Guida Introduttiva alla Programmazione Orientata agli Oggetti con Python (OOP)

La programmazione è un'arte. E come nell'arte, selezionare i pennelli e i colori adeguati è essenziale per produrre le opere migliori. La programmazione orientata agli oggetti Python rientra in questa categoria.

Scegliere il giusto linguaggio di programmazione è una parte cruciale di qualsiasi progetto, e può portare sia a uno sviluppo fluido e piacevole che a un completo incubo. È per questo che sarebbe meglio usare il linguaggio più adatto al vostro caso d'uso.

Questa è la ragione principale per imparare la programmazione orientata agli oggetti in Python, che è anche uno dei linguaggi di programmazione più popolari.

Iniziamo!

Un Esempio di Programma Python

Prima di addentrarci nell'argomento, iniziamo con una domanda: avete mai scritto un programma Python come quello qui sotto?

```
secret_number = 20

while True:
   number = input('Guess the number: ')

   try:
       number = int(number)
   except:
       print('Sorry that is not a number')
       continue

   if number != secret_number:
       if number > secret_number:
           print(number, 'is greater than the secret number')

       elif number < secret_number:
           print(number, 'is less than the secret number')
   else:
       print('You guessed the number:', secret_number)
       break
```

Questo codice è un semplice strumento per indovinare i numeri. Provate a copiarlo in un file Python e a eseguirlo nel vostro sistema. Compie perfettamente il suo scopo.

Ma qui arriva un enorme problema: cosa succederebbe se vi chiedessimo di implementare una nuova funzionalità? Potrebbe essere qualcosa di semplice, per esempio:

“Se l’input è un multiplo del numero segreto, dai un suggerimento all’utente”.

Il programma diventerebbe presto complesso e pesante all’aumentare del numero di funzioni e, quindi, del numero totale di condizionali annidate.

Questo è esattamente il problema che la programmazione orientata agli oggetti cerca di risolvere.

Requisiti per Imparare Python OOP

Prima di addentrarsi nella programmazione orientata agli oggetti (OOP, object-oriented programming), vi consigliamo vivamente di avere una solida conoscenza delle basi di Python.

Classificare gli argomenti considerati “di base” può essere difficile. Per questo abbiamo progettato un cheat sheet con tutti i principali concetti necessari per imparare la programmazione orientata agli oggetti in Python.

- **Variabile:** Nome simbolico che punta a un oggetto specifico (vedremo cosa significa il termine **oggetto** nel corso dell’articolo).
- **Gli operatori aritmetici:** Addizione (+), sottrazione (-), moltiplicazione (*), divisione (/), divisione intera (//), modulo (%).
- **Tipi di dati incorporati:** Numerici (interi, float, complessi), sequenze (stringhe, liste, tuple), booleani (vero, falso), dizionari e insiemi.
- **Espressioni booleane:** Espressioni in cui il risultato è **Vero** o **Falso.**
- **Condizionale:** Valuta un’espressione booleana e genera qualche processo a seconda del risultato. Gestito da dichiarazioni **if/else**.
- **Loop:** Esecuzione ripetuta di blocchi di codice. Possono essere loop **for** o **while** .
- **Funzioni:** Blocco di codice organizzato e riutilizzabile. Si creano con la parola chiave **def**.
- **Argomenti:** Oggetti passati a una funzione. Per esempio: `sum([1, 2, 4])`
- **Eseguire uno script Python**: Aprire un terminale o riga di comando e digitare “python .
- **Aprire una shell Python**: Aprire un terminale e digitare `python` o `python3` a seconda del vostro sistema.

Ora che avete questi concetti cristallini, potete andare avanti con la comprensione della programmazione orientata agli oggetti.

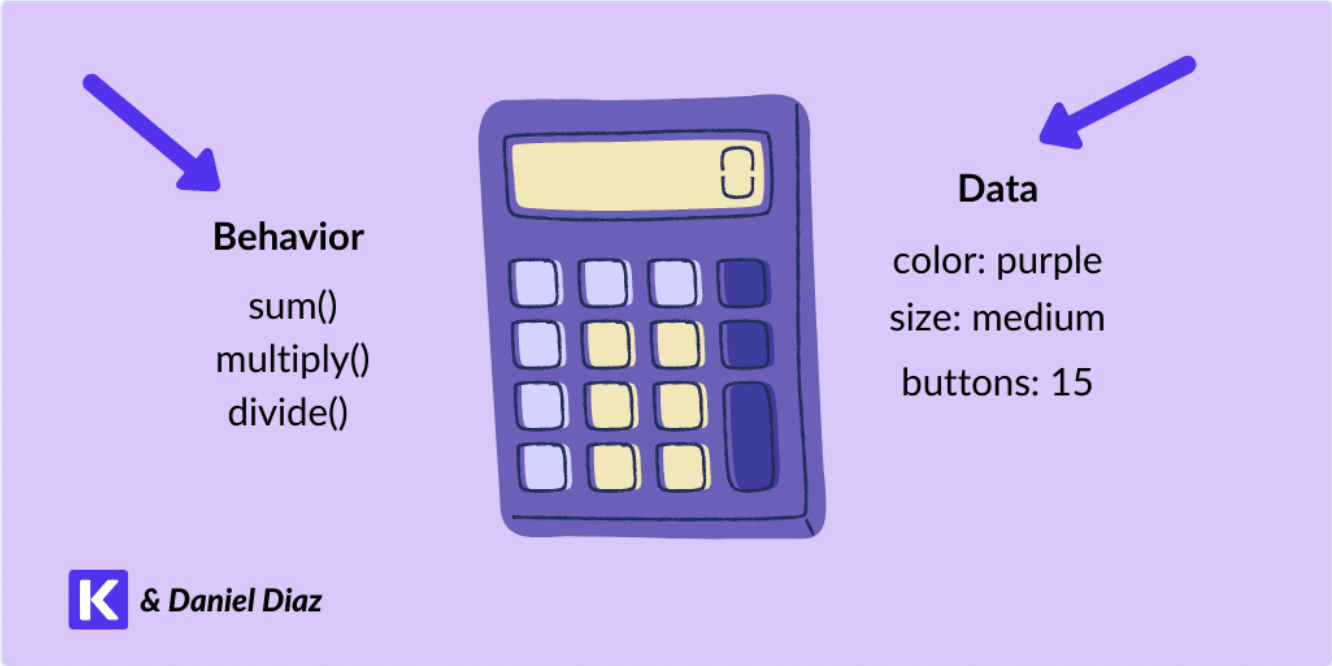
Cos’È la Programmazione Orientata agli Oggetti in Python?

La Programmazione Orientata agli Oggetti (OOP) è un paradigma di programmazione in cui possiamo pensare a problemi complessi come oggetti.

Un paradigma è una teoria che fornisce la base per risolvere i problemi.

Quindi quando parliamo di OOP, ci riferiamo a un insieme di concetti e modelli che usiamo per risolvere problemi con oggetti.

Un oggetto in Python è una singola collezione di dati (attributi) e comportamenti (metodi). Potete pensare agli oggetti come a cose reali che vi circondano. Per esempio, considerate le calcolatrici:



— Una calcolatrice può essere un oggetto.

Come potete notare, i dati (attributi) sono sempre sostantivi, mentre i comportamenti (metodo) sono sempre verbi.

Questa compartimentazione è il concetto centrale della programmazione orientata agli oggetti. Si costruiscono oggetti che immagazzinano dati e contengono specifici tipi di funzionalità.

Perché Usiamo la Programmazione Orientata agli Oggetti in Python?

OOP permette di creare software sicuro e affidabile. Molti framework e librerie Python usano questo paradigma per costruire il loro codice base. Alcuni esempi sono Django, Kivy, pandas, NumPy e TensorFlow.

Vediamo i principali vantaggi dell'uso di OOP in Python.

Vantaggi di Python OOP

Le seguenti ragioni vi faranno optare per l'uso della programmazione orientata agli oggetti in Python.

Tutti i Linguaggi di Programmazione Moderni Usano OOP

Questo paradigma è indipendente dalla lingua. Se imparate l'OOP in Python, sarete in grado di usarlo anche in:

- Java
- PHP (assicuratevi di leggere il confronto tra PHP e Python)
- Ruby
- Javascript
- C#
- Kotlin

Tutti questi linguaggi sono nativamente orientati agli oggetti o includono opzioni per funzionalità orientate agli oggetti. Se volete imparare uno di questi dopo aver appreso Python, sarà più facile perché troverete molte somiglianze tra i linguaggi che lavorano con gli oggetti.

OOP Vi Permette di Codificare Più Velocemente

Codificare più velocemente non significa scrivere meno righe di codice. Significa che potete implementare più funzioni in meno tempo senza compromettere la stabilità di un progetto.

La programmazione orientata agli oggetti permette di riutilizzare il codice implementando l' astrazione. Questo principio rende il vostro codice più conciso e leggibile.

Come forse sapete, i programmatori passano molto più tempo a leggere il codice che a scriverlo. È la ragione per cui la leggibilità è sempre più importante che pubblicare le caratteristiche il più velocemente possibile.

— La produttività diminuisce con codice non leggibile

Vedrete più avanti il principio di astrazione.

OOP Vi Aiuta e Evitare il "Codice Spaghetti"

Vi ricordate il programma indovina numeri scritto all'inizio di questo articolo?

Se continuate ad aggiungere funzioni, in futuro avrete molte dichiarazioni **if** annidate. Questo groviglio di linee di codice infinite è chiamato codice spaghetti, e si dovrebbe evitare il più possibile.

OOP ci dà la possibilità di comprimere tutta la logica in oggetti, evitando così lunghi pezzi di **if** annidati.

OOP Migliora l'Analisi di Qualsiasi Situazione

Una volta acquisita un po' di esperienza con l'OOP, sarete in grado di pensare ai problemi come a piccoli e specifici oggetti.

Questa comprensione porta a una rapida inizializzazione del progetto.

Programmazione Strutturata vs. Programmazione Orientata agli Oggetti

La programmazione strutturata è il paradigma più usato dai principianti perché è il modo più semplice per costruire un piccolo programma.

Implica l'esecuzione di un programma Python in modo sequenziale. Questo significa che state dando al computer una lista di compiti e poi li eseguite dall'alto verso il basso.

Vediamo un esempio di programmazione strutturata con un programma dedicato a chi vuole comprare caffè:

```
small = 2
regular = 5
big = 6

user_budget = input('What is your budget? ')

try:
```

```
    user_budget = int(user_budget)
except:
    print('Please enter a number')
    exit()

if user_budget > 0:
    if user_budget >= big:
        print('You can afford the big coffee')
        if user_budget == big:
            print('It\'s complete')
        else:
            print('Your change is', user_budget - big)
    elif user_budget == regular:
        print('You can afford the regular coffee')
        print('It\'s complete')
    elif user_budget >= small:
        print('You can buy the small coffee')
        if user_budget == small:
            print('It\'s complete')
        else:
            print('Your change is', user_budget - small)
```

Il codice qui sopra agisce come una persona che vende caffè. Vi chiederà un budget, poi vi "venderà" il caffè più grande che siete in grado di comprare.

Provate a eseguirlo nel terminale. Verrà eseguito passo dopo passo, a seconda del vostro input.

Questo codice funziona perfettamente, ma abbiamo tre problemi:

1. Include un sacco di logica ripetuta.
2. Usa molti condizionali **if** annidati.
3. Sarà difficile da leggere e modificare.

La OOP è stata inventata come soluzione a tutti questi problemi.

Vediamo il programma di cui sopra implementato con OOP. Non preoccupatevi se non lo capite ancora. È solo per confrontare la programmazione strutturata e la programmazione orientata agli oggetti.

```
class Coffee:
        # Constructor
        def __init__(self, name, price):
                self.name = name
                self.price = float(price)
        def check_budget(self, budget):
                # Check if the budget is valid
                if not isinstance(budget, (int, float)):
                        print('Enter float or int')
                        exit()
                if budget = self.price:
                        print(f'You can buy the {self.name} coffee')
                        if budget == self.price:
                                print('It\'s complete')
                        else:
                                print(f'Here is your change {self.get_cha

                        exit('Thanks for your transaction')
```

Nota: tutti i concetti che seguono saranno spiegati più a fondo nell'articolo.

Il codice qui sopra rappresenta una **classe** chiamata "Caffè". Ha due attributi – "name" e "price" – ed entrambi sono usati nei metodi. Il metodo principale è "sell", che elabora tutta la logica necessaria per completare il processo di vendita.

Se provate a eseguire questa classe, non otterrete alcun output. Ciò accade principalmente perché stiamo solo dichiarando il "modello" per i caffè, non i caffè stessi.

Implementiamo questa classe con il seguente codice:

```
small = Coffee('Small', 2)
regular = Coffee('Regular', 5)
big = Coffee('Big', 6)

try:
   user_budget = float(input('What is your budget? '))
except ValueError:
   exit('Please enter a number')

for coffee in [big, regular, small]:
   coffee.sell(user_budget)
```

Qui stiamo creando **istanze** o oggetti caffè, della classe “Coffee”, poi chiamiamo il metodo “sell” di ogni caffè finché l’utente non può permettersi qualsiasi opzione.

Otterremo lo stesso risultato con entrambi gli approcci, ma possiamo estendere la funzionalità del programma molto meglio con OOP.

Qui sotto c’è una tabella che confronta la programmazione orientata agli oggetti e la programmazione strutturata:

OOP	**Programmazione strutturata**
Più facile da mantenere	Più complessa da mantenere
Approccio DRY (Don’t Repeat Yourself)	Codice ripetuto in molti posti
Piccoli pezzi di codice riutilizzati in molti posti	Una grande quantità di codice in pochi posti
Approccio a oggetti	Approccio a blocchi di codice
Più facile fare il debug	Più difficile da debuggare
Ripida curva di apprendimento	Curva di apprendimento più semplice
Usato in grandi progetti	Ottimizzato per programmi semplici

Per concludere il confronto dei paradigmi:

- Nessuno dei due paradigmi è perfetto (OOP può essere pesante da usare in progetti semplici).

- Questi sono solo due modi di risolvere un problema; ce ne sono altri là fuori.
- OOP è usato in grandi codebase, mentre la programmazione strutturata è principalmente per progetti semplici.

Passiamo agli oggetti incorporati in Python.

In Python Tutto È un Oggetto

Vi diremo un segreto: avete usato OOP tutto il tempo senza accorgervene.

Anche quando usate altri paradigmi in Python, state ancora usando gli oggetti per fare quasi tutto.

Questo perché, in Python, *tutto* è un oggetto.

Ricordate la definizione di oggetto: un oggetto in Python è una singola collezione di dati (attributi) e comportamenti (metodi).

Questo corrisponde a qualsiasi tipo di dati in Python.

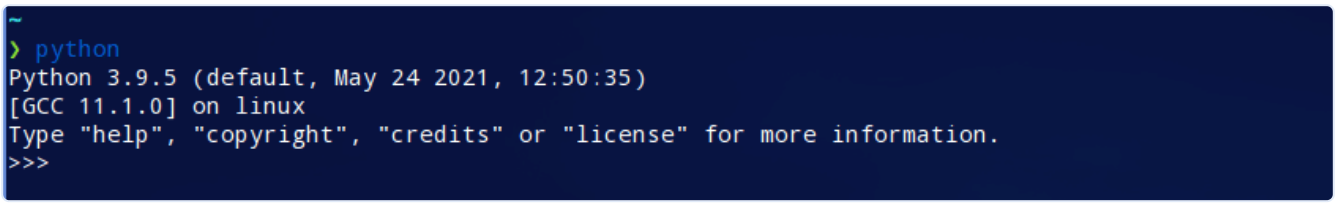
Una stringa è una collezione di dati (caratteri) e comportamenti (**upper()**, **lower()**, ecc.). Lo stesso vale per i numeri **integrali**, **float**, **boolean**, le **liste** e per i dizionari.

Prima di continuare, rivediamo il significato degli attributi e dei metodi.

Attributi e Metodi

Gli attributi sono **variabili interne** agli oggetti, mentre i metodi sono **funzioni** che producono un certo comportamento.

Facciamo un semplice esercizio nella shell di Python. Potete aprirla digitando `python` o `python3` nel vostro terminale.



— Python shell

Ora lavoriamo con la shell Python per scoprire metodi e tipi.

```
>>> kinsta = 'Kinsta, Premium Application, Database, and Managed WordPres
>>> kinsta.upper()
'KINSTA, PREMIUM APPLICATION, DATABASE, AND MANAGED WORDPRESS HOSTING'
```

Nella seconda riga, stiamo chiamando un metodo di stringa, **upper()**. Restituisce il contenuto della stringa tutto in maiuscolo senza cambiare la variabile originale.

```
>>> kinsta
'Kinsta, Premium Application, Database, and Managed WordPress hosting'
```

Approfondiamo le funzioni più preziose quando si lavora con gli oggetti.

La funzione **type()** permette di ottenere il tipo di un oggetto. Il “tipo” è la classe a cui appartiene l’oggetto.

```
>>> type(kinsta)
# class 'str'
```

La funzione **dir()** restituisce tutti gli attributi e i metodi di un oggetto. Proviamola con la variabile **kinsta**.

```
>>> dir(kinsta)
['__add__', '__class__',  ..... 'upper', 'zfill']
```

Ora provate a pubblicare alcuni degli attributi nascosti di questo oggetto.

```
 >>> kinsta.__class__ # class 'str' e>
```

Questo mostrerà la classe a cui appartiene l'oggetto **kinsta**. Quindi possiamo dire che l'unica cosa che la funzione **type** restituisce è l'attributo __**class**__ di un oggetto.

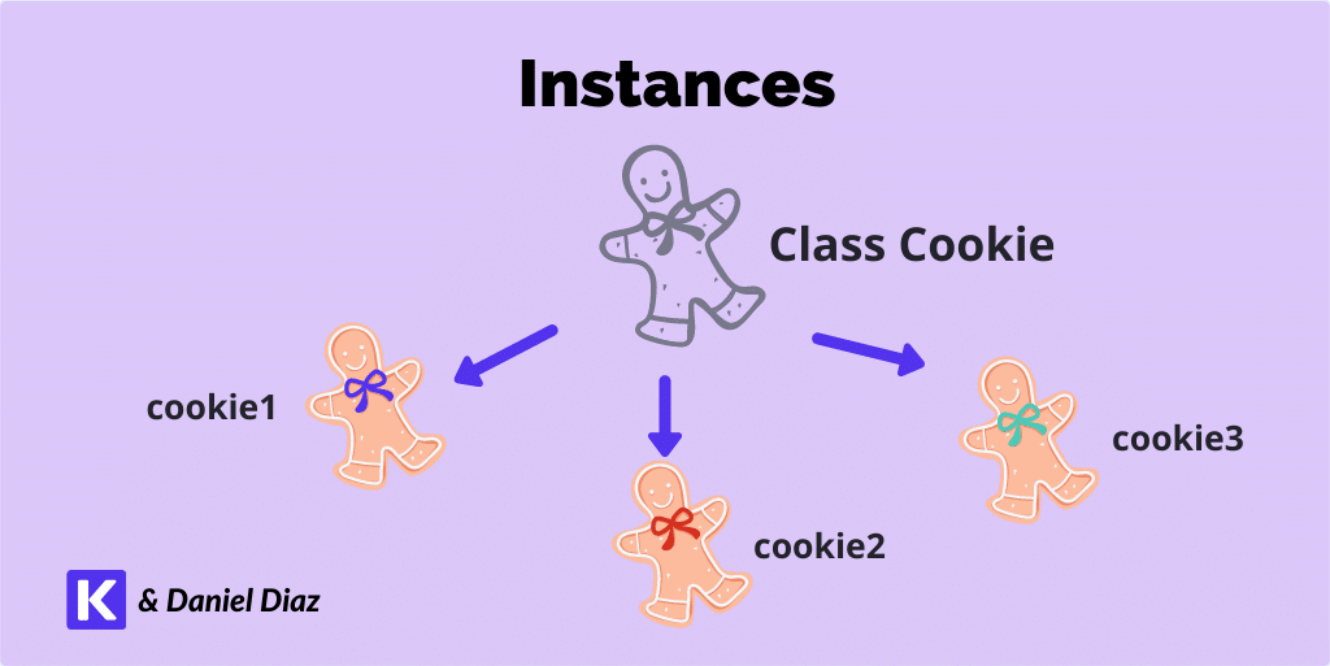
Potete sperimentare tutti i tipi di dati, scoprendo tutti i loro attributi e metodi direttamente sul terminale. Potete imparare di più sui tipi di dati incorporati nella documentazione ufficiale.

Il Vostro Primo Oggetto in Python

Una **classe** è come un template. Vi permette di creare oggetti personalizzati basati sugli attributi e i metodi che definite.

Potete pensarla come uno **stampino per biscotti** che modificate per cuocere i biscotti perfetti (oggetti, non cookie di tracciamento), con caratteristiche definite: forma, dimensione e altro.

Dall'altra parte, abbiamo le **istanze**. Un'istanza è un oggetto individuale di una classe, che ha un indirizzo di memoria unico.



— Istanze in Python

Ora che sapete cosa sono le classi e le istanze, definiamone alcune!

Per definire una classe in Python, si usa la parola chiave **class**, seguita dal suo nome. In questo caso, creerete una classe chiamata **Cookie**.

Nota: in Python, usiamo la convenzione Camel Case per i nomi delle classi.

```
class Cookie:
    pass
```

Aprite la vostra shell Python e digitate il codice qui sopra. Per creare un'istanza di una classe, basta digitare il suo nome e la parentesi dopo di esso. È lo stesso processo che invocare una funzione.

```
cookie1 = Cookie()
```

Congratulazioni: avete appena creato il vostro primo oggetto in Python! Potete controllare il suo id e il suo tipo con il seguente codice:

```
id(cookie1)
140130610977040 # Unique identifier of the object

type(cookie1)
```

Come potete vedere, questo cookie ha un identificatore unico in memoria, e il suo tipo è **Cookie**.

Potete anche controllare se un oggetto è un'istanza di una classe con la funzione **isinstance()**.

```
isinstance(cookie1, Cookie)
# True
isinstance(cookie1, int)
# False
isinstance('a string', Cookie)
# False
```

Metodo Costruttore

Il metodo **__init__()** è anche chiamato “costruttore”: viene richiamato Python ogni volta che istanziamo un oggetto.

Il costruttore crea lo stato iniziale dell’oggetto con l’insieme minimo di parametri di cui ha bisogno per esistere. Modifichiamo la classe **Cookie**, in modo che accetti parametri nel suo costruttore.

```
class Cookie:
    # Constructor
    def __init__(self, name, shape, chips='Chocolate'):
        # Instance attributes
        self.name = name
        self.shape = shape
        self.chips = chips
```

Nella classe **Cookie**, ogni biscotto deve avere un nome, una forma e dei chip. Abbiamo definito l’ultimo come “Chocolate”.

D’altra parte, **self** si riferisce all’istanza della classe (l’oggetto stesso).

Provate a incollare la classe nella shell e create un’istanza del cookie come al solito.

```
cookie2 = Cookie()
# TypeError
```

Otterrete un errore. Questo perché dovete fornire l'insieme minimo di dati di cui l'oggetto ha bisogno per vivere; in questo caso, **name** e **shape** dato che abbiamo già impostato **chips** su "Chocolate".

```
cookie2 = Cookie('Awesome cookie', 'Star')
```

Per accedere agli attributi di un'istanza, dovete usare la notazione per punti.

```
cookie2.name
# 'Cookie impressionante'
cookie2.shape
# 'Stella'.
cookie2.chips
# 'Cioccolato'
```

Per ora, la classe **Cookie** non include niente di troppo succoso. Aggiungiamo un metodo di esempio **bake()** per rendere le cose più interessanti.

```
classe Cookie:
    # Costruttore
    def __init__(self, name, shape, chips='Chocolate'):
        # Attributi dell'istanza
        self.name = nome
        self.shape = forma
        self.chips = chips

    # L'oggetto passa se stesso come parametro
    def bake(self):
        print(f'Questo {self.nome}, viene cotto con la forma {self.forma}
        print('Goditi il tuo biscotto!')
```

Per richiamare un metodo, usate la notazione per punti e richiamatelo come una funzione.

```
cookie3 = Cookie('Baked cookie', 'Tree')
cookie3.bake()
# This Baked cookie, is being baked with the shape Tree and chips of Choc
Enjoy your cookie!
```

I 4 Pilastri della OOP in Python

La programmazione orientata agli oggetti comprende quattro pilastri principali:

1. Astrazione

L’astrazione nasconde la funzionalità interna di un’applicazione all’utente. L’utente potrebbe essere il cliente finale o altri sviluppatori.

Possiamo trovare l'**astrazione** nella nostra vita quotidiana. Per esempio, sapete come usare il vostro telefono, ma probabilmente non sapete esattamente cosa succede al suo interno ogni volta che aprite un'applicazione.

Un altro esempio è Python stesso. Sapete come usarlo per costruire software funzionale, e potete farlo anche se non capite il funzionamento interno di Python.

Applicare lo stesso concetto al codice vi permette di raccogliere tutti gli oggetti in un problema e **astrarre** le funzionalità standard in classi.

2. Ereditarietà

L'ereditarietà ci permette di definire più **sottoclassi** da una classe già definita.

Il suo scopo principale è quello di seguire il principio di DRY. Sarete in grado di riutilizzare un sacco di codice implementando tutti i componenti di condivisione in **superclassi**.

Potete pensarlo come il concetto concreto di **eredità genetica**. I figli (sottoclassi) sono il risultato dell'eredità tra due genitori (superclassi). Essi ereditano tutte le caratteristiche fisiche (attributi) e alcuni comportamenti comuni (metodi).

3. Polimorfismo

Il polimorfismo ci permette di modificare leggermente i metodi e gli attributi delle **sottoclassi** precedentemente definite nella **superclasse**.

Il significato letterale di polimorfismo è "**molteplici forme**". Questo perché costruiamo metodi con lo stesso nome ma con funzionalità diverse.

Tornando all'idea precedente, anche i bambini sono un perfetto esempio di polimorfismo. Possono ereditare un comportamento definito **get_hungry()** ma in un modo leggermente diverso, per esempio, avere fame ogni 4 ore invece che ogni 6.

4. Incapsulamento

L'incapsulamento è il processo con cui proteggiamo l'integrità interna dei dati in una classe.

Anche se non c'è una dichiarazione **private** in Python, potete applicare l'incapsulamento usando il name mangling in Python. Ci sono metodi speciali chiamati **getters** e **setters** che ci permettono di accedere ad attributi e metodi unici.

Immaginiamo una classe **Human** che ha un attributo unico chiamato _**height**. È possibile modificare questo attributo solo entro certi vincoli (è quasi impossibile per una persona superare l'altezza di 3 metri).

Costruire un Calcolatore di Area per Forme Diverse

Una delle cose migliori di Python è che ci permette di creare una grande varietà di software, da un programma CLI (interfaccia a riga di comando) a una complessa web app.

Ora che avete imparato i concetti pilastro dell'OOP, è il momento di applicarli a un progetto reale.

Nota: tutto il seguente codice sarà disponibile all'interno di questo repository di GitHub. Uno strumento di revisione del codice che ci aiuta a gestire le versioni del codice con Git.

Il vostro compito è quello di creare un calcolatore di aree per le seguenti forme:

- Quadrato
- Rettangolo
- Triangolo
- Cerchio
- Esagono

Classe di Base per le Forme

Per prima cosa, create un file **calculator.py** e apritelo. Dato che abbiamo già gli oggetti con cui lavorare, sarà facile **astrarli** in una classe.

Potete analizzare le caratteristiche comuni e scoprire che sono tutte **forme 2D**. Pertanto, l'opzione migliore è creare una classe **Shape** con un metodo **get_area()** da cui ogni forma erediterà.

Nota: tutti i metodi dovrebbero essere verbi. Questo perché questo metodo si chiama **get_area()** e non **area()**.

```
class Shape:
    def __init__(self):
        pass

    def get_area(self):
        pass
```

Il codice sopra definisce la classe; tuttavia, non c'è ancora nulla di interessante in essa.

Implementiamo le funzionalità standard della maggior parte di queste forme.

```
class Shape:
    def __init__(self, side1, side2):
        self.side1 = side1
        self.side2 = side2

    def get_area(self):
        return self.side1 * self.side2

    def __str__(self):
        return f'The area of this {self.__class__.__name__} is: {self.get
```

Analizziamo cosa stiamo facendo con questo codice:

- Nel metodo __**init**__, richiediamo due parametri, **side1** e **side2**. Questi rimarranno come **attributi dell'istanza**.

- La funzione **get_area()** restituisce l'area della forma. In questo caso, sta usando la formula dell'area di un rettangolo poiché sarà più facile da implementare con altre forme.
- Il metodo **__str__()** è un "metodo magico" proprio come **__init__().** Consente di modificare il modo in cui un'istanza verrà pubblicata.
- L'attributo **self.__class__.__name__** nascosto si riferisce al nome della classe. Se state lavorando con una classe **Triangle**, questo attributo sarà "Triangle."

Classe Rectangle

Dal momento che abbiamo implementato la formula dell'area del rettangolo, potremmo creare una semplice classe per il rettangolo (che nel nostro esempio abbiamo lasciato in inglese, **Rectangle**) che non fa altro che ereditare dalla classe **Forma**.

Per applicare l'**ereditarietà** in Python, si crea una classe come al solito e si circonda la **superclasse** da cui si vuole ereditare con delle parentesi.

```
# Folded base class
class Shape: ...

class Rectangle(Shape): # Superclass in Parenthesis
    pass
```

Classe Square

Possiamo avere un eccellente approccio al **polimorfismo** con la classe per il quadrato, che nel nostro esempio abbiamo lasciato in inglese: **Square**.

Ricordate che un quadrato è solo un rettangolo i cui quattro lati sono tutti uguali. Questo significa che possiamo usare la stessa formula per ottenere l'area.

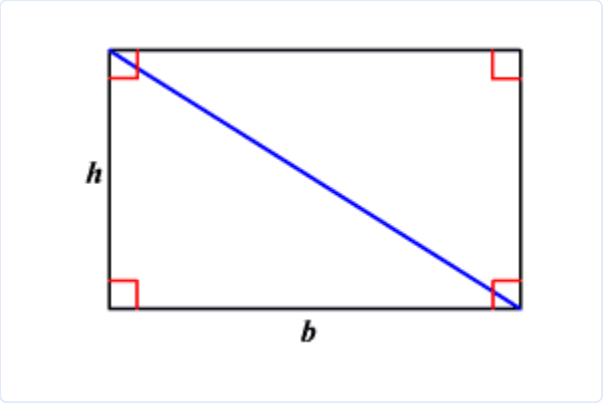
Possiamo farlo modificando il metodo **init**, accettando solo un lato come parametro, quindi **side** in inglese, e passando il valore di quel lato al costruttore della classe **Rectangle**.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...

class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)
```

Come potete vedere, la super funzione passa il parametro **side** due volte alla **superclasse**. In altre parole, sta passando **side** sia come **side1** che come **side2** al costruttore precedentemente definito.

Classe Triangle

Un triangolo è grande la metà del rettangolo che lo circonda.



— Relazione tra triangoli e rettangoli (Fonte immagine: Varsity tutors).

Possiamo quindi ereditare dalla classe **Rectangle** e modificare il metodo **get_area** per far corrispondere la formula dell'area del triangolo, che è la metà della base moltiplicata per l'altezza.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...

class Triangle(Rectangle):
    def __init__(self, base, height):
        super().__init__(base, height)

    def get_area(self):
        area = super().get_area()
        return area / 2
```

Un altro caso d'uso della funzione **super()** è chiamare un metodo definito nella **superclasse** e memorizzare il risultato come variabile. Questo è quello che succede con il metodo

get_area().

Classe Circle

Per trovare l'area del cerchio si usa la formula **?r²**, dove **r** è il raggio del cerchio. Questo significa che dobbiamo modificare il metodo **get_area()** per implementare questa formula.

Nota: possiamo importare il valore approssimativo di **?** dal modulo math:

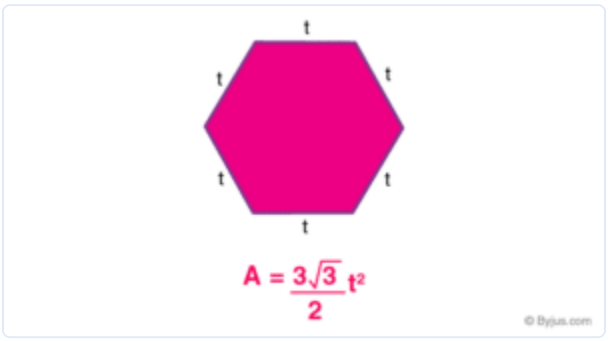
```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …

# At the start of the file
from math import pi

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return pi * (self.radius ** 2)
```

Il codice sopra definisce la classe **Circle**, che usa un diverso costruttore e metodi **get_area()**.

Sebbene **Circle** erediti dalla classe **Shape**, potete ridefinire ogni singolo metodo e attribuirlo a vostro piacimento.

Classe Regular Hexagon

Abbiamo solo bisogno della lunghezza di un lato di un esagono regolare per calcolare la sua area. È simile alla classe **Square**, dove passiamo solo un argomento al costruttore.



— Formula dell'area dell'esagono (Fonte immagine: BYJU'S)

Tuttavia, la formula è abbastanza diversa, e implica l'uso di una radice quadrata. Ecco perché userete la funzione **sqrt()** del modulo math.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …
class Circle(Shape): …

# Import square root
from math import sqrt

class Hexagon(Rectangle):
```

```
        def get_area(self):
            return (3 * sqrt(3) * self.side1 ** 2) / 2
```

Testare le Nostre Classi

È possibile entrare in una modalità interattiva quando si esegue un file Python usando un debugger. Il modo più semplice per farlo è usare la funzione integrata breakpoint.

Nota: questa funzione è disponibile solo in Python 3.7 o nelle versioni più recenti.

```
from math import pi, sqrt
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …
class Circle(Shape): …
class Hexagon(Rectangle): …

breakpoint()
```

Ora eseguite il file Python e giocate con le classi che avete creato.

```
$ python calculator.py

(Pdb) rec = Rectangle(1, 2)(Pdb) print(rec)
The area of this Rectangle is: 2
```

```
(Pdb) sqr = Square(4)
(Pdb) print(sqr)
The area of this Square is: 16
(Pdb) tri = Triangle(2, 3)
(Pdb) print(tri)
The area of this Triangle is: 3.0
(Pdb) cir = Circle(4)
(Pdb) print(cir)
The area of this Circle is: 50.26548245743669
(Pdb) hex = Hexagon(3)
(Pdb) print(hex)
The area of this Hexagon is: 23.382685902179844
```

Esercizio

Create una classe con un metodo **run** dove l'utente può scegliere una forma e calcolarne l'area.

Quando avete completato l'esercizio, potete inviare una richiesta di pull al repo GitHub o pubblicare la vostra soluzione nella sezione dei commenti.

Riepilogo

La programmazione orientata agli oggetti è un paradigma in cui risolviamo i problemi pensando a essi come a **oggetti**. Se capite la OOP di Python, potete anche applicarla facilmente in linguaggi come Java, PHP, Javascript, e C#.

In questo articolo, avete imparato a conoscere:

- Il concetto di orientamento agli oggetti in Python
- Svantaggi della programmazione orientata agli oggetti rispetto a quella strutturata
- Fondamenti della programmazione orientata agli oggetti in Python
- Concetto di **classi** e come usarle in Python
- Il **costruttore** di una classe in Python
- **Metodi** e **attributi** in Python

- I quattro pilastri della OOP
- Implementazione di **astrazione**, **ereditarietà** e **polimorfismo** in un progetto

Ora tocca a voi!

Se vi è piaciuta questa guida, date un'occhiata al nostro articolo sui tutorial di Python.

Fateci sapere la vostra soluzione dell'esercizio, scrivetela nei commenti! E non dimenticate di leggere la nostra guida al confronto tra Python e PHP.