ADSUM formazione d'eccellenza

Organismo accreditato MIM Ministero dell'Istruzione e del Merito

REGIONE PUGLIA

# MASTER IN *Intelligenza Artificiale*

EDIZIONE 1

2024-2025

## MODULO FORMATIVO N.1
## Fondamenti di Programmazione

## Programmazione Orientata agli Oggetti: Concetti base usando Python. + LABORATORIO

A cura del docente Marco Del Coco

# About me

## Marco Del Coco

Researcher at the **CNR - *ISASI***

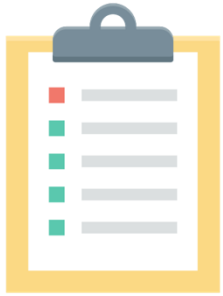
**National Research Council** (CNR)
*Institute of Applied Science and Intelligent Systems (ISASI)*

- *Ph.D in Information Engineering*
- *Master degree in Telecommunication Engineering*

*Email: marco.delcoco@cnr.it*

# Programme at a glance

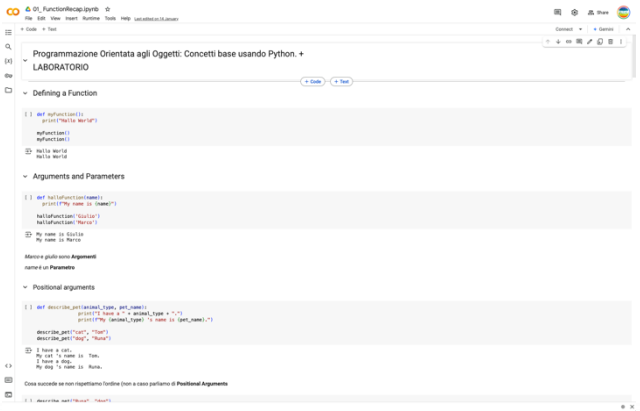
- Recap on Python **functions**
- OOP: Object Oriented Programming
- Hands on. Let's create a **class**
- Work with *Virtual Environment*
- Python advanced Concept

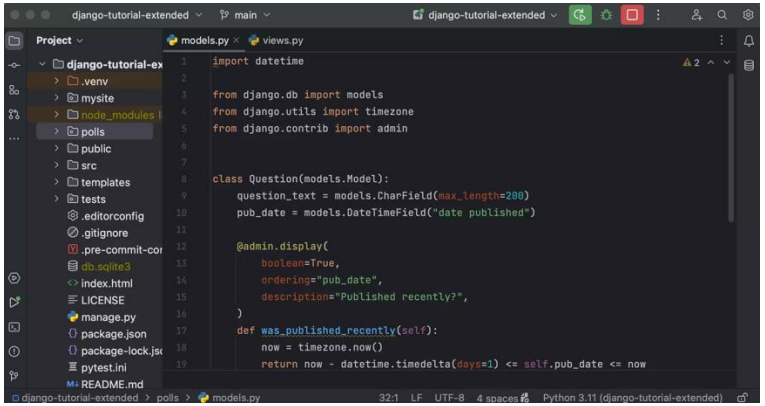


# Instruments

**Google Colab** is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs. Colab is especially well suited to machine learning, data science, and education.

**PyCharm** is an integrated development environment (IDE) used for programming in Python. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems



# Interesting resources & books

**On-line resources**

- **Python Documentation**: https://www.python.org/doc/
- **Geek4Geek**: https://www.geeksforgeeks.org/python3-tutorial
- **DataCamp**: https://www.datacamp.com/tutorial/category/python
- **w3School**: http://www.w3schools.com/python/
- **Medium**: https://martinxpn.medium.com/100-days-of-python-9dd04d0995f1


**Books**

- Python Crash Course: A Hands-On, Project-Based Introduction to Programming (3rd Edition)
- Beyond the Basic Stuff with Python: Best Practices for Writing Clean Code

**Python Notebook** Open in Colab

- https://github.com/beppe2hd/CoursePython

# Recap on **Python functions**

- Defining a Function
- Argument and Parameters
- Positional Arguments
- Keyword Arguments
- Default Values (equivalent function calls)
- Making an Argument Optional
- Argument type
- Return Values (including type output)
- Arbitrary arguments
- Arbitrary Keyword Arguments
- Variable Scope
- List Comprehension

Open in Colab Let's go → https://github.com/beppe2hd/CoursePython/blob/main/01__FunctionRecap.ipynb

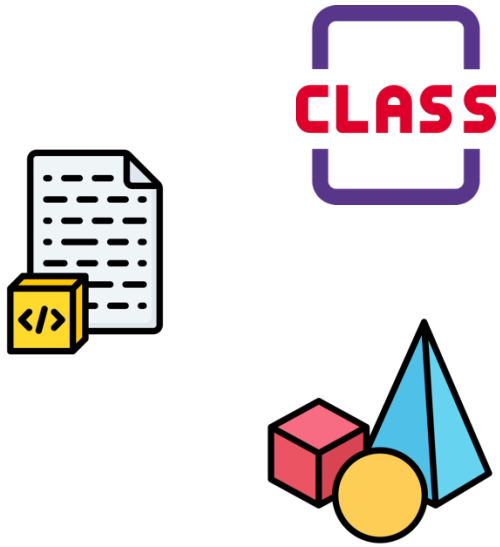
# OOP: Object Oriented Programming ----------

**Object-oriented programming (OOP**) o Programmazione orientate agli Oggetti é un *paradigma di programmazione* basato sul concetto di **oggetti** contenenti dati and codice.

- Dati in forma di campi (**attributes**)
- Codice in forma of procedure (**methods**).

Nell OOP, I programmi sono strutturati per essere composti da oggetti che interagiscono con altri oggetti.

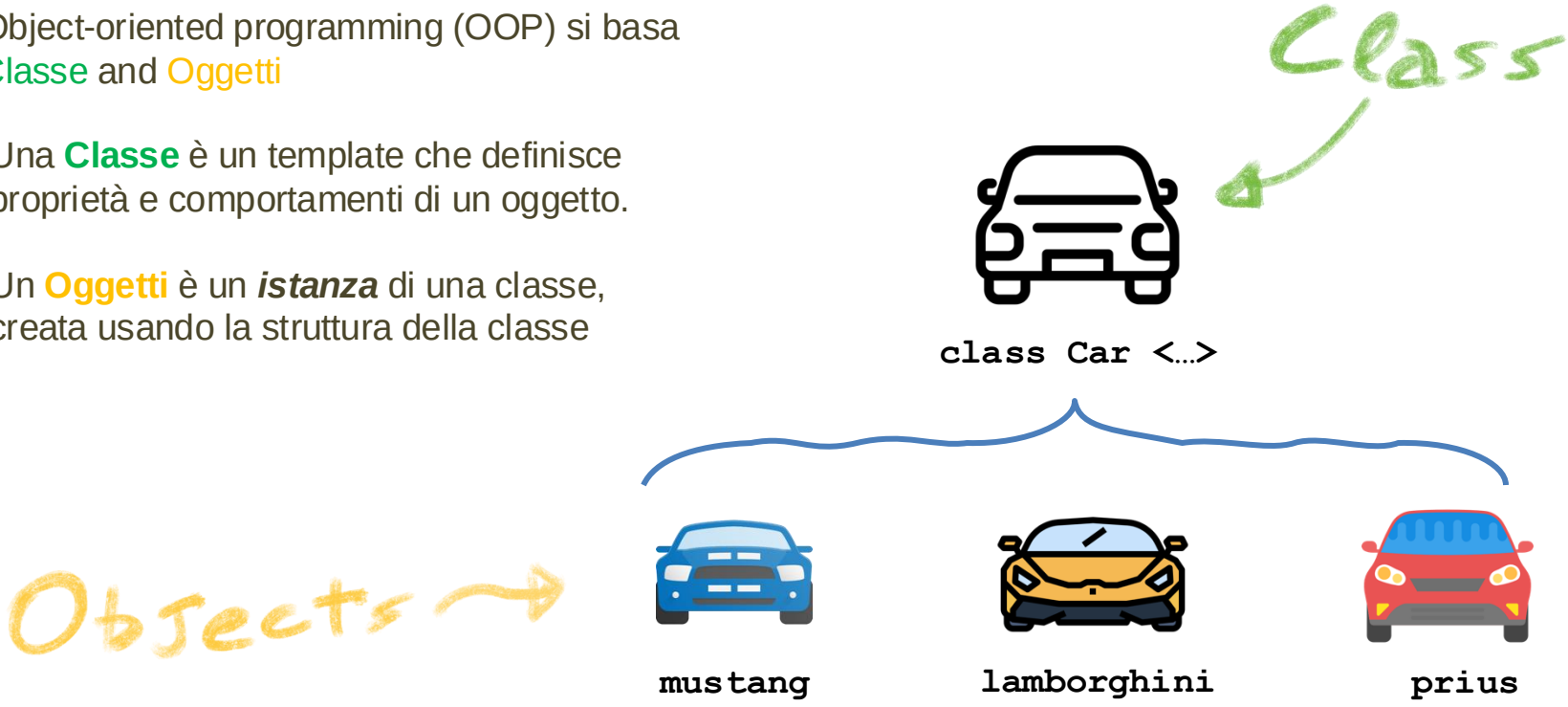
Python è un linguaggio di programmazione multi-paradigm che supporta la programmazione object-oriented programming tipicamente combinata con l'imperative programming.



# OOP: Classes and Objects ----------

La Object-oriented programming (OOP) si basa su Classe and Oggetti

- Una **Classe** è un template che definisce proprietà e comportamenti di un oggetto.
- Un **Oggetti** è un ***istanza*** di una classe, creata usando la struttura della classe



# OOP: Class Example ----------


Le **Classi** sono fatte principalmente da ***Attribbuti*** e ***Metodi.***

*Per definire una classe utilizziamo la parola chiave **class** seguita dal nome della classe*

Gli **attribbuti** di una classe sono le variabili e all'interno della classe sono preceduti dal prefisso ***self.***

I **metodi** sono le funzioni che fanno parte di una classe, anche loro all'interno della classe sono preceduti dal prefisso ***self.***

Il metodo __init__() è un metodo speciale che Python esegue automaticamente ogni volta che si crea una nuova istanza basata sulla classe Dog.

__init__() non ha mai un **return**

```
class Dog():
    """A simple attempt to model a dog."""
    def __init__(self, name, age):
        """Initialize name and age attributes."""
        self.name = name
        self.age = age


    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(self.name.title() + " is now sitting.")

    def roll_over(self):
        """Simulate rolling over in response to a command."""
        print(self.name.title() + " rolled over!")
```

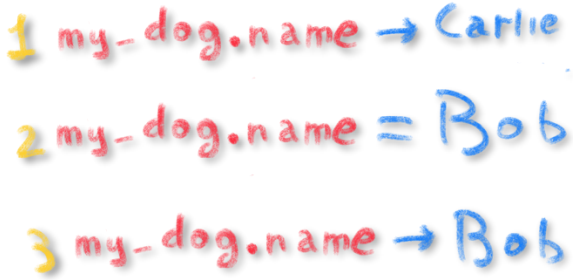
# OOP: Attributi ----------

Gli **attributi** sono variabili associate a un oggetto

La documentazione Python descrive gli attributi come "qualsiasi nome seguito da un punto"

Ogni oggetto ha il proprio insieme di attributi.


È possibile accedere e impostare questi attributi come qualsiasi altra variabile.



# OOP: Class Example ----------


Una volta definite una classe è possibile crearne una o più istanze

Una volta creata un'istanza è possibile utilizzare il **.** per

- utilizzarne i metodi
- accedere agli attributi

```
my_dog = Dog("Charlie", 7)
your_dog = Dog("Freddy", 3)

my_dog.sit()
your_dog.sit()

print(my_dog.name)
```

```
Charlie is now sitting.
Freddy is now sitting.
Charlie
```

# Function type() and __qualname__ ----------


La funzione type() ci dice il tipo di dato dell'oggetto

I termini ***type***, ***data type*** e ***class*** hanno lo stesso significato in Python.

```
print(type(4))
```

```
<class 'int'>
```

L'attributo __qualname__ permette di ottenere un risultato più leggibile

```
print(type(4).__qualname__)
```

```
int
```

# Exercise ----------


## Creiamo una classe Correntista

La classe dovrà contenere:

Attributi:
- Nome
- Cognome
- Saldo_disponibile
- Saldo_investito

Metodi:
- preleva(somma_prelevata)
- versa(somma_versata)
- investi(somma_da_investire)

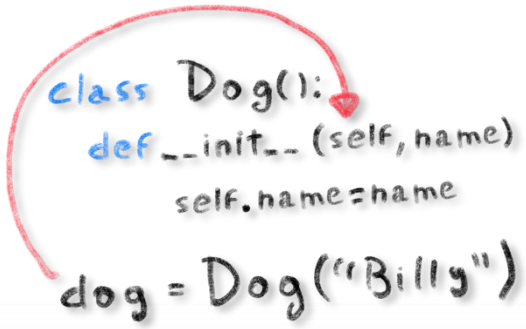
# OOP: Metodi, *__init__()*, and *self* ----------


I **metodi** sono funzioni associate agli oggetti di una particolare classe.

Gli **oggetti** (istanze di una classe) sono creati chiamando il nome della classe come fosse una funzione. Questa funzione viene chiamata costruttore.

La chiamata del costruttore fa sì che Python crei il nuovo oggetto e poi il metodo *__init__()* con cui si *impostano i valori iniziali degli attributi*.

Tutti i metodi hanno un primo parametro chiamato **self**. Quando un metodo viene richiamato su un oggetto, l'oggetto viene automaticamente passato al parametro self.



# Exercise


## Creiamo una classe MeteoHistory

La classe dovrà contenere:

Attributi:
- temp
- humidity
- rain
- wind

Metodi:
- calcola_media_temp()
- calcola_media_hum()
- calcola_media_ rain()
- calcola_media_ wind()

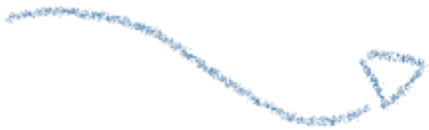
```
DateTime,temperature_2m,relative_humidity_2m,precipitation,wind_speed_10m
2007-04-20 00:00:00,9.516999,38.3732,0.0,7.628263
2007-04-20 01:00:00,9.066999,37.102863,0.0,7.5170207
2007-04-20 02:00:00,8.017,43.268085,0.0,4.2136917
2007-04-20 03:00:00,6.567,47.95537,0.0,7.0911775
2007-04-20 04:00:00,4.6670003,63.21383,0.0,4.5536795
2007-04-20 05:00:00,5.367,47.607956,0.0,1.1384199
2007-04-20 06:00:00,5.217,46.496143,0.0,2.5455842
2007-04-20 07:00:00,1.117,68.520515,0.0,5.4477882
2007-04-20 08:00:00,-0.133,74.71392,0.0,4.73506
2007-04-20 09:00:00,-0.73300004,77.75646,0.0,5.4477882
2007-04-20 10:00:00,-1.133,78.86919,0.0,5.771239
2007-04-20 11:00:00,-1.433,80.01844,0.0,6.4799995
2007-04-20 12:00:00,-1.683,80.5863,0.0,6.130579
2007-04-20 13:00:00,-1.783,81.18132,0.0,5.4119864
```

# OOP: WizCoin Example ----------


Immaginiamo di voler creare una classe per gestire la valuta di un mondo fantastico.

Questa valuta ha 3 tipi di monete:

- Galleon
- Sickle
- Knut



- Galleon = 17 Sickle
- Sickle = 29 Knut.
- Gallons
- Sickle = 11.34 g
- Knut = 5.0 g

Vogliamo una classe che possa gestire il «portafoglio» di un abitante del nostro mondo fantastico fornendo il **valore in Knut** e il **valore in grammi di oro** del suo portafoglio

# OOP: WizCoin Example ----------


Creiamo la classe **WizCoin**

```python
class WizCoin:
  def __init__(self, galleons, sickles, knuts):
    """Create a new WizCoin object with galleons, sickles, and knuts."""
    self.galleons = galleons
    self.sickles = sickles
    self.knuts = knuts

  def value(self):
    """The value (in knuts) of all the coins in this WizCoin object."""
    return (self.galleons * 17 * 29) + (self.sickles * 29) + (self.knuts)

  def weightInGrams(self):
    """Returns the weight of the coins in grams."""
    return (self.galleons * 31.103) + (self.sickles * 11.34) + (self.knuts* 5.0)
```

# OOP: WizCoin Example ----------

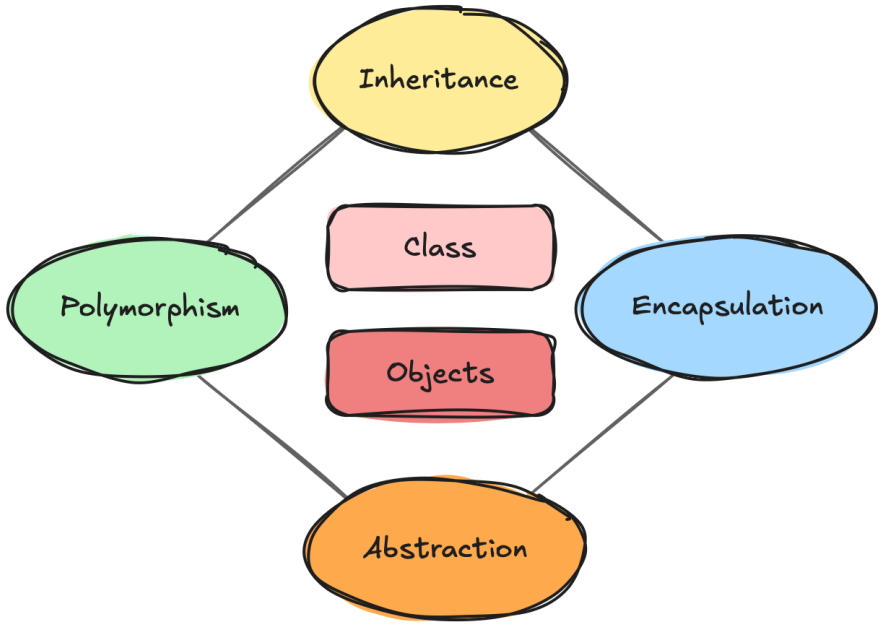

Creiamo la classe **WizCoin**

```python
class WizCoin:
  def __init__(self, galleons, sickles, knuts):
    """Create a new WizCoin object with galleons, sickles, and knuts."""
    self.galleons = galleons
    self.sickles = sickles
    self.knuts = knuts

  def value(self):
    """The value (in knuts) of all the coins in this WizCoin object."""
    return (self.galleons * 17 * 29) + (self.sickles * 29) + (self.knuts)

  def weightInGrams(self):
    """Returns the weight of the coins in grams."""
    return (self.galleons * 31.103) + (self.sickles * 11.34) + (self.knuts* 5.0)
```


# The OOP pillars: *overview* ----------

La *Object-oriented programming* si basa su 4 pilastri fondamentali

- Encapsulation: si riferisce all'idea di nascondere i dettagli implementativi di un oggetto
- Inheritance: è un meccanismo che consente a una classe di ereditare proprietà e metodi da un'altra classe, detta superclasse o classe madre.
- Polymorphism è la capacità di un oggetto di assumere più forme.
- Abstraction: è il concetto di mostrare all'esterno solo le informazioni necessarie, nascondendo i dettagli non necessari.



# The OOP pillars: *Encapsulation* ----------


Encapsulation consists of **hiding** the implementation details of an object from the outside world and only exposing the necessary.

By using **private** and **protected** fields, developers can control the visibility and accessibility of class ***attributes*** and ***methods***, preventing accidental or intentional misuse.

Python supports encapsulation through **conventions** and **programming practices** rather than enforced access modifiers (like C, C++).

Python relies on the "We are all adults here" idea, so you can only implement encapsulation as a mere convention.

# Non Pubblic *Attributes* and *Methods* ----------


In Python tutti gli attributi e i metodi sono sono di fatto ad **accesso pubblico**

Il codice esterno alla classe può accedere ad attributi e metodi di un oggetto in qualsiasi momento.


Esiste pero una *convenzione* per «definire» un metodo o un attributo come «non pubblico»: porre un _ (underscore) prima del nome.

```
self._my_attribute
```

```
def _my_function():
    # some code
```

# Name Mangling ----------

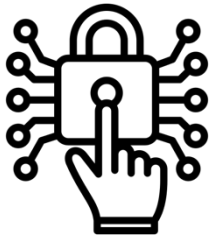
Per i casi più delicati in cui è opportuno avere un controllo più «stretto»
In questi casi esiste un modo per preservare un metodo o un attributo ponendo **due** __ (underscore) prima del nome.


```
class Something:
    self.__my_attribute
```

Questo sistema è noto come Name Mangling e trasforma il nome con cui riferirsi al metodo o all'attributo ponendo davanti ad esso il nome della classe


# *Someone say **Protected** and **Private** ----------*

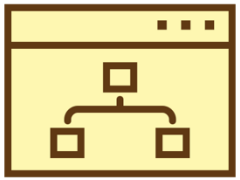

- **Public Access Modifier**: i metodi e i campi pubblici sono accessibili direttamente da qualsiasi classe.
- **Protected Access Modifier** ( _ ): i metodi e i campi protetti **dovrebbero** essere accessibili all'interno della stessa classe dichiarata e della sua sottoclasse.
- **Private Access Modifier** ( __ ): i metodi e i campi privati **dovrebbero** essere accessibili solo all'interno della stessa classe dichiarata. → name mangling



# The OOP pillars: *Inheritance* ----------


Inheritance è una tecnica di riutilizzo del codice che si può applicare alle classi.

È l'atto di mettere le classi in una relazione genitore-figlio in cui la classe figlia eredita una copia dei metodi della classe genitrice, evitando di duplicare un metodo in più classi.



L'ereditarietà è a volte sopravvalutata o addirittura pericolosa, a causa della complessità aggiuntiva che grandi reti di classi ereditate aggiungono a un programma. -→ ***Inheritance is evil***

Tuttavia, un *utilizzo consapevole* può essere fondamentale nell'ottenere un codice:
- Ordinato
- Manutenibile
- Riusabile


*Utilizzo consapevole* → ***Esperienza*** → *Sbaglia, Impreca, Correggi*

# The OOP pillars: *Inheritance* ----------

Proviamo a scrivere una scrivere una classe padre e delle classi figlie

```
class ParentClass:
  def printHello(self):
    print('Hello, world!')

class ChildClass(ParentClass):
  def someNewMethod(self):
    print('ParentClass objects don't have this method.')

class GrandchildClass(ChildClass):
  def anotherNewMethod(self):
    print('Only GrandchildClass objects have this method.')
```

# Overriding Methods (*Inheritance*) ----------


Le classi figlio ereditano tutti i metodi delle classi genitore.

Una classe figlio può sovrascrivere un metodo ereditato (**Override**) fornendo il proprio metodo con il proprio codice.

Il metodo sovrascritto della classe figlio avrà lo stesso nome del metodo della classe genitore.

```
class ParentClass:
  def printHello(self):
    print('Hello, world!')

  def printGoodNight(self):
    print('Good Noght')

class ChildClass(ParentClass):
  def someNewMethod(self):
    print("ParentClass objects don't have this method.")

  def printGoodNight(self): #override method
    print('Good Noght Bro')
```

# The *super()* function ----------


Il metodo sovrascritto (**overridden method**) di una classe figlio è spesso simile al metodo della classe genitore.

Anche se l'ereditarietà è una tecnica di riutilizzo del codice, la sovrascrittura di un metodo potrebbe causare la riscrittura dello stesso codice del metodo della classe genitore come parte del metodo della classe figlio.

Per evitare questa duplicazione di codice, la funzione built-in **super()** consente a un metodo sovrascritto di richiamare il metodo originale della classe genitore.

***super().__init__()*** si assicura che il costruttore della classe genitore sia chiamato, inizializzando tutti gli attributi definiti nella classe genitore. È buona norma chiamare il costruttore della classe genitore, se si tratta di un'inizializzazione importante.

# Favor Composition Over Inheritance ----------


Ma non sempre si vuole che la classe genitore e le sottoclassi siano così strettamente accoppiate

La classe figlio è un tipo della classe genitore → Child "is a" Parent → Ereditarietà

Una classe include un'altra classe → Class "has a" Class → Composizione

```
class WizardCustomer(WizCoin):
  def __init__(self, name):
    self.name = name
    super().__init__(0, 0, 0)


wizard = WizardCustomer('Alice')
print(f'{wizard.name} has {wizard.value()} knuts worth of money.')
print(f'{wizard.name}\'s coins weigh {wizard.weightInGrams()} grams.')
```

# Plymorphism ----------


Il polimorfismo consente agli oggetti di un tipo di essere trattati come oggetti di un altro tipo.

Ad esempio, la funzione **len()** restituisce la lunghezza dell'argomento ad essa passato.

Si può passare una stringa a len() per vedere quanti caratteri ha

Ma si può anche passare una lista o un dizionario a len() per vedere quanti elementi o coppie chiave-valore ci sono.

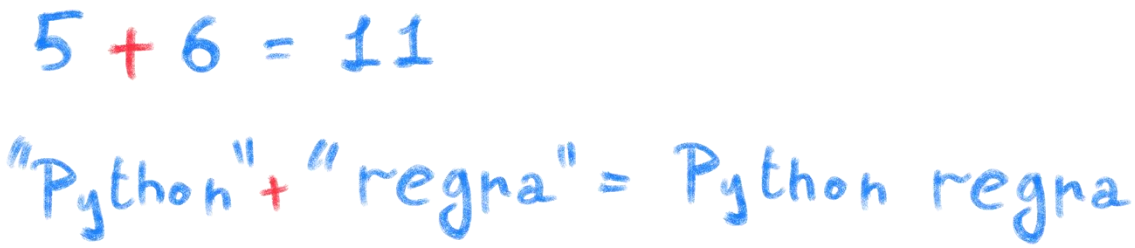
Questa forma di polimorfismo è chiamata ***generic functions*** *o* ***parametric polymorphism***, perché è in grado di gestire oggetti di molti tipi diversi.

# Polymorphism ----------

Il polimorfismo si riferisce anche al **polimorfismo ad hoc** o all'**overload degli operatori**.

Gli operatori (come + o *) possono avere un comportamento diverso in base al tipo di oggetti su cui operano.

Ad esempio, l'operatore + esegue una somma quando opera su due valori interi o float, ma concatenazione di stringhe quando opera su due stringhe.




In questo caso si parla anche di **Numeric Dunder Methods**

# Dunder Methods ----------


Python ha diversi nomi di metodi speciali che iniziano e finiscono con doppi trattini, abbreviati in **dunder**.

Uno di questi che abbiamo già visto è **__init__**

**Altri abbastanza famosi e utili sono:**

__str__()

__len__()

__add__()

__mul__()

```
def __add__(self, other):
  """Adds the coin amounts in two WizCoin objects together."""
  if not isinstance(other, WizCoin):
    return NotImplemented

  return WizCoin(other.galleons + self.galleons, other.sickles +
                 self.sickles, other.knuts + self.knuts)
```

# Class Attributes and Class Methods ----------


I **Class Methods** e I **Class Attributes** sono una cosa diversa dai Metodi di una classe e dagli attributi di una classe.

I **Class Methods** sono associati con la classe e non al singolo oggetto.

Sono facilmente riconoscibili dall'uso

- Del decoratore ***@classmethod***
- Della parola chiave ***cls come primo parametro***

I **Class Attributes** sono variabili che appartengono alla classe invece che ad un oggetto

```
class ExampleClass:

  example_class_attribute = 'This is a class attribute.'

  def exampleRegularMethod(self):
    print('This is a regular method.')

  @classmethod
  def exampleClassMethod(cls):
    print('This is a class method.')

ExampleClass.exampleClassMethod()
print(ExampleClass.example_class_attribute)
```

```
class CreateCounter:

  count = 0 # This is a class attribute.

  def __init__(self):
    CreateCounter.count += 1


print('Objects created:', CreateCounter.count)
a = CreateCounter()
b = CreateCounter()
c = CreateCounter()
print('Objects created:', CreateCounter.count)
```

# Static Methods ----------


Gli **Static Method** sono metodi di una classe che:

- Sono identificati con il *decoratore* **@staticmethod**
- Non hanno un parametro **self**
- Non hanno un parametro **cls**
- Non possono accedere ad attributi e metodi della classe

```
class ExampleClassWithStaticMethod:

  @staticmethod
  def sayHello():
    print('Hello!')

# Note that no object is created, the class name precedes sayHello():
ExampleClassWithStaticMethod.sayHello()
```

**Sono una tipologia di metodo che è preferibile evitare.**