezioni del tipo specificato, ma anche tutti i suoi **sotto-tipi**.

```
>>>                                     try:
                                     n = 5 / 0
except ZeroDivisionError as err:
    print('Invalid operation ({})!'.format(err))
Invalid operation (division by zero)!
```

È possibile aggiungere dopo l'**except**, la keyword **as** seguita dal nome di una variabile (ad esempio **err**). Questo rende accessibile l'errore all'interno del blocco di codice dell'**except** permettendoci, tra le altre cose, di stamparlo.

È possibile aggiungere più di un **except** in modo da gestire eccezioni diverse in modo diverso. Quando il codice nel blocco del **try** genera un'eccezione, Python eseguire il primo **except** che specifica un'eccezione del tipo corretto.

Inoltre è possibile aggiungere un **else** dopo l'**except** che viene chiamato se il codice nel blocco del **try** viene eseguito senza che sollevi nessuna eccezione.

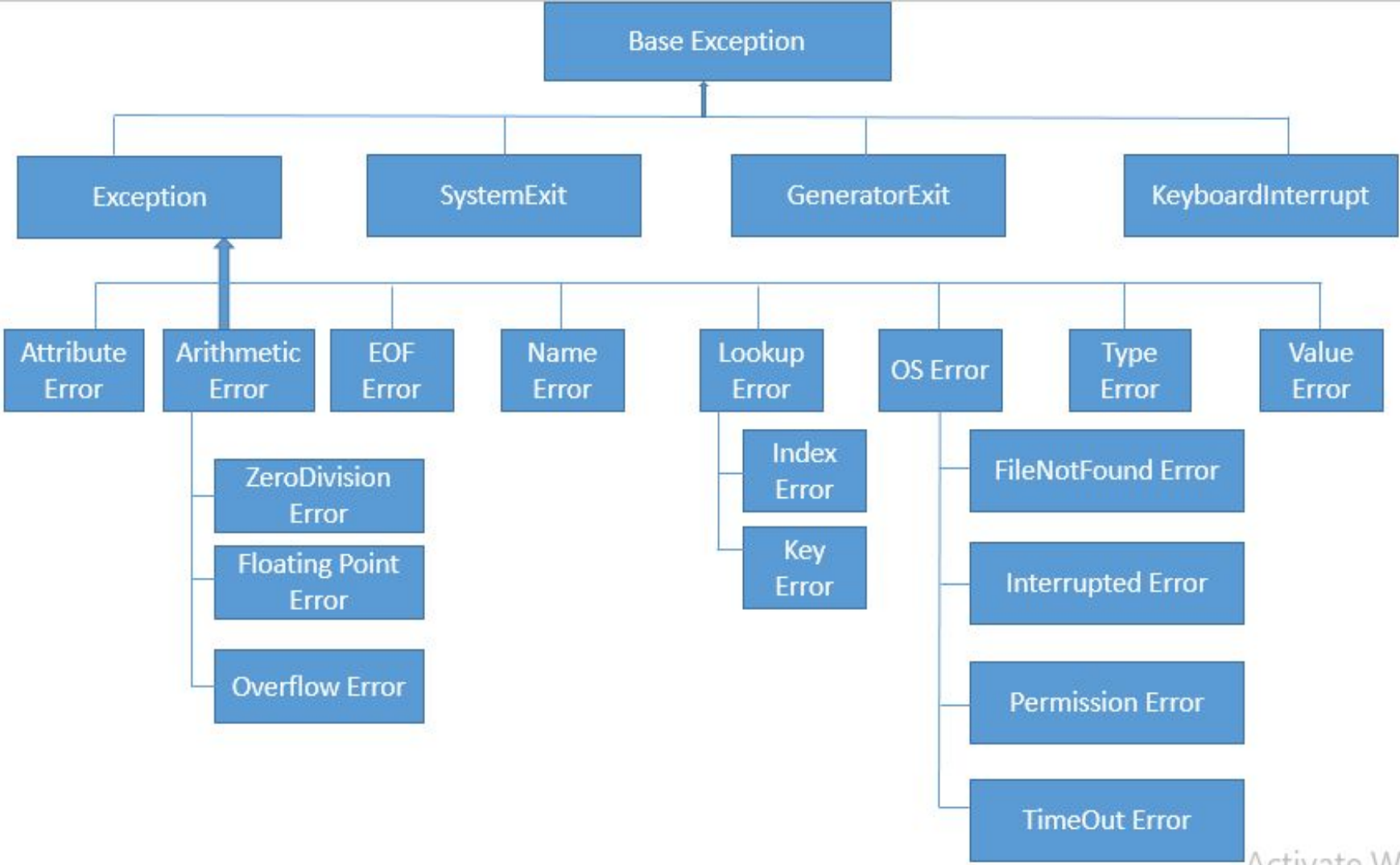
Se vogliamo specificare una o più operazioni che vanno eseguite sia in caso di errore che in caso di successo, possiamo aggiungere un **finally** seguito dai due punti e da un blocco di codice indentato che viene sempre eseguito. È anche possibile aggiungere un **finally** dopo **else/except**.

```
try:
    n = int(x)    # prova a convertire x in intero
except ValueError:
    # eseguito in caso di ValueError
    print('Invalid number!')
except TypeError:
    # eseguito in caso di TypeError
    print('Invalid type!')
else:
    # eseguito se non ci sono errori
    print('Valid number!')
finally:
    # eseguito in ogni caso
    print('End!')
```

Oltre a “gestire” opportunamente un’eccezione a finché essa non provochi la terminazione inaspettata del nostro programma è anche possibile lanciare una nuova eccezione al verificarsi di un determinato evento oppure propagare un’eccezione che abbiamo gestito tramite il costrutto `try/except`. Per farlo viene utilizzata la parola chiave **raise** come mostrato nell’esempio seguente:

```
>>> def div(num, den) :  
    if den == 0 :  
        # se il denominatore è 0 riporta un'eccezione  
        raise ZeroDivisionError('Impossibile dividere per 0')  
    return num / den
```

nel momento in cui viene trovata la keyword **raise** l’esecuzione del blocco di codice viene terminato e l’eccezione propagata al chiamante. Se nella catena di chiamanti l’eccezione propagata non viene gestita in nessun livello il programma verrà terminato.



Test e debugging (debugger IDLE)


L'arte e la tecnica di scoprire e correggere gli errori prende il nome di debugging (o diagnostica).

Purtroppo, delle volte, Python non è molto preciso nell'indicare la riga in cui si è verificato un errore, come nell'esempio che segue:

L'errore è qui: manca una parentesi chiusa.

```
r = float(input("Raggio: "))  
if r > 0:  
    print("Raggio:", r)
```

La segnalazione dell'errore è su questa riga, anche se di per sé è corretta.



Python non trova la parentesi chiusa nella prima riga, per cui la cerca nella riga successiva. Non trovandola neppure qui, si ferma segnalando un errore di sintassi.

Questo esempio conduce al seguente consiglio: *in caso di errore di sintassi, controlla la correttezza dell'istruzione indicata e anche di quelle immediatamente precedenti.*

Una tecnica di debugging spartana ma efficace è quella di stampare il valore di alcune variabili significative nel corso del programma. Questo rende più facile scoprire perché un programma non funziona e anche capire qual è la logica di un codice funzionante scritto da altri. Nel momento in cui questi print non servono più si possono commentare o cancellare.

Riprendiamo il problema del calcolo dell'interesse composto visto nel Capitolo 3. In questa versione, suddividiamo il programma in due funzioni e stampiamo anche una tabella dei valori intermedi, anno e capitale, per capire meglio il funzionamento del programma.


```
"""Calcolo dell'interesse composto con stampa anno per anno"""  
def interesse_composto(n_anni):  
    global capitale # Dichiaro un riferimento alla variabile globale capitale  
    for i in range(n_anni):  
        anno = anno_iniziale + i  
        capitale = capitale*tasso # Aggiungi al capitale gli interessi annuali  
        print_capitale(anno, capitale)  
    return capitale  
def print_capitale(anno, capitale):  
    print("{}\t{}".format(anno, round(capitale, 2)))  
if __name__ == "__main__":  
    capitale = 100  
    tasso = 0.03 # Corrisponde al 3%  
    n_anni = 5  
    anno_iniziale = 2017  
    interesse_composto(n_anni)  
    print("Dopo", n_anni, "anni il capitale è:", round(capitale, 2))
```

Lanciando il programma, ci accorgiamo che il risultato è sbagliato perché la variabile capitale non cresce come dovrebbe:

2017	3.0
2018	0.09
2019	0.0
2020	0.0
2021	0.0
Dopo 5 anni il capitale è: 0.0	

Scoviamo il bug nella riga:

```
capitale = tasso*capitale # Aggiungi al capitale gli interessi annuali
```

Invece di aggiungere l'interesse maturato nell'anno al valore del capitale con l'operatore +=, lo sostituiamo con una normale assegnazione (=). Corretta l'istruzione il programma si comporta come ci aspettiamo. Provare per credere!

IDLE permette di attivare lo strumento di debug integrato in Python (debugger) selezionando nel menu Debug della Python Shell la voce Debugger. Accenniamo solo ad alcune sue funzionalità basilari.

Quando la finestra **Debug Control** è aperta, lanciando un programma, l'esecuzione si blocca alla prima riga e si avvia la modalità di esecuzione passo passo (step by step) che permette di osservare i valori delle variabili, eseguire un'istruzione alla volta, eccetera.

La seguente mostra il debugger che sta eseguendo il programma sul calcolo dell'interesse composto appena visto. Leggiamo dall'alto al basso alcune informazioni presenti nella finestra di debug: sotto i pulsanti viene indicata che l'istruzione corrente si trova nel programma `interesse_composto2.py` alla riga 13 nella funzione `print_capitale()`.

Debug Control

① Go ② Step ③ Over ④ Out ⑤ Quit ⑥ Stack ⑦ Source ⑧ Locals ⑨ Globals

interesse_composto2.py:13: print_capitale()

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
'__main__'.<module>(), line 20: interesse_composto(n_anni)
'__main__'.interesse_composto(), line 8: print_capitale(anno, capitale)
> '__main__'.print_capitale(), line 13: print("{}\t{}".format(anno, round(capitale, 2
```

Locals

anno 2017
capitale 3.0

Globals

__builtins__	<module 'builtins' (built-in)>
__doc__	"Calcolo dell'interesse co...to con stampa anno per anno"
__file__	'/home/mauri/Apogeo/Imparar...ammi/interesse_composto2.py'
__loader__	<class 'frozen_importlib.BuiltinImporter'>
__name__	'__main__'
__package__	None
__spec__	None
anno_iniziale	2017
capitale	3.0
interesse_composto	<function interesse_composto at 0x7fe7339dd400>
n_anni	5
print_capitale	<function print_capitale at 0x7fe739af1378>
tasso	0.03

Ecco maggiori dettagli di quanto presente nella figura precedente.

1. Go: continua l'esecuzione normalmente.
2. Step: esegue un'istruzione e si ferma prima di eseguire la successiva.
3. Over: esegue una funzione in un unico passo.
4. Out: esce dalla funzione corrente, ritornando alla funzione chiamante.
5. Quit: ferma il programma.
6. Stack: visualizza lo stack delle chiamate di funzione.
7. Source: evidenzia nel programma sorgente Python la prossima istruzione da eseguire.
8. Locals: visualizza le variabili locali.
9. Globals: visualizza le variabili globali.
10. Stack delle chiamate di funzione.
11. Lista delle variabili locali della funzione corrente nel debug.
12. Lista delle variabili globali.

Proseguendo l'osservazione della Figura 8.4, troviamo lo stack (o pila) delle chiamate. Il modulo principale (**__main__**) alla linea 20 ha richiamato la funzione **interesse_composto()**, che alla linea 8, a sua volta, ha richiamato la funzione **print_capitale()**. I due riquadri successivi visualizzano nell'ordine:

- le variabili locali (*Locals*) definite nella funzione corrente (anno e capitale);
- le variabili globali (*Globals*) definite all'esterno delle funzioni (**anno_iniziale**, capitale, ecc.) e le funzioni definite nel programma (**interesse_composto()** e **print_capitale()**).

I nomi che iniziano col doppio underscore “__” (come **__main__**) sono parole riservate di Python per oggetti e attributi interni al linguaggio.

Esercitazioni

Sviluppare un **package**, seguendo il paradigma **OOP**, per la gestione di una **libreria**.

La libreria dovrà gestire l'inserimento, la visualizzazione e la vendita di Libri, DVD e Riviste implementando rispettivamente le classi **Book**, **Dvd** e **Magazine**.

Tutti questi prodotti devono avere almeno un identificativo numerico, un titolo ed un prezzo. Si richiede, inoltre, di gestire l'autore per i libri, lo studio di produzione per i DVD e l'editoriale per le riviste (gestire l'informazione con una stringa in tutti e tre i casi).

Inoltre si dovranno implementare i seguenti metodi per la libreria:

1. `libreria.warehouse()`: stampa la lista aggiornata di libri con rispettiva disponibilità in magazzino. Se il magazzino non dispone di copie del prodotto mostrare il titolo seguito dal testo "non disponibile";

2. `libreria.buy()`: la funzionalità dovrà far selezionare all'utente la categoria, tra quelle presenti (Libro, DVD, Rivista) per la quale intende acquistare un prodotto. Alla selezione mostrerà l'elenco di prodotti disponibili con la relativa disponibilità in magazzino, richiedendo all'utente di selezionare l'id del prodotto che vuole acquistare.

Se il prodotto è disponibile in magazzino dovrà essere decrementato il numero di elementi disponibili e mostrato all'utente il messaggio "Grazie per aver acquistato <Titolo prodotto>. Il totale da pagare è <Prezzo prodotto>". Nel caso in cui il prodotto non sia presente all'interno del magazzino dovrà essere mostrato il messaggio "Siamo spiacenti ma il prodotto richiesto è attualmente terminato". Inoltre si dovrà controllare che l'input inserito dall'utente sia effettivamente un numero, in caso contrario dovrà essere sollevata un'eccezione invitando l'utente ad inserire nuovamente l'id del prodotto.

Infine se l'id del prodotto non è presente tra i prodotti in vendita in quella categoria si dovrà mostrare il messaggio "Prodotto non trovato".

3. `libreria.add(Prodotto, n)`: dovrà essere previsto un metodo che inserisca all'interno della libreria un prodotto con la relativa disponibilità.
Il metodo dovrebbe controllare la presenza di un prodotto con lo stesso progressivo ed eventualmente impedirne l'inserimento mostrando un messaggio in cui si motiva il mancato inserimento.

Il **package** dovrà funzionare utilizzando lo script **main.py** fornito nella cartella **22_libreria** del repository del corso.