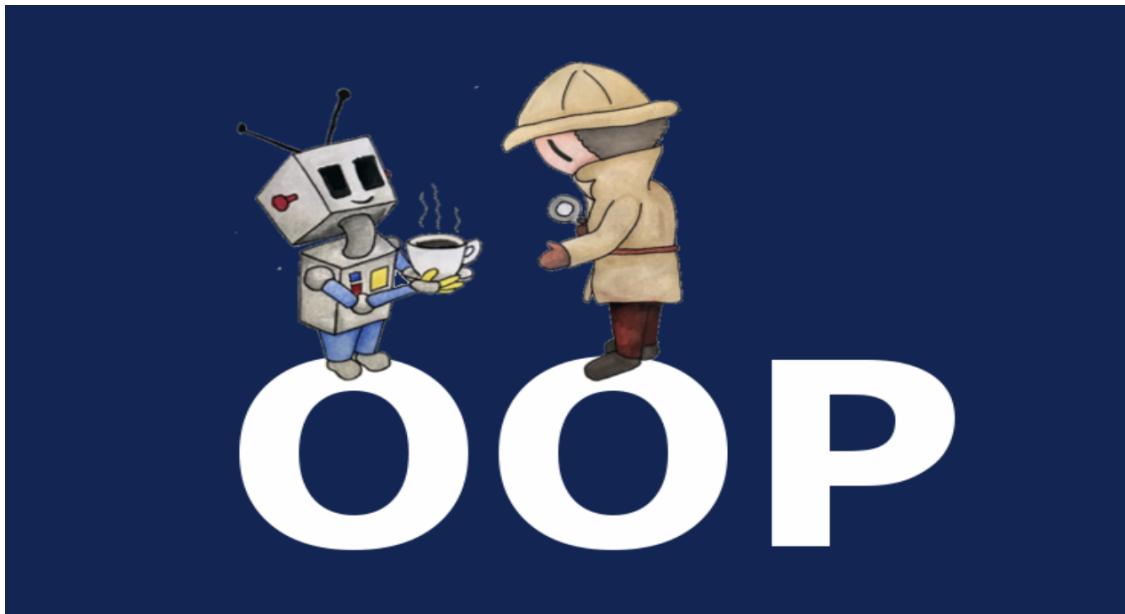


Object Oriented Programming

Appunti di Luciano Imbimbo

August 2022



Contents

1 Object Oriented Programming	7
1.1 Incapsulamento	7
1.2 Ereditarietà	7
1.3 Polimorfismo	7
1.4 UML	11
2 Java Programming	14
2.1 Basics	16
2.2 Strings	19
2.3 Array	20
2.4 Random Numbers	22
2.5 Classes	23
2.5.1 Definizione di classe	23
2.5.2 Override	26
2.5.3 Upcast e Downcast	26
2.5.4 Metodi Abstract e Interfacce	29
2.5.5 Approfondimenti	30
2.6 Java Collections	31
2.6.1 List Interface	33
2.6.2 Set Interface	33
2.6.3 Queue Interface	34
2.6.4 Map Interface	34
2.6.5 Iteratori	35
2.6.6 Ricerca ed Ordinamento	35
2.7 Generics	37
2.8 Exception	39
2.9 Approfondimento Composition	41
2.10 Swing Frameworks	42
2.10.1 GamePanel	45
2.11 JDBC	46
2.11.1 ResultSet	48
2.11.2 Transactions	49
2.12 Functional Interfaces	50
2.13 REST	53
2.13.1 Object Mapper	57
2.14 Multi-Threading	58
2.14.1 Thread Synchronization	61
3 Python Programming	67
3.1 The Python Environment	67
3.1.1 Timeline	67
3.1.2 Advantages and disadvantages	67
3.1.3 The Zen of Python	68
3.1.4 Python Enhancement Proposals	69
3.1.5 Python Virtual Machine	69
3.1.6 Using Python	69
3.1.7 Libraries	70
3.1.8 Which Python version am I running?	71
3.2 Basic concepts	71
3.2.1 Main function	71
3.2.2 Multi-line statements	71
3.2.3 Indentation	71

3.2.4	Naming rules	72
3.2.5	Variable Assignment	72
3.2.6	Constants	72
3.2.7	Everything Is an Object	73
3.3	Literals	74
3.3.1	Numeric literals	74
3.3.2	Boolean literals	74
3.3.3	String literals	76
3.3.4	Implicit Casting	76
3.3.5	Explicit Casting	77
3.4	Sequences	78
3.4.1	Operations (for Sequences)	78
3.4.2	Functions (for Sequences)	79
3.5	Strings	79
3.5.1	Strings	79
3.5.2	Accessing characters	80
3.5.3	Combining Strings	80
3.5.4	Formatting Strings	81
3.5.5	Dealing with whitespaces	81
3.5.6	Dealing with cases	82
3.5.7	Finding and replacing substrings	82
3.5.8	Splitting and joining strings	82
3.6	Flow control	83
3.6.1	if .. else	83
3.6.2	if .. elif .. else	83
3.6.3	Comparison operators	84
3.6.4	Logical operators	85
3.6.5	for loop	86
3.6.6	while loop	87
3.6.7	pass	87
3.7	Functions	88
3.7.1	General Syntax	88
3.7.2	Passing parameters	88
3.7.3	Default Arguments	89
3.7.4	Keyword Arguments	89
3.7.5	Arbitrary Arguments	90
3.7.6	Arbitrary keyword arguments	90
3.7.7	Lambda expressions	91
3.7.8	Returning multiple values	92
3.8	Decorators	94
3.8.1	Inner Functions	94
3.8.2	Decorators as wrappers	95
3.8.3	Pie (@) syntax	95
3.8.4	Passing arguments	95
3.8.5	Returning arguments	96
3.9	Exceptions	97
3.9.1	Definition	97
3.9.2	try - except	97
3.9.3	Delegation	98
3.9.4	Raising exceptions	98
3.9.5	try - except - else	99
3.9.6	try - except - finally	99
3.9.7	Summarizing	100
3.10	File I/O	101

3.10.1	Working with paths	101
3.10.2	Checking paths	101
3.10.3	Reading files	101
3.10.4	Writing files	102
3.11	Comments	102
3.11.1	Docstring	102
3.11.2	What makes a good comment?	103
3.11.3	When should you write comments?	103
3.12	Data Structures	103
3.12.1	General Concepts	103
3.12.2	Containers	103
3.12.3	Iterables	104
3.12.4	Iterables and for loops	105
3.12.5	Iterators	106
3.12.6	Functions acting on iterables	106
3.13	List	107
3.13.1	Accessing elements	107
3.13.2	Changing elements	109
3.13.3	Adding elements	109
3.13.4	Removing elements	109
3.13.5	Sorting	110
3.13.6	Other List operations	111
3.14	Tuple	112
3.14.1	Accessing elements	113
3.14.2	Changing elements	113
3.14.3	Other Tuple operations	114
3.14.4	Advantages of Tuple over List	115
3.15	Set	115
3.15.1	Accessing elements	116
3.15.2	Adding elements	116
3.15.3	Removing elements	117
3.15.4	Other Set operations	117
3.15.5	Frozenset	119
3.16	Dictionary	120
3.16.1	Accessing elements	120
3.16.2	Changing elements	121
3.16.3	Adding elements	121
3.16.4	Removing elements	121
3.16.5	Other Dictionary Operations	122
3.16.6	OrderedDict	123
3.17	Comprehension Expressions	123
3.17.1	Unpacking	123
3.17.2	Enumerating	124
3.17.3	Generators	125
3.17.4	Differences between generators and normal function	125
3.17.5	Generator Comprehension	127
3.17.6	Consuming generators	128
3.17.7	List & Tuple Comprehensions	129
3.17.8	Dictionary Comprehension	130
3.18	Itertools	130
3.18.1	range()	131
3.18.2	enumerate()	131
3.18.3	zip()	131
3.19	Sorting Iterables	132

3.19.1	Using sort() and sorted()	132
3.19.2	The operator module	132
3.19.3	Sorting on multiple levels	133
3.20	Copying Iterables	133
3.20.1	Shallow copy	133
3.20.2	Deep copy	136
3.21	Classes	138
3.21.1	General concepts	138
3.21.2	Object references	138
3.21.3	Object identity, type, internal state	139
3.21.4	Class definition	141
3.21.5	Constructor method	142
3.21.6	Garbage collector	142
3.21.7	Destructor method	143
3.21.8	Class attributes	143
3.21.9	@classmethod	144
3.21.10	@staticmethod	144
3.21.11	String representations	145
3.21.12	self.__dict__	146
3.21.13	One class, many objects	146
3.21.14	Docstrings	147
3.22	Encapsulation, Inheritance, Polymorphism	147
3.22.1	Encapsulation	147
3.22.2	Inheritance	150
3.22.3	Multilevel Inheritance	151
3.22.4	Multiple Inheritance	151
3.22.5	Method Resolution Order	151
3.22.6	Polymorphism	152
3.23	Informal Interfaces	153
3.24	Sorting user-defined objects	154
3.24.1	Using sort() and sorted()	154
3.24.2	The operator module	154
3.25	Modules and classes	155
3.25.1	Modules	155
3.25.2	Importing classes	156
3.25.3	Importing functions	157
3.25.4	PEP8 guidelines	158
3.26	NumPy	158
3.26.1	What is NumPy?	158
3.26.2	Why is NumPy Fast?	159
3.26.3	Array initialization	159
3.26.4	Features	159
3.26.5	Constructor	159
3.26.6	Alternative constructors	160
3.26.7	Datatypes	161
3.27	Array attributes	161
3.27.1	Indexing and slicing	162
3.27.2	Integer indexing	162
3.27.3	Boolean indexing	162
3.27.4	Mutating elements with indexing	163
3.27.5	Slicing	164
3.28	Array math	165
3.28.1	Basic functions	165
3.28.2	Dot product	166

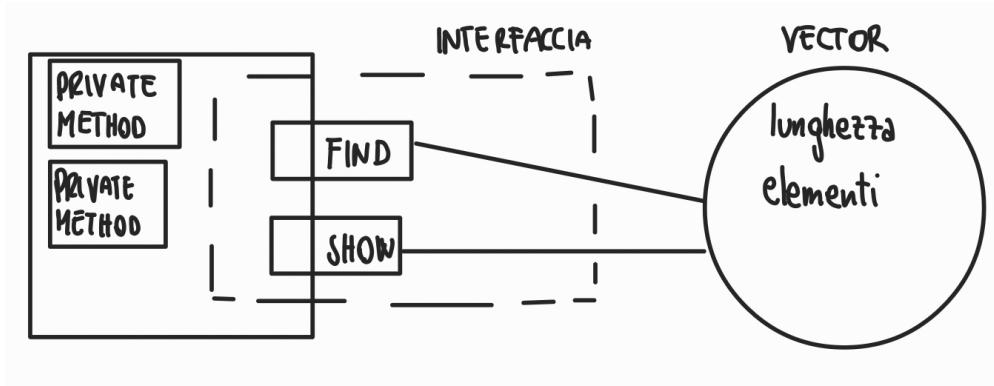
3.28.3 Computations on arrays	166
3.28.4 Reshaping	167

1 Object Oriented Programming

Object Oriented programming (OOP) is a programming paradigm. Traducendo e andando piú nel dettaglio, la programmazione ad oggetti é un modo di programmare standard dell'industria del software e principalmente basata su 3 concetti fondamentali.

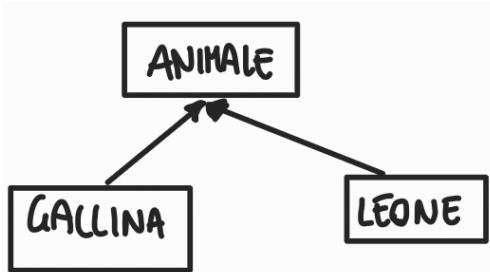
1.1 Incapsulamento

Il primo concetto fondamentale é l'incapsulamento, regola per cui i dati dell'oggetto sono encapsulati da uno scudo di codice. Le variabili interne, accessibili dall'esterno tramite metodi prestabiliti (Get and Set), sono dati che descrivono lo stato interno dell'oggetto.



1.2 Ereditarietà

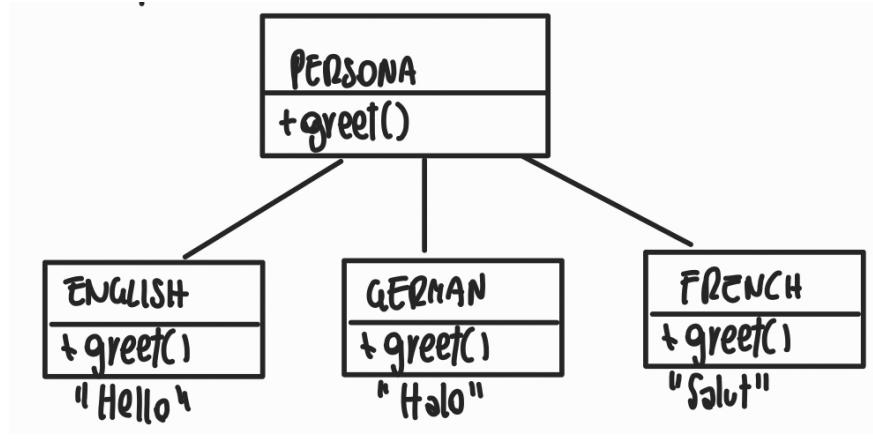
É possibile creare delle gerarchie (padre-figlio, nonno-nipote, ecc...) tra i vari oggetti. Considerando una relazione padre-figlio, l'oggetto figlio eredita tutte le caratteristiche (metodi e variabili) dell'oggetto padre. Questa principio ci permette di effettuare delle specializzazioni, cioè di sviluppare oggetti piú specifici da oggetti con caratteristiche generali senza creare del codice ridondante.



Gli oggetti Gallina e Leone sono delle specializzazioni dell'oggetto generale Animale. Entrambi gli oggetti ereditano caratteristiche comuni, tipiche di tutti gli animali, ma ogni figlio si specializza con delle proprie variabili e propri metodi.

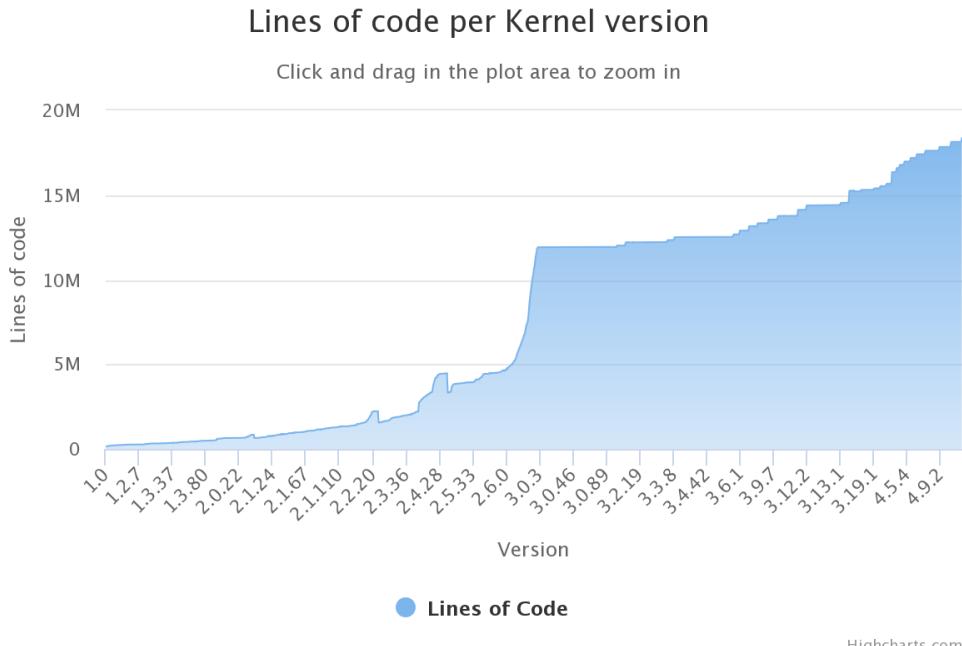
1.3 Polimorfismo

Con polimorfismo, intendiamo la possibilità che pezzi di codice, in particolare funzioni e metodi, riportino risultati diversi in contesti diversi. Solitamente è legato alle relazioni di eredità tra classi, che garantisce che tali oggetti, pur di tipo differente, abbiano una stessa interfaccia.



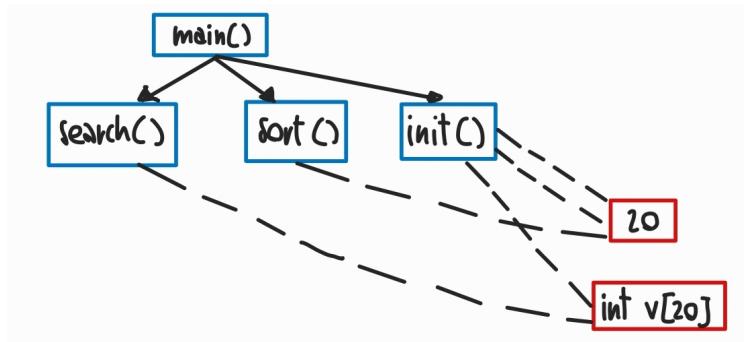
In verità tutti questi concetti non sono esclusivi della OOP ma potevano essere implementati anche nei classici linguaggi procedurali, tra i quali il C e il PASCAL. A seguito, però, della software crisis degli anni '70 dovuta all'elevata complessità del codice la programmazione ad oggetti ha assunto una notevole importanza. Infatti grazie all'uso del polimorfismo, il programmatore ha il totale controllo delle dipendenze tra i vari pezzi di codice del sistema (rompendo la dipendenza tra metodo chiamante e metodo chiamato). Quindi la programmazione ad oggetti portò:

- Maggiore facilità nello Sviluppo Cooperativo
- Maggiore Efficienza nel controllo di codice crescente; gli errori sono maggiormente localizzati nelle classi
- Minori costi di sviluppo e mantenimento

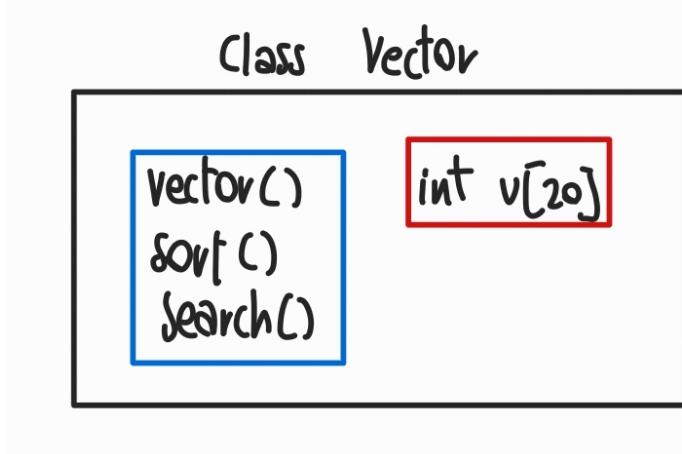


Come già detto, i concetti fondamentali dell'OOP possono essere implementati anche nei linguaggi procedurali grazie all'uso di puntatori a struct di funzioni. La differenza principale, però, è che nei linguaggi procedurali i dati sono separati dai metodi, mentre nella programmazione ad oggetti l'idea fondamentale è che si considera il software come un insieme ben definito di entità contenente i dati e le funzioni che lavorano su quei dati.

Consideriamo l'esempio di un vettore.



Il Vettore e la sua dimensione sono due entità separate. Inoltre il vettore non è un dato primitivo ma solo uno spazio in memoria. Non esiste una vera e propria protezione dei dati e si verifica il cosiddetto Issue dello Spaghetti Code, cioè c'è una difficoltà nella manutenzione per via dell'elevata dipendenza tra i vari moduli.

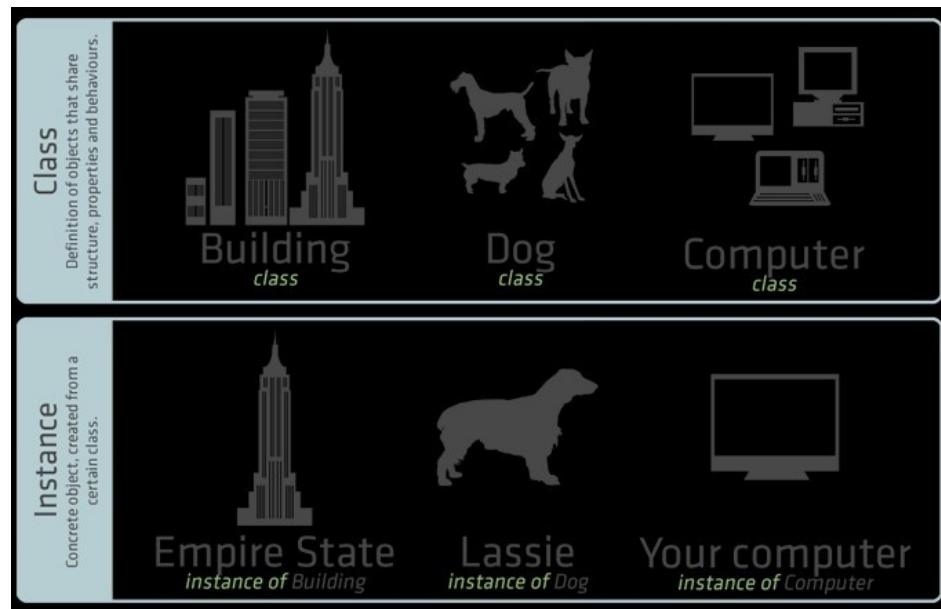


Il vettore, la sua dimensione e i suoi metodi fanno parte di un unico blocco chiamata Classe; questo permette una efficace protezione dei dati e ci si avvicina ad un modello Client-Server, in cui l'oggetto diventa un server che eroga servizi (metodi) e il programmatore diventa il cliente che riceve quei servizi.

C'è un ultimo punto che dobbiamo trattare, la differenza tra oggetto e classe. Fino ad ora, abbiamo utilizzato i due termini indistintamente ma esiste una sostanziale differenza:

- La CLASSE è l'entità generale che descrive metodi e attributi, cioè definisce dati, funzioni e costruttori (simile alla definizione di un tipo)

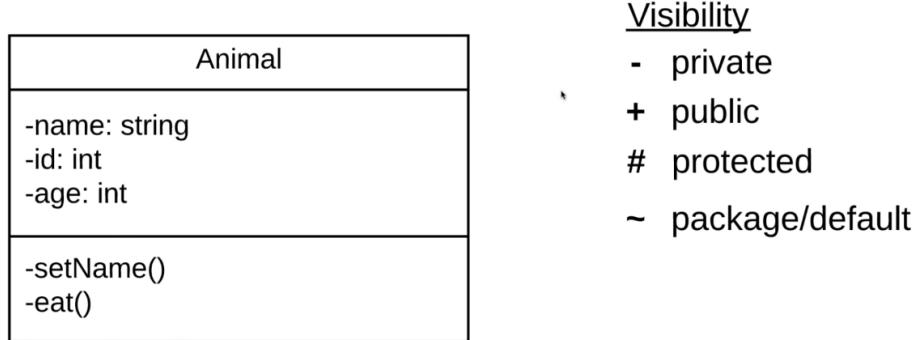
- L'OGGETTO di una classe è l'istanza della classe, cioè è un implementazione specifica di quella classe. In genere l'oggetto descrive l'identità e lo stato.



1.4 UML

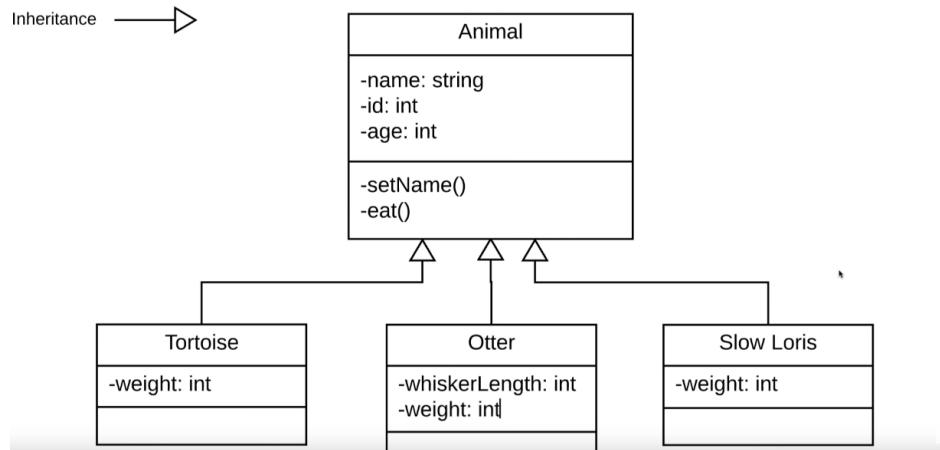
Per sviluppare applicativi di buone dimensione(piú di 8 classi) con linguaggi ad oggetti abbiamo bisogno di una chiara e pianificata progettazione dell'organizzazione delle classi. Per sviluppare graficamente un progetto ci viene in aiuto l'UML o Unified Modeling Language, che ci permette di pianificare l'organizzazione delle classi del progetto. Una volta sviluppato graficamente l'UML chiunque potrà scriverlo e trasformarlo in codice di un qualsiasi linguaggio ad oggetti.

All'interno dell'UML potremmo graficare sia le classi, con metodi e attributi, sia le relazioni tra le varie classi di un progetto. Per definire classi astratte e interfacce basta mettere il nome della classe tra parentesi angolari «».

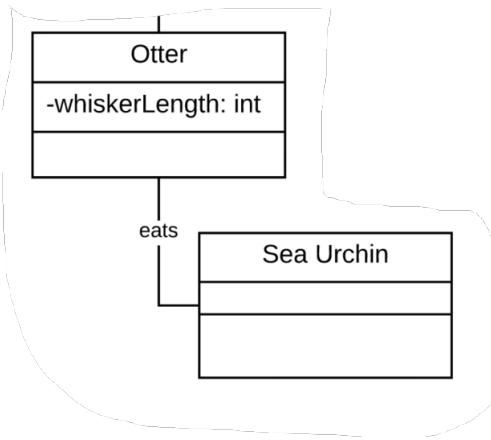


Tra le varie classi possono esserci diversi tipi di relazione.

- Inheritance



- Association



- Aggregation in Composition

Multiplicity

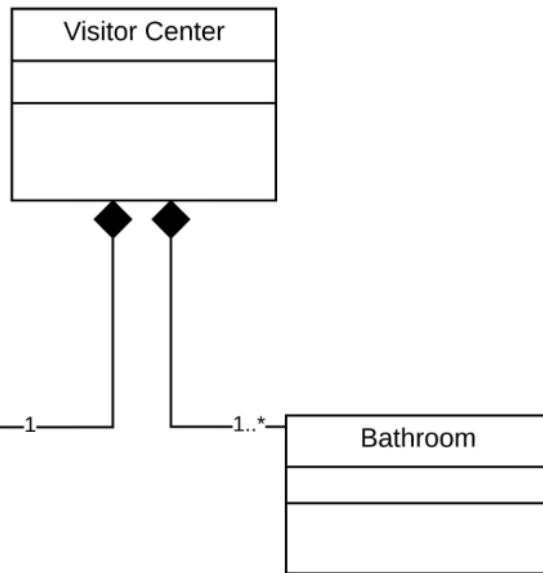
0..1 zero to one (optional)

n specific number

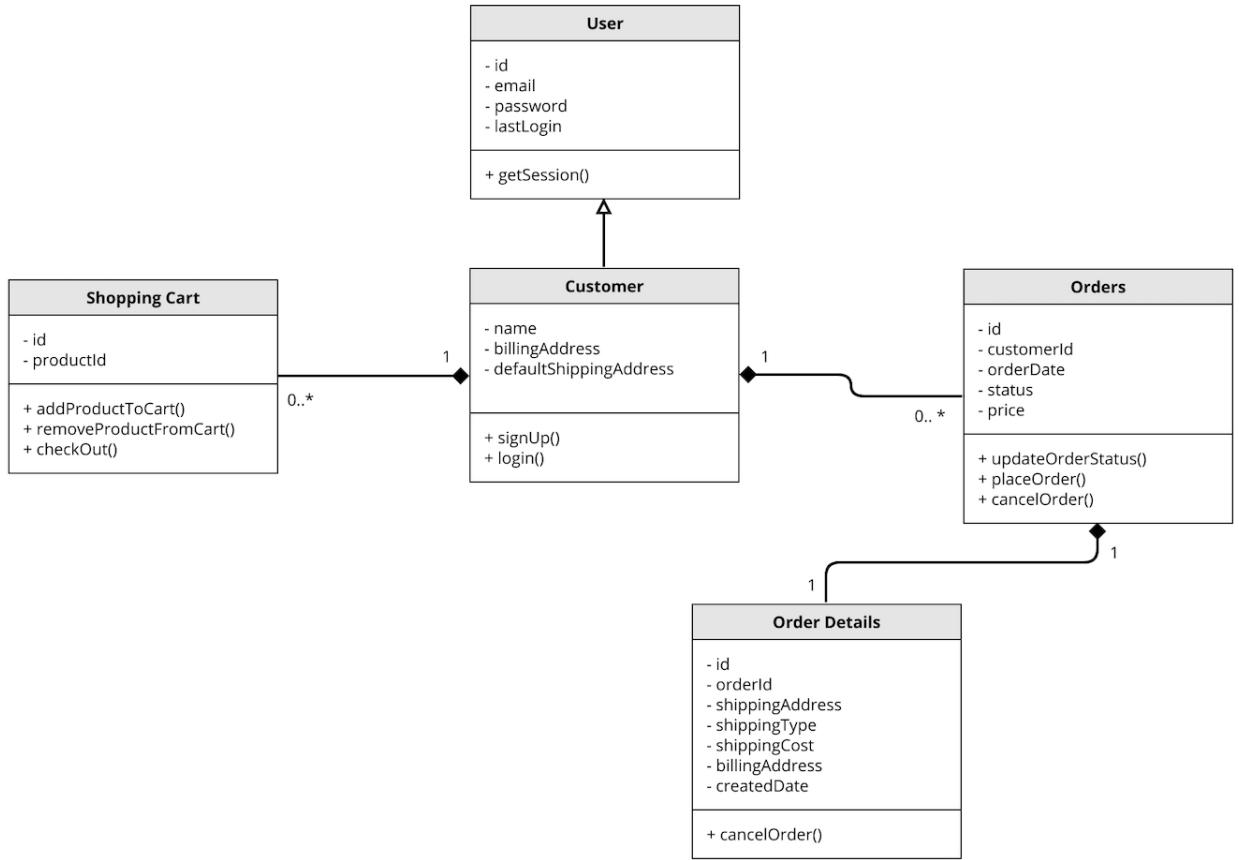
0..* zero to many

1..* one to many

m..n specific number range



Quindi un diagramma UML completo si presenta nel seguente modo:

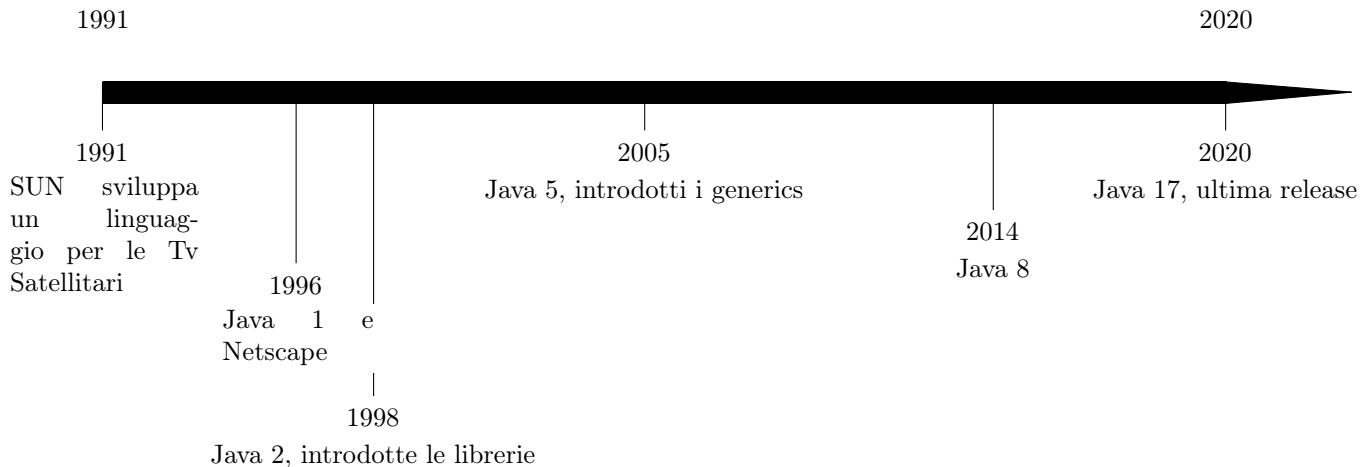


2 Java Programming

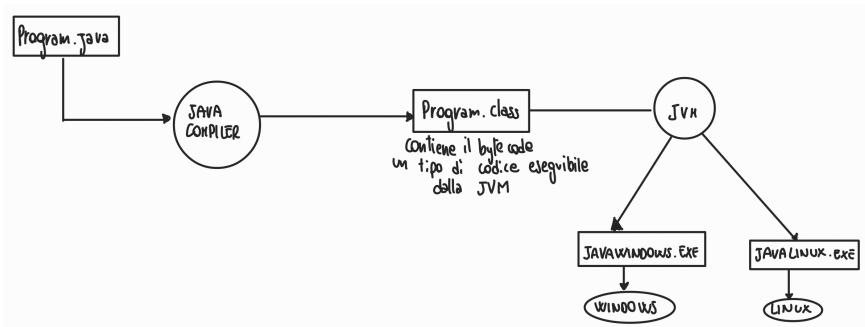
Uno dei piú famosi linguaggi di programmazione usati nel paradigma dell'OOP é JAVA. I motivi principali per cui JAVA ha raggiunto un successo tale sono:

- Indipendenza dalla piattaforma, cioè é indipendente dall'OS e dal hardware su cui viene eseguito. Infatti il codice JAVA non viene eseguito dalla macchina fisica ma bensí dalla Java Virtual Machine (JVM).
- Pure Object Oriented Language.
- Strettamente tipizzato, cioè il programmatore è tenuto a specificare il tipo di ogni elemento sintattico che durante l'esecuzione denota un valore.
- Incorpora una garbage collection automatica, quindi il sistema si occupa di rimuovere lo spazio occupato.
- Gestione degli errori attraverso l'uso di eccezioni.

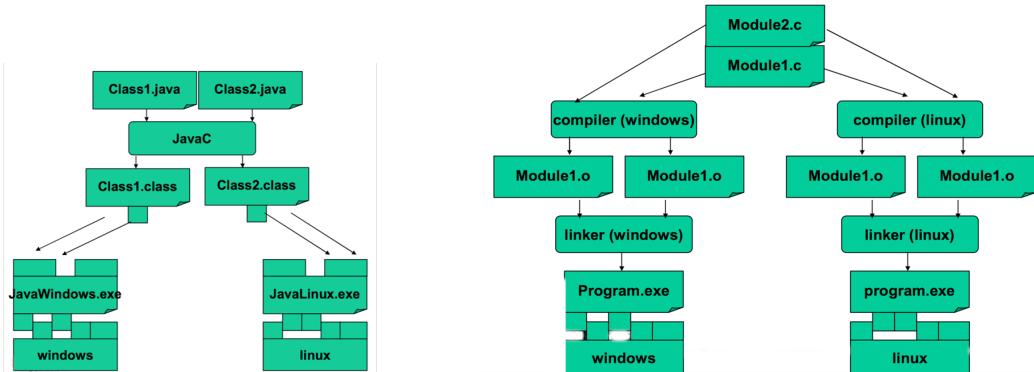
Storia dello sviluppo di Java



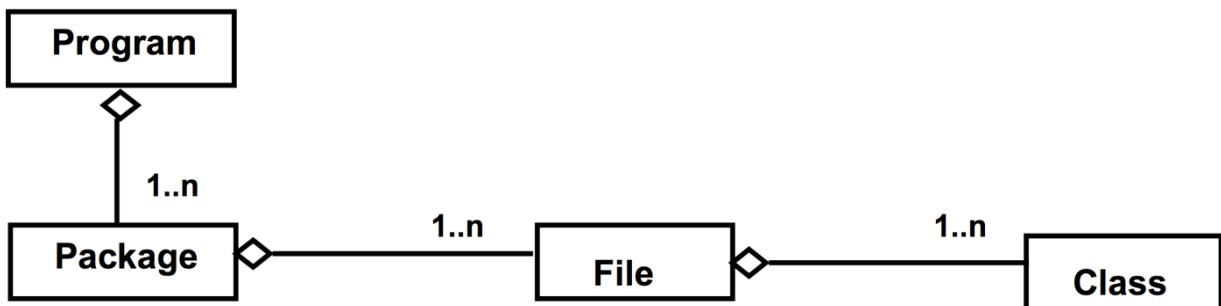
Ma in che modo Java riesce ad essere indipendente dalla macchina fisica? Vediamo il procedimento di compilazione ed esecuzione (Build and Run)



Tutto parte dal file .java che viene dato in pasto ad un compilatore specifico che produce come output un file .class, contenente il bytecode, un tipo di codice eseguibile dalla Java Virtual Machine. Il file .class viene dato alla JVM che interpreta il bytecode e lo converte in linguaggio macchina. Infine la JVM produce un file eseguibile, uno per ogni piattaforma (Windows,Linux,Unix), e lo invia al sistema operativo che eseguirà il programma.



La struttura di un programma Java è simile a quella di un modello ER, utilizzato nei database, con relazioni 1a1,1aN,ecc.... . L'unità organizzativa fondamentale di tutto il programma è il package, meccanismo per organizzare le classi Java in gruppi logici. All'interno del package possiamo trovare N file che a loro volta contengono 1 o più classi, a patto che ci sia una sola classe pubblica e tutte le altre private.



La funzione che rende eseguibile il codice ed il file .java, come anche nel C, é la procedura main

```
public static void main(String[] args){  
    ....  
}
```

2.1 Basics

Partiamo dalle basi di Java. Come in tutti i linguaggi di programmazione, in Java é possibile utilizzare:

- Commenti

```
//commenti su una linea  
/* commenti su piu  
linee */
```

- Strutture di controllo

```
//if-else  
if(x){  
    ...  
}else{  
    ...  
}  
  
//while  
while(x){  
    ...  
}  
  
//do-while  
do{  
    ...  
}while(x)  
  
//break, continue  
while(x){  
    if(y){  
        continue;  
    }else{  
        break;  
    }  
}  
  
//for  
for(int i = 0; i < N; i++){  
    ...  
}  
  
//foreach  
for(type_var: arr) {  
    //ciclo  
}  
  
//switch-case
```

```

switch(x){
    case 1:
        break;
    case 2:
        break;
    default:
        break;
}

```

- Il Passaggio dei parametri avviene per valore, ma con valore si intende una copia del riferimento all'oggetto e non il valore dell'oggetto. Bisogna quindi usare i parametri, cioè le copie dei riferimenti, per modificare lo stato interno dell'oggetto. Vediamo un chiaro esempio.

```

public class Parameters {
    public static void main(String[] args) {
        Point p1 = new Point(0,0);
        Point p2 = new Point(10, 10);
        System.out.println(p1);
        System.out.println(p2);

        swap(p1, p2);

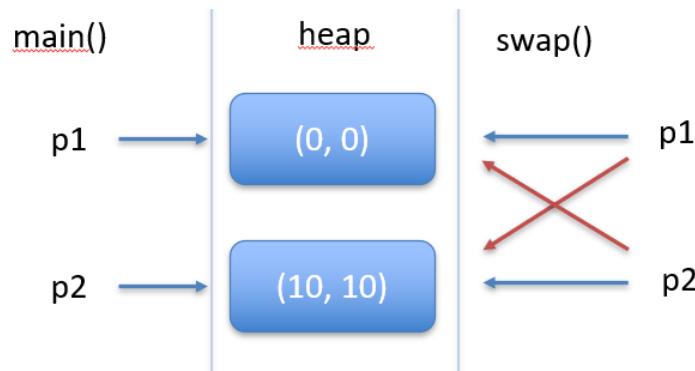
        System.out.println(p1); // 0, 0
        System.out.println(p2); // 10, 10
    }

    public static void swap(Point p1, Point p2) {
        Point tmp = p1;
        p1 = p2;
        p2 = tmp;
    }
}

```

La situazione descritta da questo pezzo di codice è questa:

Vengono scambiati i riferimenti e non i valori degli oggetti, proprio perché i parametri sono una copia dei riferimenti.



```

public class Parameters {
    public static void main(String[] args) {
        Point p1 = new Point(0, 0);
        Point p2 = new Point(10, 10);
        System.out.println(p1);
        System.out.println(p2);

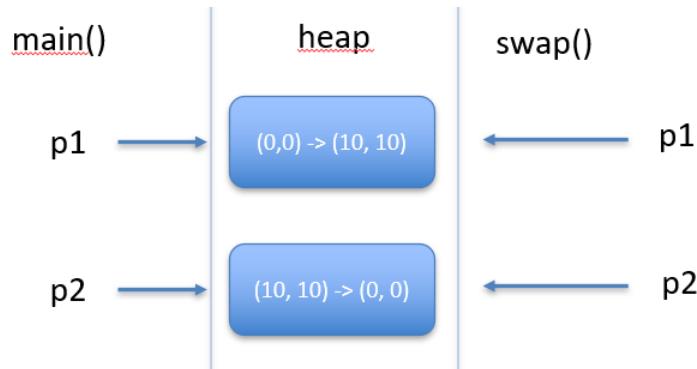
        swap(p1, p2);

        System.out.println(p1); // 10, 10
        System.out.println(p2); // 0, 0
    }

    public static void swap(Point p1, Point p2) {
        int p1X = p1.x;
        int p1Y = p1.y;
        p1.x = p2.x;
        p1.y = p2.y;
        p2.x = p1X;
        p2.y = p1Y;
    }
}

```

Per scambiare il valore di due oggetti dobbiamo ricorrere a quest'altra soluzione, in cui scambiamo direttamente i valori dei due oggetti senza ricorrere a riferimenti temporanei.



- Tipi primitivi: byte, char, short, int, long, float, double, boolean, void
- Per determinare una costante si utilizza il modificatore final e tipicamente, per convenzione, il nome della variabile da rendere costante è tutto maiuscolo.

```
final float PI = 3.1415
```

- Gli operatori, in Java, hanno la stessa sintassi del linguaggio C: =+*/!. Un'altra similitudine con il C è i caratteri sono utilizzano come interi; la più grande differenza è che gli operatori logici(if) lavorano solo con valori booleani, true e false, e non con gli interi come in C.
- Convenzione del codice e importanza dell'indentazione

```

ClassName /*i nomi delle classi si scrivono in camelCase
con anche la lettera iniziale maiuscola*/
attributeName // i nomi degli attributi si scrivono in camelCase

```

```
methodName // i nomi dei metodi si scrivono in camelCase
```

2.2 Strings

In Java non esistono tipi primitivi per rappresentare le stringhe. Essendo Java un linguaggio 'puramente' ad oggetti le stringhe, a differenza del C, non sono sequenze di caratteri ma istanze della Classe String. Quindi possiamo definire le stringhe come degli oggetti che mantengono una sequenza di caratteri alfanumerici. Per via della loro natura ad oggetto le stringhe sono immutabili e quindi i valori delle istanze non possono essere modificati ed inoltre il confronto, tipicamente, non viene fatto utilizzando l'operatore `==`. Vediamo un esempio.

```
//Stringhe C-like allocate nella memoria statica
String s1 = "ciao";
String s2 = "ciao";
if(s1 == s2){ //funziona
    System.out.println("Le due stringhe sono uguali");
}

//Stringhe Java like, cioe' oggetti istanziati nell'heap tramite malloc
String s3 = new String("Hello");
String s4 = new String("Hello");
if(s1 == s2){ //non funziona perche' confronta i riferimenti
    System.out.println("Le due stringhe sono uguali");
}
if(s1.equals(s2)){ //funziona perche' confronta il contenuto dei due
    oggetti
    System.out.println("Le due stringhe sono uguali");
}
```

Quindi se vogliamo confrontare due stringhe dobbiamo utilizzare il metodo `equals`, metodo della classe String. Essendo quindi la stringa un'oggetto, quindi un'istanza di una classe ad esso sono associati tutti gli attributi e tutti i metodi della Class String, tra i piú importanti troviamo:

- `length();` restituisce la lunghezza in int
- `substring(int start,int end);` restituisce una sottostringa dalla posizione start alla posizione end della stringa
- `charAt(int i);` restituisce il carattere alla posizione i
- `indexOf(int char);` restituisce la posizione della prima occorrenza del carattere char

Una delle operazioni piu comuni eseguita sulle stringhe é la concatenazione; esistono vari principalmente due modi per effettuarla:

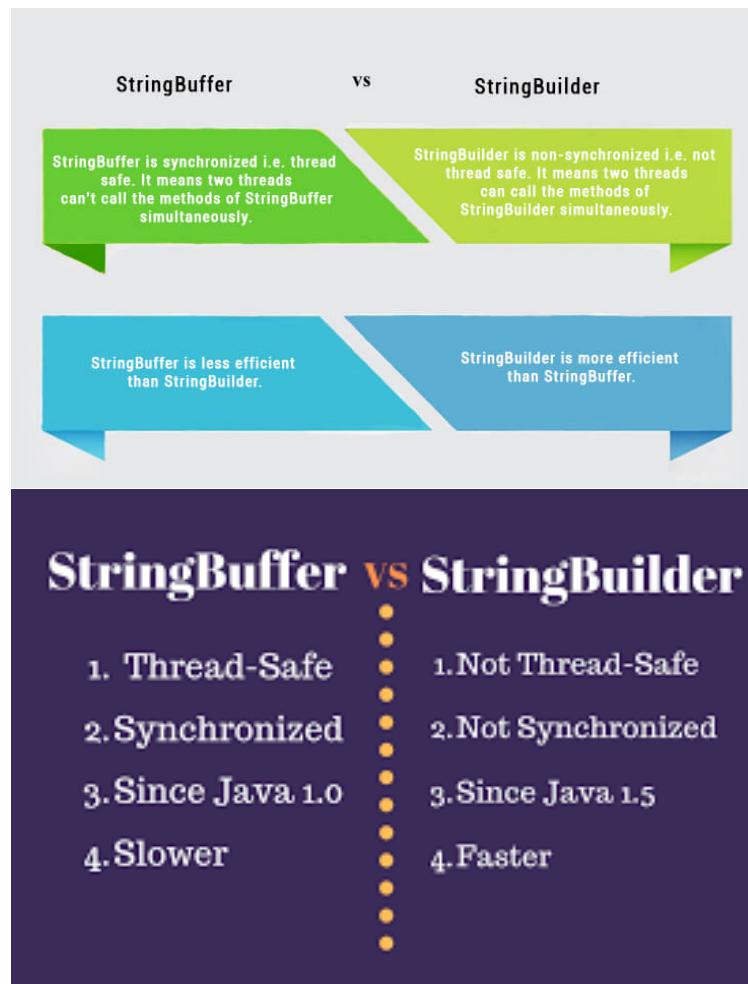
- Operatore `+`, questo tipo di concatenazione é molto lenta e dispendiosa

```
String result = "(s1 + " " + s2)";
```

- `StringBuffer`, se dobbiamo effettuare molte concatenazioni é utile utilizzare la classe `StringBuffer`

```
StringBuffer sb = new StringBuffer();
sb.append("a");
```

- In alternativa a `StringBuffer`, si puó utilizzare la classe `StringBuilder` che oltre ad essere piú veloce, non contiene, come `StringBuffer`, metodi già presenti nella class `String` e poco utilizzati.



2.3 Array

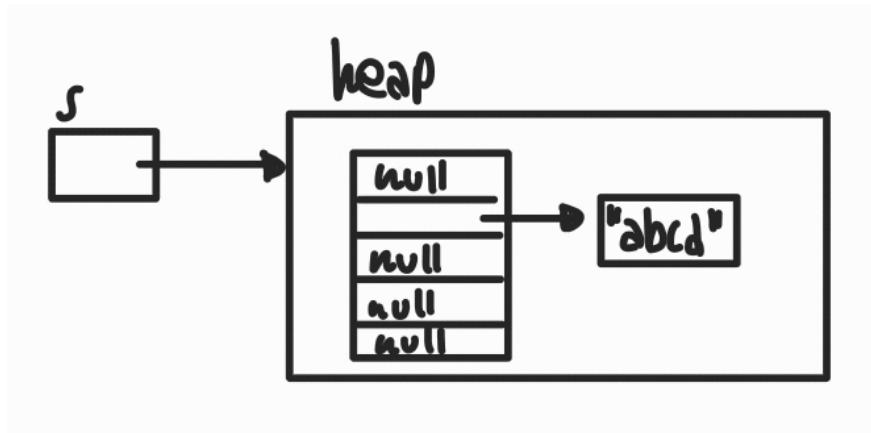
Gli array in Java possono contenere sia tipi primitivi che riferimenti ad oggetti. Anche l'array, come la stringa, è un oggetto e quindi deve essere istanziato dalla classe **Array** con l'operatore **new** e memorizzato nell'heap.

```
int[] v = new int[16]; /*inizializzazione classica,  
in cui tutti gli elementi del vettore vengono inizializzati a 0 */  
int[] v = {2,3,5,7,11}; //inizializzazione statica
```

Gli Array, come già detto, possono contenere anche dei riferimenti ad oggetti, vediamo come.

```
String[] s = new String[6];  
s[1] = new String("abcd");
```

La situazione in memoria é questa:



La dimensione dell'array é data dall'attributo interno length che describe la dimensione effettiva e non il numero di elementi contenuti. Un'importante particolaritá degli array in Java é che il riferimento ad un array non é un puntatore al primo elemento dell'array, infatti non esiste l'aritmetica dei puntatori, ma é un riferimento all'oggetto array.

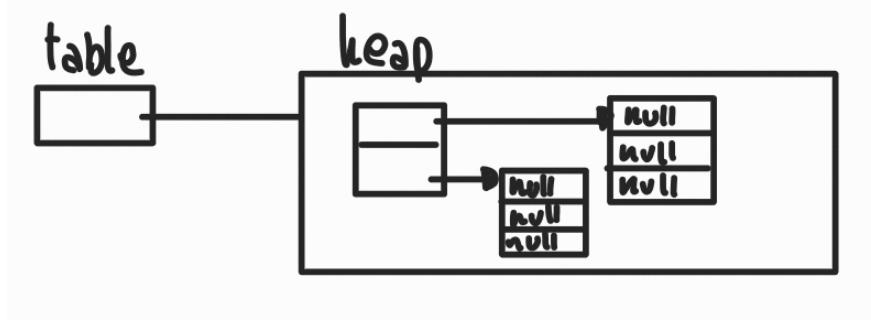
Con gli array, così come con le stringhe, torna molto utile il costrutto foreach.

```
Point [] v = new Point [6];
v[1] = new Point(1,0);
v[2] = new Point(1,5);
v[3] = new Point(5,7);

for(Point tmp : v){
    ...
}
```

Anche in Java é possibile definire degli Array Multidimensionali, da non confondere con le Matrici. In memoria heap la situazione é quella descritta dall'immagine sottostante.

```
Person [][] table = new Person [2] [3];
```



Per manipolare gli array ed effettuare ordinamenti, ricerche ecc.. possiamo utilizzare la `java.util.Arrays`, cioé una classe che contiene una serie di funzioni statiche (queste classi sono utilizzate in modo improprio, quasi come delle librerie del C). Le funzioni principali sono `sort()`, `fill()`, `toString()`, `binarySearch()`.

```
import java.util.Arrays;

public class Example {
```

```

public static void main(String args[])
{
    int [] arr = { 5, -2, 23, 7, 87, -42, 509 };

    Arrays.sort(arr); // il metodo non e' una funzione dell'oggetto
                      arr
    Arrays.fill(arr,10);
    Arrays.binarySearch(arr,5);

    System.out.println("The sorted array is: ");
    for (int num : arr) {
        System.out.print(num + " ");
    }
}
}

```

Un'altro metodo molto utile per gli array è `arraycopy()`, metodo della classe `System`.

2.4 Random Numbers

Per la generazione di numeri casuali possiamo utilizzare diversi metodi di differenti classi.

1. `Math.random()`
2. `java.util.Random`

```

Random random = new Random();
random.nextInt();

```

3. `java.util.random.RandomGenerator`

```

RandomGenerator generator = new Random();
RandomGenerator generator = RandomGenerator.getDefault();

```

4. `java.util.concurrent.ThreadLocalRandom`

```

ThreadLocalRandom.nextInt();
ThreadLocalRandom.nextDouble();

```

2.5 Classes

2.5.1 Definizione di classe

L'idea della classe come classificazione di una categoria di oggetti deriva dalla filosofia idealista, per la quale il mondo reale non è la realtà ultima ma solo il riflesso di un mondo ideale. La classe rappresenta l'idea perfetta e incorruttibile della categoria di oggetti, invece l'istanza è una rappresentazione concreta di una determinata classe.

```
public class Car{      //Nome della classe
    public boolean isOn; //Attributi
    public String getIsOn(){ //Metodi
        return isOn;
    }
}
```

Con `public` e `private` possiamo definire il livello di sicurezza e visibilità degli attributi, dei metodi e delle classi stesse. La convenzione impone che gli attributi delle classi siano privati e che per poterli lavorare in scrittura e lettura dobbiamo utilizzare le funzioni, messe a disposizione dalla classe, `getter()` e `setter()`. La funzione `getter()` ci permette di ricevere i valori degli attributi, invece il metodo `setter()` ci permette di modificare i valori degli attributi.

Visibility of fields in a class

Access modifiers →\Access location ↓	Public	Protected	Friendly (default)	Private Protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same Package	Yes	Yes	Yes	Yes	No
Other classes in same packages	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

La prima cosa fondamentale di un oggetto è il COSTRUTTORE. Se non definito esplicitamente, un costruttore di default viene automaticamente definito.

```
public point(double x,double y){
    this.x = x;
    this.y = y; // this e' una keyword per la classe per riferirsi a se' stessa
}
```

Un'altra funzione molto importante è il metodo `toString()` per visualizzare l'oggetto in una modalità testuale (senza definire il metodo, la funzione stampa a schermo il nome della classe con l'ID univoco dell'oggetto nella JVM)

```

public String toString(){
    return "x = " + x + "y = " + y ;
}

```

Entrambe queste due funzioni possono essere generate automatica in ambienti di sviluppo come IntelliJIdea. Vediamo ora come i principi dell'OOP vengono utilizzati in Java.

In una classe potrebbero esserci diversi metodi con nomi uguali ma firme diverse, cioè con numero e tipo di parametri diversi (la firma di un metodo è definita dal nome della funzione, dal tipo e dal numero di parametri). Questa pratica viene chiamata Method Overloading, la quale è un primo esempio di applicazione di polimorfismo e viene usata molto nella definizione di più costruttori per una singola classe.

```

public class Body {

    double x,y,vx,vy;

    public Body() {
        this.x = 0;
        this.y = 0;
        this.vx = 0;
        this.vy = 0;
    }

    public Body(double x, double y) {
        this.x = x;
        this.y = y;
        this.vx = 0;
        this.vy = 0;
    }
}

```

Una volta definito il costruttore, possiamo passare alla istanziazione di un oggetto della nostra classe, grazie all'operatore new.

```

Car c1 = new Car(Red,Fiat,False); /*viene creato il riferimento Car c1
all'oggetto Car */

```

Per deallocare la memoria occupata dagli oggetti, Java usa il Garbage Collector automatico che disalloca la memoria degli oggetti che non hanno riferimenti attivi.

```

Square s1 = new Square(); //definiamo due riferimenti a due oggetti
                        distinti
Square s1 = new Square();

s1 = s2; /*con questo assegnamento anche s1 sara' un riferimento per
           l'oggetto 'puntato' da s2 e l'oggetto 'puntato' in precedenza da s1
           non avra' piu riferimenti e verrà deallocated perch definito non
           attivo dal Garbage Collector*/

```

Una volta creato il riferimento e definito l'oggetto per poter accedere all'oggetto e quindi ai suoi metodi e ai suoi attributi utilizziamo la Dotted Notation.

```
System.out.println("Hello world");
```

In Java abbiamo la particolarità dei metodi static, cioè metodi che vengono richiamati direttamente sulla classe e non su un'istanza specifica. Il concetto di static si avvicina molto al concetto di librerie con funzioni scorrelate tra loro e generiche dei linguaggi procedurali.

```
Math.sqrt(16);  
Arrays.fill(arr);
```

Oltre ai metodi statici, esistono anche gli attributi statici, i quali sono condivisi tra tutte le istanze della classe e dalla classe stessa.

```
Point p1 = new Point(1,1);  
Point p2 = new Point(0,3);  
  
p1.nDimension += 1; /* stiamo modificando l'attributo sia dell'istanza a  
cui si riferisce p1, sia a quella di p2 e sia alla classe Point*/
```

In un linguaggio totalmente ad oggetti dovrebbero esistere solo classi ed oggetti. In Java, però, per questioni di efficienza sono stati introdotti anche i tipi primitivi, dei quali però esiste anche la versione ad oggetto chiamate Wrapper Classes.

Boolean, Integer, Characters, Byte, Short, Float, Double, Long, Void sono tutte classi wrapper che principalmente offrono dei servizi(metodi) di conversione.

```
String s = "10.6f";  
  
float x = Float.parseFloat(s);  
  
System.out.println(x); //10.6
```

La conversione da tipo primitivo a classe wrapper è automatica ed è definita come autoboxing, viceversa la conversione da classe wrapper a tipo primitivo è definita autounboxing.

Riporto la definizione della documentazione ufficiale di Java:

“Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.”

Per organizzare e gestire al meglio classi di dimensioni notevoli, in Java sono stati definiti i cosiddetti Package che, come già spiegato, è un meccanismo per organizzare le classi Java in gruppi logici. Per importare nel main classi risiedenti in altri package possiamo utilizzare la keyword import.

```
import packageName.className;
```

Nota molto importante è che due classi possono avere due nomi uguali se risiedono in due package diversi, ma le due classi non possono essere importate insieme.

```
import java.sql.Date;  
import java.util.Date; //wrong  
  
import java.sql.Date;  
Date d1 = new Date();  
java.util.Date d2 = new java.util.Date(); //right
```

2.5.2 Override

Molto spesso, una classe é una piccola modifica rispetto ad un'altra classe. L'ereditarietá permette di minimizzare la ripetizione dello stesso codice e di localizzare il codice nelle classi. La classe che specializza contiene tutti gli attributi e i metodi della classe padre, puó definire attributi e metodi aggiuntivi e sovrascrivere i metodi ereditati, tramite override. In Java per utilizzare il concetto di ereditarietá si utilizza la keyword extends.

```
class Car{
    boolean isOn;
    void turnOn(){
        ...
    }
}

class SDCar extends Car{
    boolean isSelfDriving;
    void turnSDOff(){
        ...
    }

    @Override
    void turnOn(){
        turnSDOff();
        super.turnOn(); // super e' la keyword per rifersi alla classe
                        padre
    }
}
```

Con la keyword super possiamo accedere alle funzioni e agli attributi della superclasse. Invece con super()/super(params) chiamiamo nella classe figlia il costruttore di default o con parametri della classe padre.

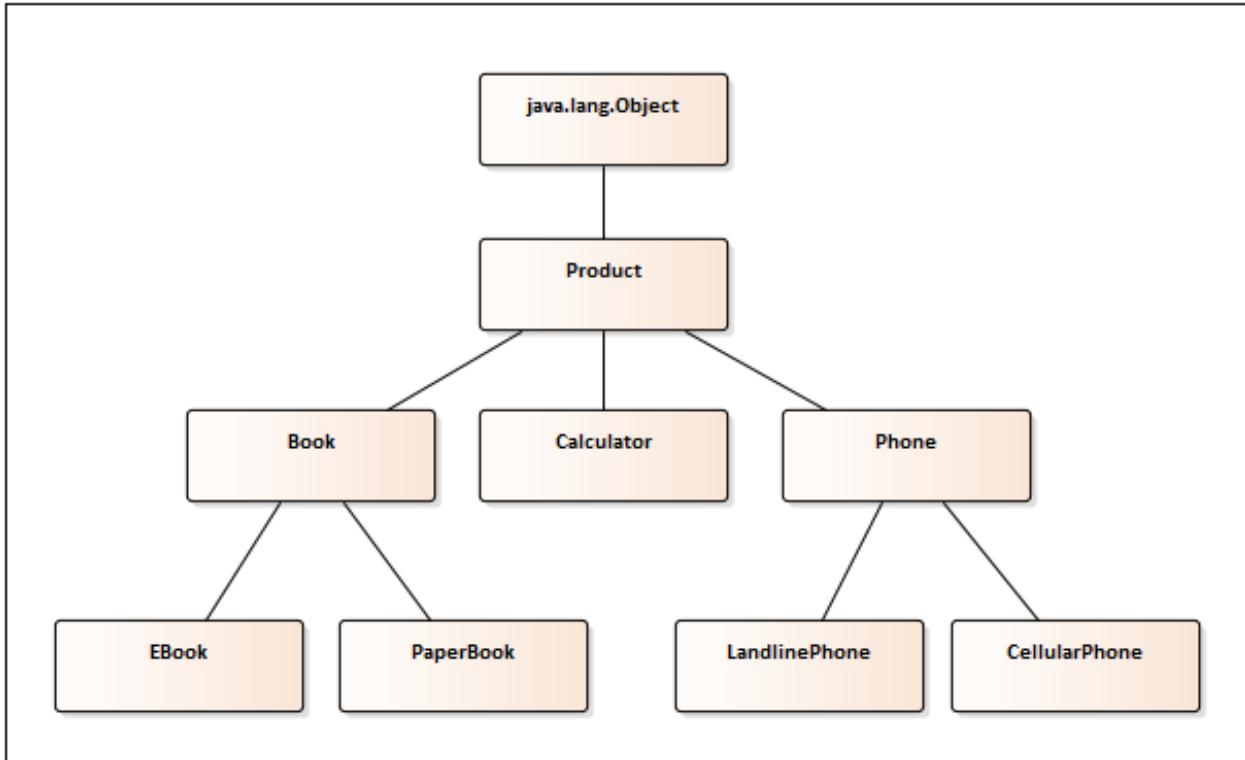
```
public SDCar(String licencePlate){
    super(licencePlate); /* la prima linea deve essere sempre la chiamata
                           al costruttore della classe padre */
    isSelfDriving = false;
}
```

Abbiamo definito, in precedenza, Java un linguaggio fortemente ad oggetti perché anche la struttura del linguaggio é ad oggetti.

Cerchiamo di chiarire con un esempio: ogni classe deriva da un antenato comune che é la classe java.lang.Object, nella quale sono definiti i servizi base come `toString()`, `equals(Object o)`, `clone`. Tutti questi metodi vengono ereditati dalle classi figlie (quindi tutte) che devono sovrascrivere questi metodi per renderli utilizzabili. In particolare, bisogna sempre fare l'override del metodo `equals`, cosí da determinare per quali attributi confrontare due oggetti. [In IntelliJIdea questi metodi possono essere autogenerati]. Notiamo, però, che quando definiamo una nuova classe non dobbiamo fare l'extends da Object, perché? Questo succede perché Java ci facilita il lavoro e l'extends ad Object é implicito e automatico.

2.5.3 Upcast e Downcast

Dopo aver esplicitato il concetto di ereditarietá, cerchiamo di concentrarci sul significato di riferimento di un oggetto e cosa é possibile fare con esso.



```

class Car{};
class SDCar extends Car{};

Car c3 = new SDCar(); /* in questo stiamo creando un riferimento di tipo
car che "punta" un oggetto SDCar; */
  
```

Da questo semplice esempio possiamo capire il concetto di riferimento, il quale é come un filtro che d la possibilt di accedere o meno ai metodi di una determinata classe. In questo caso il riferimento c3 pu accedere ai metodi di Car nonostante si riferisca ad un oggetto di tipo SDCar. Da questa propriet del riferimento possiamo distinguere due possibili casi:

- Upcast, quando il riferimento ´ di tipo maggiore rispetto all'oggetto puntato (con tipo maggiore intendiamo che ´ di grado pi alto nella gerarchia delle classi)

```
Car c3 = new SDCar(); // Car e' padre di SDCar
```

Procedura automatica, affidabile e sicura ma si sta generalizzando rispetto all'oggetto SDCar e quindi stiamo perdendo l'accesso ha metodi specializzati.

- Downcast, quando il riferimento ´ di un tipo minore rispetto all'oggetto puntato

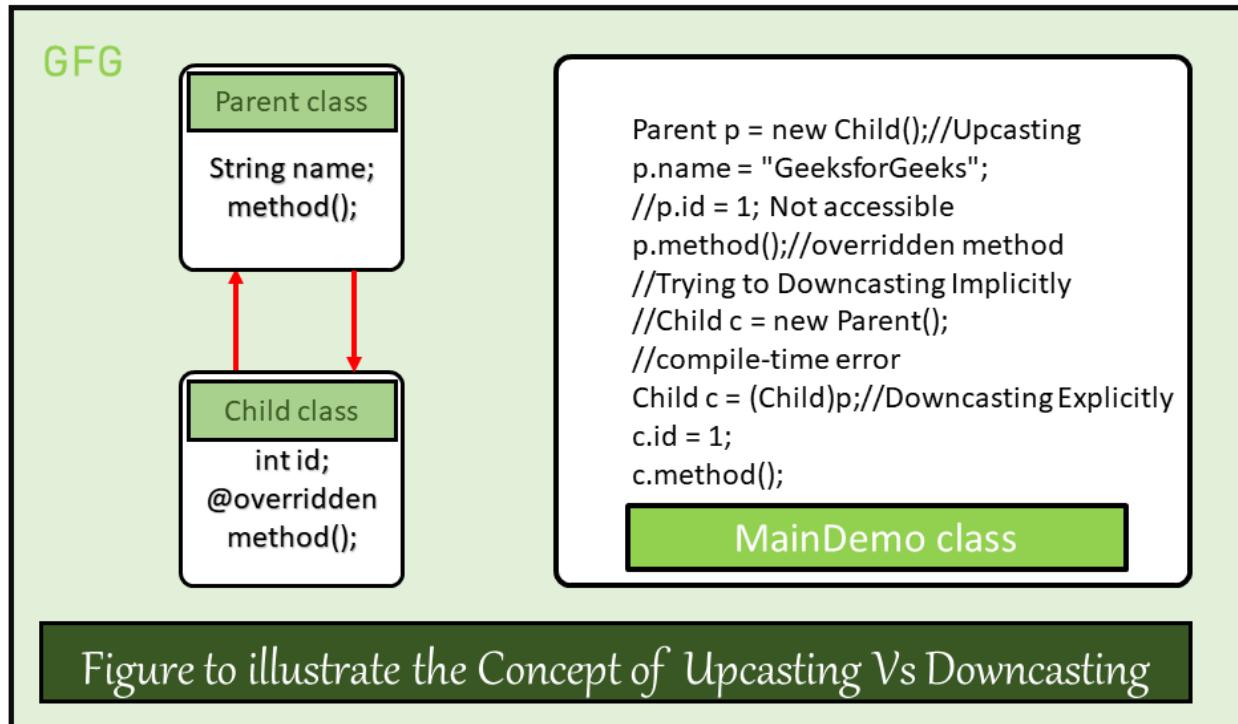
```
SDCar c1 = (SDCar) c3;
```

Procedura rischiosa perch il downcast deve essere esplicito e inoltre per funzionare anche c3 si deve riferire ad un oggetto del tipo di c1 (in questo caso anche c3 si deve riferire ad un oggetto SDCar). Per evitare errori a Runtime durante un downcasting dobbiamo utilizzare sempre l'operatore instanceof, il quale ci permette di capire se un determinato riferimento si riferisce ad un determinato oggetto.

```
if (c instanceof SDCar){
    SDCar sdc = (SDCar) c;
```

```
sdc.turnSDOn();  
}
```

È sempre possibile fare upcast e downcast verso e da Object, visto che è la classe comune a tutte le altre classi.

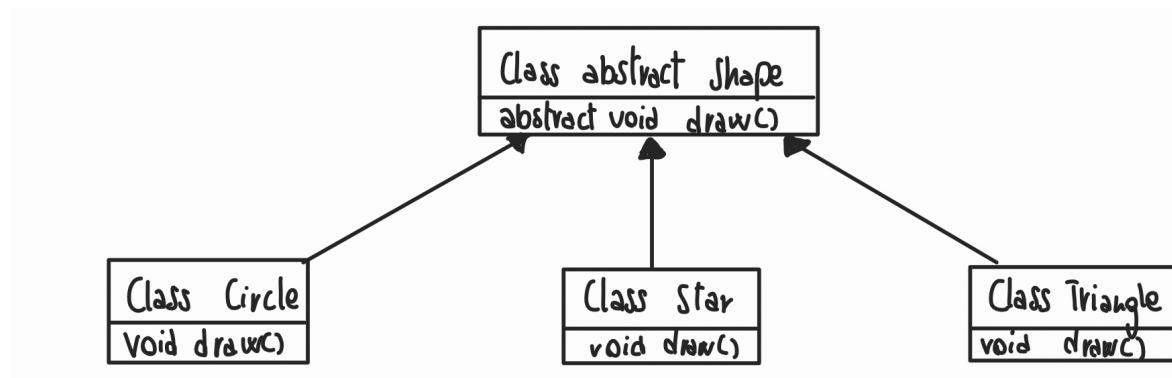


2.5.4 Metodi Abstract e Interfacce

In Java è possibile dichiarare un tipo di metodo particolare chiamato Abstract Method, il quale è fondamentalmente un metodo senza implementazione, senza il corpo del metodo.

```
public abstract void draw(int size);
```

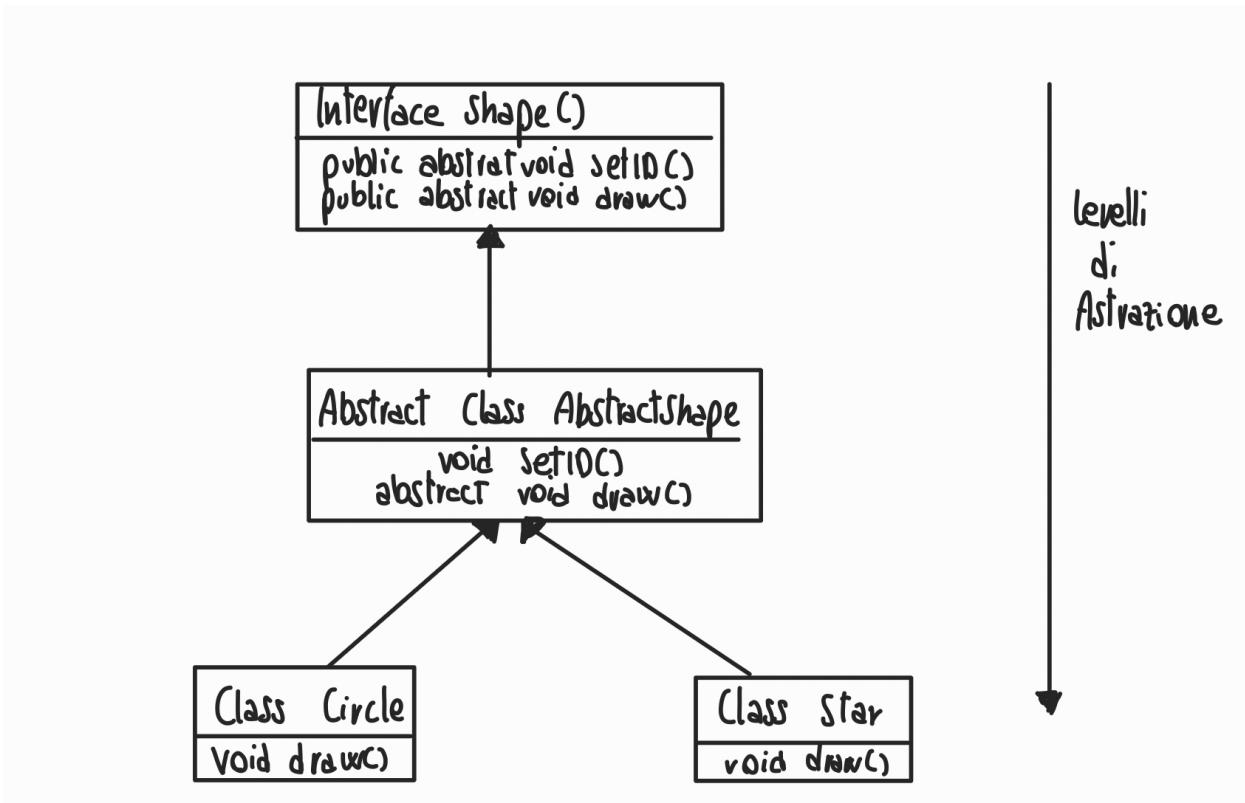
Di conseguenza una classe astratta è una classe che contiene uno o più metodi astratti e che non può essere istanziata. Ma qual è il vantaggio e l'utilità delle classi astratte? Il concetto delle classi astratte è molto utile quando si vuole definire una classe padre generale che contiene tutti i metodi comuni a tutti i figli senza però sviluppare la funzione, lasciando ai figli l'implementazione specifica. Infatti è possibile estendere le classi astratte a patto che le classi specializzate forniscano un'implementazione dei metodi astratti ereditati. A loro volta le classi specializzate possono essere definite astratte, creando una gerarchia di classi astratte. Vediamo un esempio.



Per evitare l'ambiguità dell'esistenza di metodi astratti e metodi non astratti all'interno di una classe è stata introdotto in Java il concetto di Interfaccia. L'Interfaccia è simili ad una classe astratta ma contiene solo metodi pubblici e astratti e costringe tutte le classi che implementano l'interfaccia di definire il corpo delle funzioni astratte ereditate.

```
public interface Bikers{  
    public abstract String getBike();  
    public abstract String getPU();  
}  
  
public abstract Person implements Bikers{  
    ...  
}
```

È possibile, come per le classi astratte, estendere le interfacce e quindi aggiungere ulteriori metodi. Con l'utilizzo combinato delle classi astratte e delle interfacce possiamo creare vari livelli di astrazione che vanno dalla classe più astratta (interfaccia) alla classe specializzata (classe).



In Java una classe può estendere solo un'altra classe ma può implementare diverse interfacce.

```
class Application extends JFrame implements ActionListener, KeyListener {
    ...
}
```

2.5.5 Approfondimenti

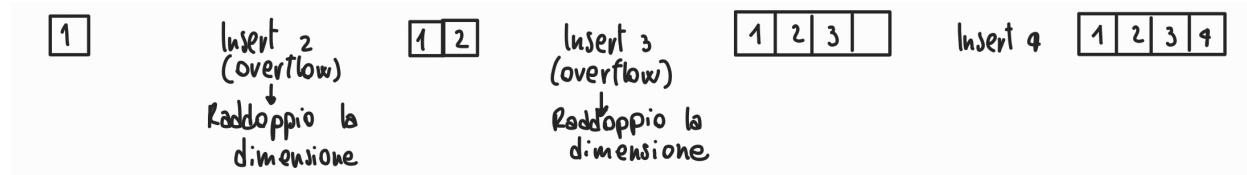
1. Operatore instanceof: controlla che la variabile sia un membro di una classe o di una interfaccia, verificando che la classe contenga i metodi della classe genitori e delle interfacce che implementa.
2. La classe Point ha già un costruttore di default che permette di fare una copia di un punto. In tutte le altre classi, invece, dobbiamo fare l'override del metodo di clone();

```
Point p1 = new Point(p2);
```

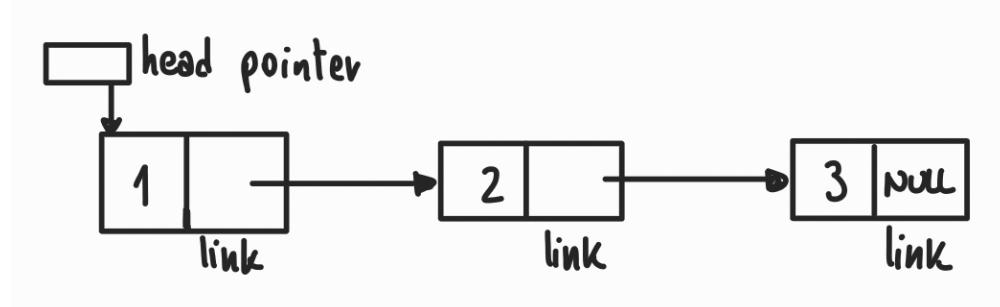
2.6 Java Collections

Il JCF o Java Collections Frameworks é un gruppo di classi ed interfacce che permettono di manipolare le cosidette data structures. È il package `java.util` che mette a disposizione del programmatore il package `java.util.Collection` che contiene interfacce, classi astratte e concrete e algoritmi per manipolare la maggior parte delle strutture dati moderne. Tutte le Collection si basano su 4 principali strutture dati:

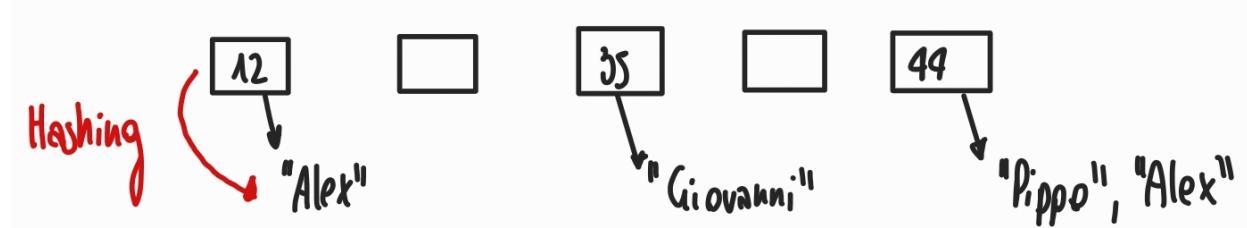
1. Array Ridimensionabile $\sim O(n)$: Array che raddoppia la dimensione ogni volta che inserendo un elemento si provoca un overflow.



2. Linked list $\sim O(n)$: Lista formata da valore e puntatore all'elemento successivo

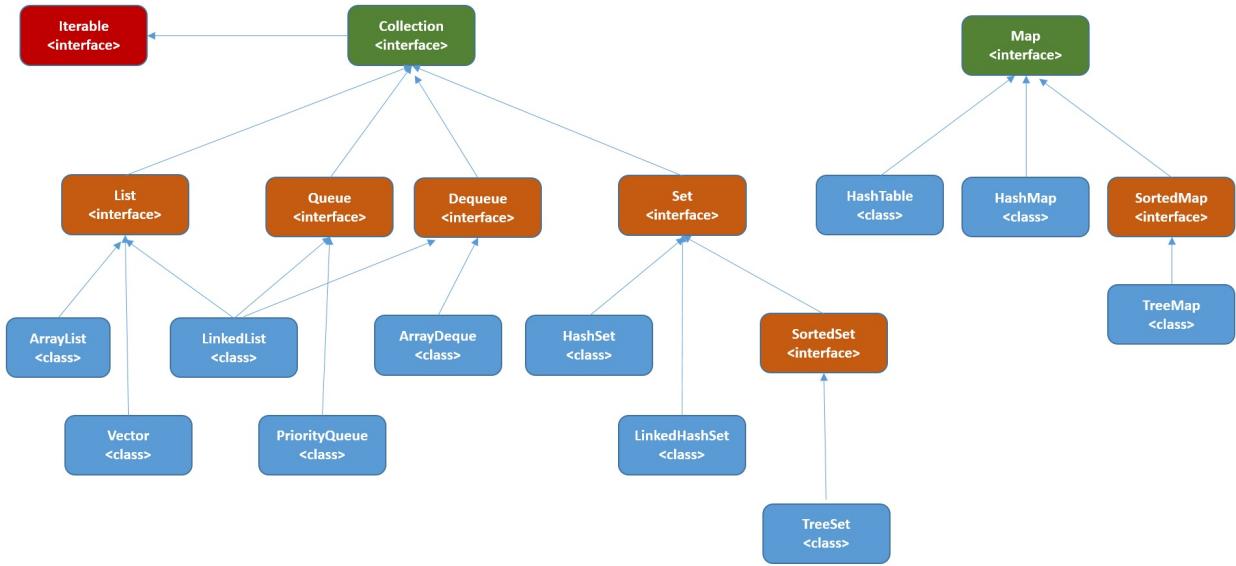


3. Balanced tree $\sim O(\log_2 n)$: Albero bilanciato, in cui per ogni nodo la differenza tra l'altezza del sottoalbero sinistro e sottoalbero destro è al massimo 1.
4. Hash Table $\sim O(1)$: Struttura dati basata sulla funzione di hash, cioè basata su una funzione che associa ad un elemento un numero in base al contenuto dell'elemento. Questo riduce la complessità della ricerca a $O(1)$ cioè la ricerca non dipende dalla posizione dell'elemento.



Come tutti i framework di Java, anche la JFC ha un determinata gerarchia.

Collection Framework Hierarchy



data structure

	Hash table	Resizable array	Balanced tree	Linked list	Hash table Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

interface classes

Tutta la gerarchia parte dall'interfaccia Iterable che contiene solo un elemento iteratore(interface Iterator), il quale ci permette di scorrere un insieme di oggetti. L'interfaccia Iterator mette a disposizione tre metodi principali: hasNext() che ci dice se esiste il prossimo elemento, next() che ci restituisce il prossimo elemento e remove() che elimina l'elemento corrente.

```
public interface Iterable<T>{
    public Iterator<T> iterator();
}
```

Dall'interfaccia Iterable passiamo all'interfaccia Collection che rappresenta un gruppo di elementi generale e non specificato. Quindi può rappresentare sia insiemi ordinati che non ordinato, sia insiemi di elementi duplicati che di elementi non duplicati e così via. L'interface Collection contiene due costruttori e diverse funzioni generali applicabili a tutte le strutture dati:

- Collection(), costruttore base
- Collection(Collection C), costruttore che genera una Collection partendo da un'altra Collection
- int size(), restituisce la dimensione
- boolean isEmpty(), segnala se la Collection è vuota o meno

Nonostante queste due interfacce siano i "capostipiti" della gerarchia delle Collection, sono le meno utilizzate per via della loro generalità. Le interfacce più utilizzate a causa della loro specificità sono le seguenti: List Interface, Set Interface, Queue Interface, Map Interface. Per ognuna di queste interfacce esistono diverse implementazioni basate su Array ridimensionabile, LinkedList, Hash Table e Balanced Tree.

2.6.1 List Interface

L'interfaccia List permette di manipolare insiemi di elementi/oggetti duplicati, in cui l'ordine di inserimento è preservato e in cui si può accedere agli elementi tramite la posizione. I principali metodi sono i seguenti:

- Object get(int index), restituisce l'elemento alla posizione index
- Object set(int index, Object o), sostituisce l'elemento alla posizione index con l'elemento o
- add(Object o), inserisce un elemento in coda
- remove(int index), rimuove e restituisce l'elemento alla posizione index
- remove(Object o), rimuove la prima occorrenza dell'elemento o

```
List<Object> l = new ArrayList<Object>(); /* all'interno delle <> viene
    specificato il tipo di elementi della lista*/
List<Object> l = Arrays.asList(14, 73, 18); //metodo di java.util.Arrays
List<Object> l = new LinkedList<Object>();
```

Dell'interfaccia List esistono 2 implemtazioni:

1. **ArrayList**, basato su Array ridimensionabile e in cui il metodo get ~ O(cost) e il metodo add ~ O(n)
2. **LinkedList**, basato su liste linkate e code ed in cui sia il metodo get che il metodo add ~ O(n)

2.6.2 Set Interface

Rispetto a Collection non contiene funzioni aggiuntive e quindi non specializza ma permette di creare strutture dati che non accettano duplicati. I Set vengono principalmente utilizzati per rimuovere gli elementi duplicati da una lista, grazie al costruttore di Collection.

Dell'interfaccia Set esistono 3 diverse implementazioni:

1. **HashSet**, basata su hashTable e quindi veloce ma non mantiene l'ordine di inserimento
2. **LinkedHashSet**, similare a HashSet ma mantiene l'ordine di inserimento

3. TreeSet, basato su BalanceTree e quindi l'ordine deve essere definito

```
Set<String> s = new HashSet<String>();
Set<String> s = new LinkedHashSet<String>();
Set<String> s = new TreeSet<String>();
Set<String> s = new HashSet<String>(list);
```

2.6.3 Queue Interface

Permette di controllare delle code, cioè dei gruppi di elementi con priorità (un video su Youtube è una coda di frame). Per far ciò aggiunge i concetti di head e tail e diverse funzioni:

- Object peek(), restituisce la testa della coda e in caso la coda fosse vuota ritorna null(like C)
- Object element(), restituisce la testa della coda ma in caso di errore utilizza le eccezioni
- Object poll(), restituisce e rimuove la testa della coda e in caso la coda fosse vuota ritorna null(like C)
- Object remove(), restituisce e rimuove la testa della coda ma in caso di errore utilizza le eccezioni

Di questa interfaccia esistono 2 implementazioni:

1. **LinkedList**, basata su liste e code(First In First Out Policy)
 2. **PriorityQueue**, basato su Balanced Tree e di cui può essere definito l'ordinamento interno attraverso le interfacce Comparable e Comparator(vedremo in seguito). Sia PriorityQueue che TreeSet sono basate su Balanced Tree ma hanno notevoli differenze.
- **Similarities**
 - Both provide $O(\log(N))$ time complexity for adding, removing, and searching elements
 - Both provide elements in sorted order
 - **Differences**
 - **TreeSet is a Set and doesn't allow a duplicate element**, while PriorityQueue is a queue and doesn't have such restriction.
 - Another key difference between TreeSet and PriorityQueue is *iteration order*, though you can access elements from the head in a sorted order e.g. head always give you lowest or highest priority element depending upon your Comparable or Comparator implementation but **iterator returned by PriorityQueue doesn't provide any ordering guarantee**.

2.6.4 Map Interface

Interfaccia che permette di manipolare "contenitori" di coppia chiave-valore, cioè ad ogni chiave è associato un valore; questa caratteristica è il motivo principale per il quale l'interface Map non deriva da Iterable. I principali metodi di questa interfaccia sono:

- Object put(Object Key, Object Value), inserisce una coppia chiave-valore
- Object get(Object Key), restituisce il valore alla chiave specificata da parametro

Anche dell'interfaccia Map esistono diverse implementazioni:

1. **HashMap**, basata su hashTable e quindi veloce ma non mantiene l'ordine di inserimento e di cui si può definire la dimensione iniziale e il carico massimo attraverso il proprio costruttore.

2. **LinkedHashMap**, simile a **HashMap** ma mantiene l'ordine di inserimento.

3. **TreeMap**, basato su Balanced Tree.

```
Map<String, Integer> m = new HashMap<String, Integer>();  
  
m.put("Luciano", 3);  
m.put("Agata", 2);  
m.put("Luciano", 7); // sostituiamo l'associazione precedente della chiave  
                    Luciano  
  
// scorrere gli elementi di una mappa attraverso il foreach  
for(int i: m.keySet()) {  
    System.out.println(i + " --> " + m.get(i));  
}
```

Per capire quale struttura dati utilizzare nelle varie situazioni dobbiamo sempre valutare le operazioni che dobbiamo fare per poi poter scegliere la Collection che ha un tempo medio minore sulle varie operazioni da svolgere (inserimento, ordinamento, ricerca, ...).

2.6.5 Iteratori

Le varie strutture dati possono essere visualizzate a schermo scorrendole con i metodi classici (foreach) ma se dobbiamo anche modificarle dobbiamo stare molto attenti perché possono verificarsi errori a Runtime. In questo caso dobbiamo utilizzare i cosiddetti iteratori, cioè degli elementi sviluppati ad hoc per scorrere le varie Collection:

- **Iterator**, utilizzato per rimuovere elementi e procede solo in avanti
- **ListIterator**, utilizzato per rimuovere ed aggiungere elementi e può scorrere la Collection in entrambe le direzioni

```
for(Iterator<Person> i = p.iterator(); i.hasNext()){  
    Person p = i.next(); // vado all'elemento successivo  
    System.out.println(p);  
}
```

2.6.6 Ricerca ed Ordinamento

Due delle operazioni più frequenti fatte sulle Collection sono la ricerca di un elemento (`search()`) e ordinamento della struttura dati (`sort()`). Java mette a disposizione la classe statica `java.util.Collections` che mette a disposizione una serie di metodi che ci permettono di controllare le varie strutture dati.

- `sort()`, mergesort
- `binarySearch()`, applicabile solo su liste ordinate, altrimenti il risultato è indefinito
- `shuffle()`
- `reverse()`
- `min()` e `max()`
- `rotate()`

L'ordinamento di una collections di oggetti non è definito a priori ma Java mette a disposizione due interfacce per definire l'ordine che sono del tutto analoghi.

1. L'interfaccia Comparable, la quale deve essere implementata all'interno della classe dell'oggetto definendo la funzione compareTo.

```
public interface Comparable<T>{
    public int compareTo(T obj);
}
```

2. Interface Comparator, la quale deve essere implementata in una classe specifica

```
public interface Comparator<T>{
    public int compare(T obj1,T obj2);
}
```

Entrambe queste funzioni ritornato un int che avrà valore minore di 0 se this.object precede obj, = 0 se this.object si trova nella stessa posizione di obj e maggiore di 0 se this.object segue obj.

```
class Student implements Comparable<Student>{
    String name;
    int age;
    Student(String name,int age){
        this.name=name;
        this.age=age;
    }

    public int compareTo(Student st){
        if(age==st.age)
            return 0;
        else if(age>st.age)
            return 1;
        else
            return -1;
    }
}

import java.util.*;
class AgeComparator implements Comparator{
    public int compare(Object o1,Object o2){
        Student s1=(Student)o1;
        Student s2=(Student)o2;

        if(s1.age==s2.age)
            return 0;
        else if(s1.age>s2.age)
            return 1;
        else
            return -1;
    }
}
```

2.7 Generics

Un tipico errore che si può presentare in Java è il ClassCast.

```
List l = new ArrayList();
l.add("apple");
l.add(4);
System.out.println((String) l.get(1)); /* essendo Java un linguaggio
    fortemente tipizzato questa operazione provoca errore
    ClassCastException */
```

Per rimuovere il rischio di ClassCastException a runtime e per fornire type checking a tempo di compilazione sono stati introdotti in Java 5 i Generics. Un generics è uno strumento che permette la definizione di un tipo parametrizzato, che viene esplicitato successivamente in fase di compilazione; i generics permettono di definire delle astrazioni sui tipi di dati definiti nel linguaggio.

Grazie ai Generics riusciamo a sviluppare metodi ed interfacce del tutto generali e utilizzabili per qualsiasi tipo di dato.

```
public interface List<T> extends Collection<T>{
    // con T specifichiamo il tipo di oggetto
    ...
}

public static void shuffle(List<?> list){
    // ? e' una wildcard per specificare che la lista puo' di essere di un
    // qualunque tipo
}
```

Attraverso i generics possiamo anche sviluppare classe del tutto generali. Vediamo l'esempio di uno shop generico che funziona sia con i singoli oggetti che con le collection.

```
public class Shop<T> {
    Queue<T> items;

    /* buy and sell a single item */
    void buy(T item);
    T sell();

    /* buy and sell a group of items */
    void buy(Collection<T> cart);
    void sell(Collection<T> cart, int n);

    /* returns the internal items */
    Collection getItems();
}

//single object
Shop<Fruit> fruitShop = new Shop<>();
fruitShop.buy(new Fruit());

//collection
Shop<Fruit> fruitShop = new Shop<>();
fruitShop.buy(List.of(new Fruit(), new Fruit(), new Fruit()));

//subtype single object
Shop<Fruit> fruitShop = new Shop<>();
```

```

fruitShop.buy(new Orange()); //Orange e' figlio di Fruit
Product product = fruitShop.sell();

//subtype collection
Shop<Fruit> fruitShop = new Shop<>();
List<Orange> OList = new ArrayList<>(List.of(new Orange(), new
Orange()));
fruitShop.buy(OList); //wrong

```

Il problema generale è il subtyping con le collection perché non valgono i rapporti di ereditarietà tra Collection e quindi non è possibile utilizzare i metodi generali (buy e sell in questo caso). Questo avviene perché non si può effettuare il cast al tipo più generale Object e poi inserire in quella collection Object qualsiasi tipo perché verrebbe meno la forte tipizzazione di Java e quindi torniamo al problema iniziale. Per ovviare a questo problema è stato introdotto la lista di unknown, cioè la lista più generale di tutte (non esiste la lista di Object); a questo tipo di lista non si può aggiungere nulla(tranne NULL) e quindi risolve il problema dell'upcasting. Da questa lista si può solo prelevare e trattare istanze di Object e attraverso le wildcard super,extends possiamo anche ridurre il livello di incertezza.

```

List<?> ol = pl; //pl e' una lista qualsiasi

List<? extends Fruit> /* contiene qualsiasi cosa che specializza Fruit
solitamente utilizzato per la lettura dei valori */

List<? super Fruit> /* contiene qualsiasi cosa piu' generica Fruit
solitamente utilizzato per la scrittura dei valori */

<T extends Comparable<T>> /* accetta tutti i tipi che estendono
comparable cioe' hanno implementato compareTo */

<T extends Comparable<? super T>> /* cosi' permettiamo di confrontare un
tipo con se stesso o con qualcosa di piu' generico */

```

Per poter scrivere su lista di unknown dobbiamo fare il cast alle liste pre-generics facendo sempre molta attenzione perché potremmo tornare al solito problema di ClassCastException

```

List<?> list;
final List l = list;

```

I generics vengono utilizzati anche in un alcuni dei metodi già visti:

```

public static void reverse(List<?> list){} /* viene utilizzato ? quando
non ci sono vincoli sui parametri */

public static <T> void sort(List<T> list, Comparator <? super T> c){}
/*viene utilizzato T quando non ci sono vincoli sui parametri e sul
valore di ritorno*/

```

Nonostante l'introduzione delle generics, la JVM non vede mai queste tipo di strutture e wildcards perché tutto è rimasto come al mondo pre-generics per avere una maggiore velocità; solo a livello di compilazione viene fatto il type checking. Per esempio, la JVM esegue le seguenti code erasures o sostituzioni:

- List<String> → List
- T[] → Object[]

2.8 Exception

Vediamo come implementare una corretta ed efficiente gestione degli errori cercando di non impattare troppo sulle performance. In Java possiamo usare diversi sistemi di gestione errore:

- System.exit(), poco efficiente e da non utilizzare all'interno di una funzione
- return -1, approccio bash che utilizza, però, codici non facilmente ricordabili e non univoci
- Exceptions

```
try{ //blocco che esegue operazioni
    open file;
} catch(fileOpenFailed){ // blocco gestione eventuali errori
    doSomething;
}
```

In Java, a differenza di altri linguaggi OOP, esistono 2 diversi tipi di Exceptions:

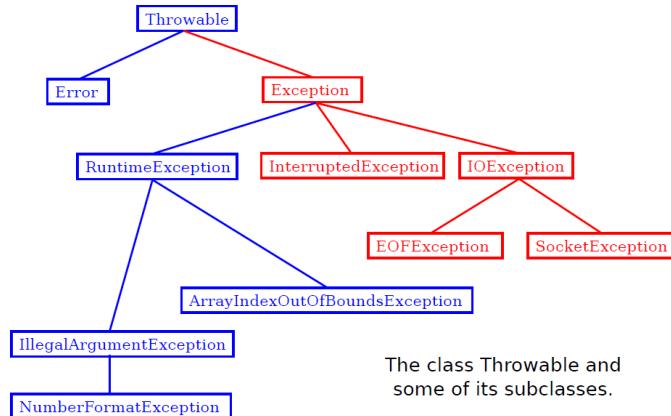
- **Check**, vengono segnalate dal compilatore come errore

```
BufferedReader r = new BufferedReader(new FileReader("filename"));
```

- **Uncheck**, non vengono segnalate dal compilatore come errore

```
List<String> l = new ArrayList();
l.get(122); /*stiamo cercando di inserire un int in una lista di
stringa ma non ci viene segnalato nulla*/
```

Come ogni classe java, anche le Exceptions hanno una gerarchia di classi ed interfacce.



La gestione degli errori, quindi, può avvenire in diversi modi:

- Intercettare le exceptions con try-catch ("sporca" notevolmente il codice)

```
try{
    ...
}catch(Exceptions e){
    e.method(); /* le eccezioni all'interno della clausola catch
                  devono rispettare la gerarchia delle classi Exception */
}finally{
    // clausola che accetta tutte le eccezioni non intercettate
}
```

- Delegare le exceptions ad una funzione specifica, intercettando l'eccezione nel main che a sua volta potrà delegare ad un'altra funzione

```
public static String read(String filename) throws IOException{
    BufferedReader r = new BufferedReader(new FileReader(filename));
    return r.readLine();
}
```

- Sviluppare le proprie classi per la gestione degli errori, generando eccezioni con la keyword throw

```
if(b == 0){
    throw new IllegalArgumentException();
}
```

- Delega parziale, utilizzando sia il try catch che la keyword throw (per una maggiore chiarezza verso l'utente finale)

```
try{
    BufferedReader r = new BufferedReader(new FileReader(filename));
    return r.readLine();
}catch(IOException e){
    throw new RuntimeException();
}
```

Per utilizzare le eccezioni nella gestione degli errori durante un ciclo su una Collection abbiamo principalmente 2 modi:

```
//1
while(something){
    try{
        ...
    }catch(Exception e){
        ...
    }
}
```

```
//2
try{
    while(something){
        ...
    }
}catch(Exception e){
    ...
}
```

Infine possiamo anche generare eccezioni costum , cioè programmate da noi ma è una pratica abbastanza inutile visto che esistono già tantissime Exception nell'API di Java). La classe di riferimento per le eccezioni è java.lang.Exception.

2.9 Approfondimento Composition

Un concetto molto importante, che differisce notevolmente dalla ereditarietà, è la composizione. La composizione si basa sul definire classi separate per rappresentare oggetti separati tra loro per poi integrarli insieme in una applicazione. Entrambe le tecniche servono per evitare la ripetizione del codice ma l'ereditarietà affida alle singole classi troppe responsabilità, che mantengono troppi dati di tipo diverso, e riduce solo in parte il problema del codice duplicato. Grazie alla composition possiamo evitare il problema della "code duplication", creando classi molto specifiche, e fornire una notevole flessibilità al paradigma di programmazione.

```
Class Employee(){  
    Person p;  
    Contract c;  
    Commission com;  
}  
/* In questo modo combiniamo in una classe(Employee) altre classi  
specifiche(Person,Contract,Commision) */
```

2.10 Swing Frameworks

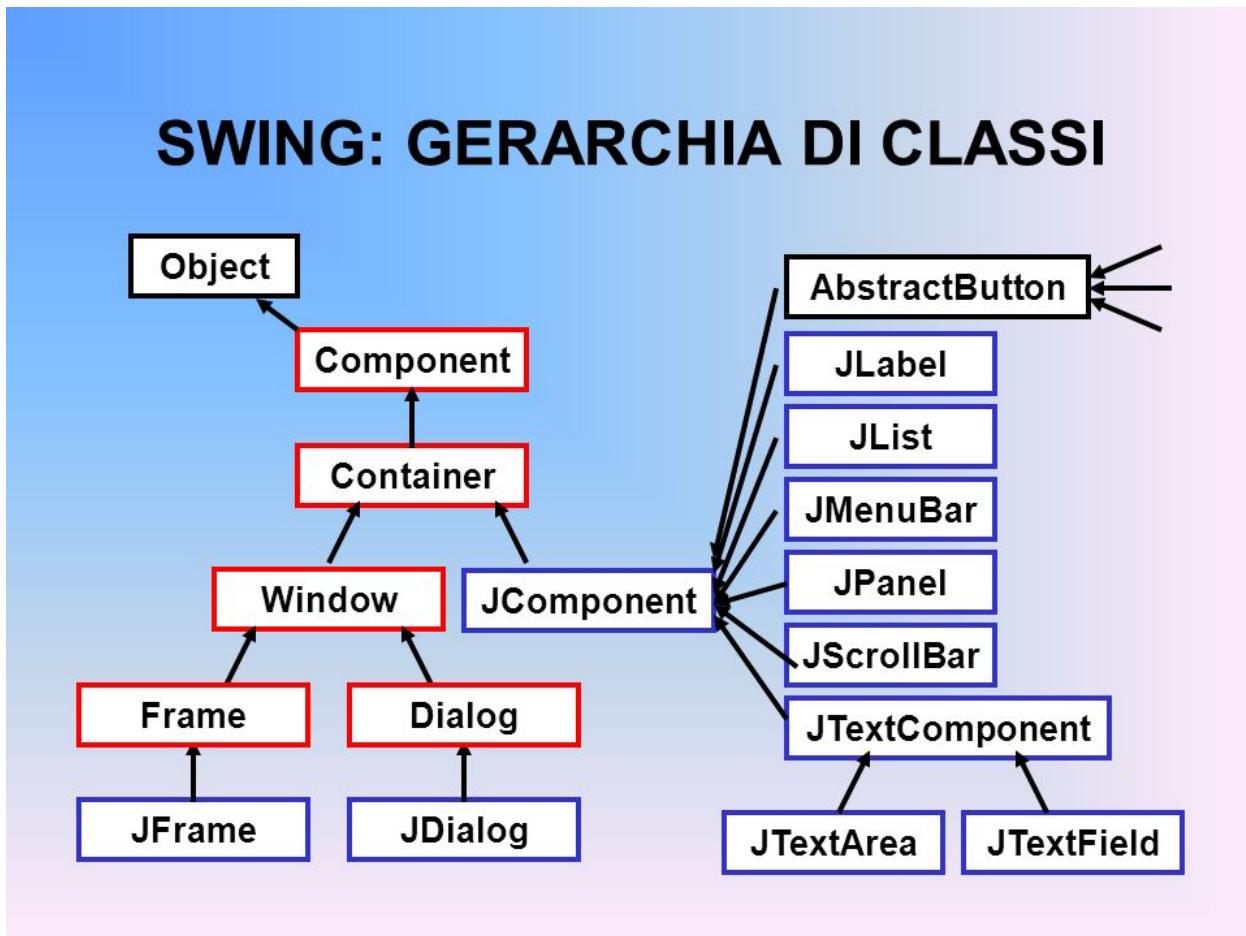
Tutte le applicazioni mobile e tutti i programmi desktop hanno un'interfaccia grafica che permette di comunicare con l'utente finale e di filtrare le informazioni, mostrando solo i dati ritenuti utili.

Durante la progettazione dell'interfaccia grafica dobbiamo tenere sempre in mente che le classi che descrivono l'aspetto dell'applicativo non devono in nessun modo dipendere dalle classi che descrivono le funzionalità dell'applicativo; questo concetto prende il nome di divisione dei domini.

Le principali classi Java che permettono di sviluppare interfacce grafiche sono:

- Java Fx
- Android Interface
- Java Swing, costruito su java.awt.*

Noi ci occuperemo di Java Swing che, come ogni altra classe Java, ha una gerarchia ben definita.

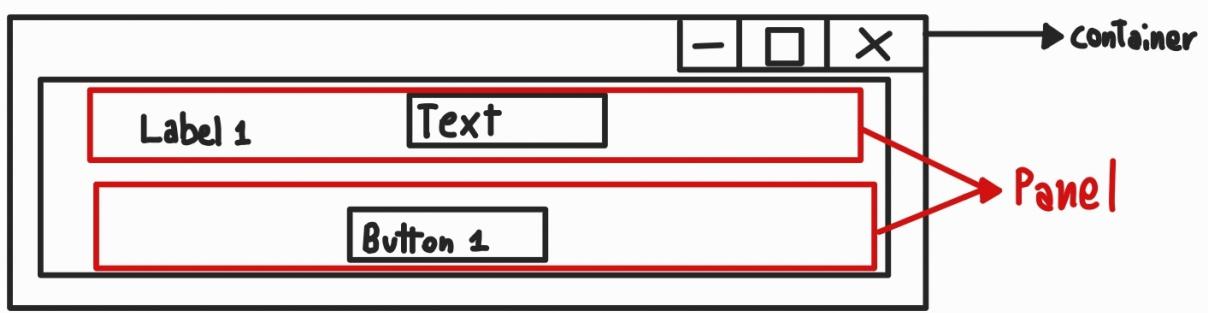


Il processo tipico per programmare un'interfaccia grafica è il seguente:

1. Definire uno o più top-level container
2. Aggiungere pannelli secondari e/o componenti
3. Selezionare un tema look&feel

Ora vediamo come implementare un'interfaccia grafica molto semplice e scarna come quella raffigurata nell'immagine soprastante.

```
// ogni interfaccia deve essere figlia (estendere) JFrame
```



```

public class SumApp extends JFrame{
    // variabili componenti
    JTextField firstNumber;
    JTextField secondNumber;
    JButton plus;
    JTextField result;
    //costruttore
    public SumApp() throws HeadlessException {
        super("Celsius Converter");
        firstNumber = new JTextField("primo numero");
        secondNumber = new JTextField("secondo numero");
        plus = new JButton("+");
        plus.setSize(40,40);
        result = new JTextField("Risultato");
        /*creazione del pannello principale all'interno del costruttore
         questo perche' il pannello principale deve essere un elemento
         non modificabile dall'utente*/
        JPanel mainPanel = new JPanel();
        mainPanel.add(firstNumber);
        mainPanel.add(plus);
        mainPanel.add(secondNumber);
        mainPanel.add(result);
        //blocco finale sempre necessario
        setContentPane(mainPanel); // set pannello principale
        setSize(200,200); //set dimensione finestra
        //definiamo l'operazione di chiusura
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setVisible(true); // rende il pannello visualizzabile
    }
}

```

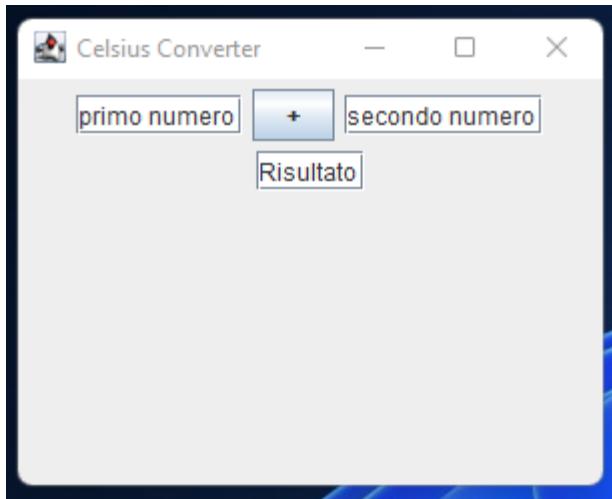
Per eseguire il programma e avviare l'interfaccia dobbiamo utilizzare la SwingUtilities all'interno del main

```

public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            new SumApp();
        }
    });
}

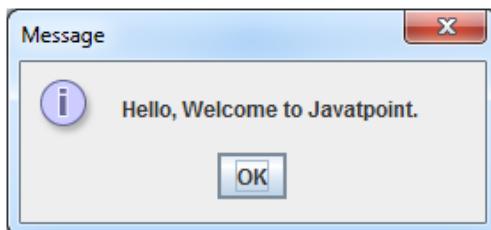
```

Il risultato finale di questa implementazione è il seguente:



Il framework Swign permette anche di generare dialoghi con l'utente, attraverso l'utilizzo della sottoclassse JDialog (extends di JDialog) o utilizzare la classe JOptionPane.

```
JOptionPane.showMessageDialog(frame, "Hello, Welcome to Javatpoint.");
showInputDialog();
showOptionDialog();
```



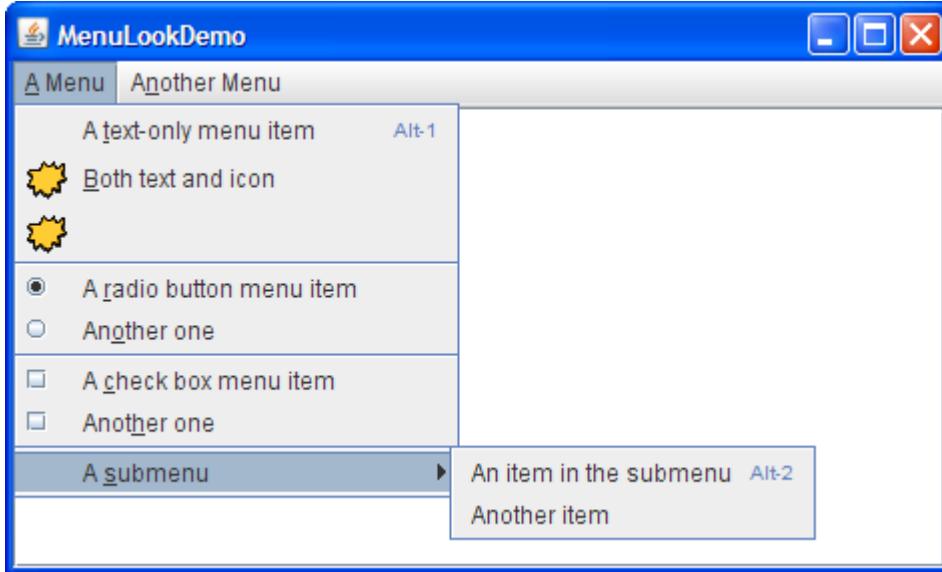
Essendo Java un linguaggio di programmazione multipiattaforma è necessario adattare l'applicativo a qualunque screen size(TV, Desktop, Smartphone,...). Per far ciò possiamo utilizzare dei layout che ci permettono di organizzare l'interfaccia secondo canoni prestabiliti:

- FlowLayout, layout di default
- GridLayout
- GridBagLayout
- BorderLayout, divide il top-panel in 5 spazi(PAGE_START,PAGE_END,...)
- CardLayout

In generale è utile combinare tra loro più layout, utilizzando un layout per il top-panel ed altri layout per i sottopannelli.

Per gestire il programma e le sue opzioni dobbiamo sviluppare un menu con l'utilizzo della sottoclassse JMenu e con l'utilizzo delle componenti JMenuItem, JMenuBar e generando il menu attraverso la funzione setJMenuBar(generateMenu()).

Infine per gestire le funzionalità delle varie componenti di un'interfaccia dobbiamo sviluppare una gestione degli eventi. La gestione degli eventi, in Java, è di tipo listener, cioè viene sviluppata una classe che rimane in attesa fino al verificarsi di un evento E (ActionListener, MouseListener, MouseMotionListener); appena una classe produce l'evento E, entra in azione la classe listener che è chiamata a gestire l'evento. In Java esistono principalmente 2 gestori degli eventi:



- Il frame gestisce gli eventi implementando il listener

```
public class Test extends JFrame implements ActionListener{
    //bisogna implementare la funzione actionPerformed
    ...
    /*attraverso queste funzioni possiamo capire qual e' la classe
     generatrice dell'evento*/
    e.getSource();
    e.getActionCommand();
}
```

- Sono gli oggetti i gestori dei propri eventi

```
//utilizziamo una classe anonima, funzione anonima o lambda
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e){
        ...
        ...
    }
});
```

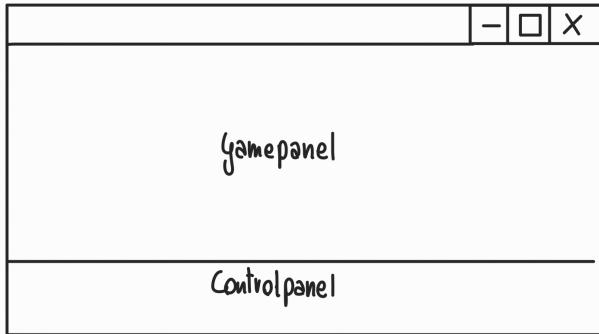
All'interno di IntelliJIdea è possibile generare automaticamente il codice di una determinata interfaccia, semplicemente disegnandola.

2.10.1 GamePanel

Per programmare un videogioco, un programma like paint o un programma a livello grafico dobbiamo sviluppare all'interno della finestra vari custom panel come estensione di JPanel (e non di JFrame).

```
Gamepanel g = new Gamepanel();
Controlpanel cp = new Controlpanel();
```

Questo modo di programmare è rischioso dal punto di vista concettuale perché potremmo andare contro al principio della divisione dei domini, facendo dipendere gli oggetti della scena (personaggi, oggetti in movimento,...) dal pannello che li contiene.



2.11 JDBC

Tutte le informazioni e i dati sensibili che vengono trattati in un applicativo Java devono essere salvati in un luogo sicuro. Ormai, tranne che per scopi didattici, non vengono più utilizzati i file testuali e binari che sono stati sostituiti dal Database, cioè da archivio di dati strutturato in modo da razionalizzare la gestione e l'aggiornamento delle informazioni e da permettere lo svolgimento di ricerche complesse. In base alle prestazioni e alla dimensione dell'applicativo possiamo scegliere tra diversi DB:

- **Networked DBMS**

Comunicano con le applicazioni attraverso le porte logiche ed il protocollo TCP

1. MySQL(3306)
2. MS ServerSQL(1433)

- **Local DB**

Database in localhost, cioè che risiedono sulla propria macchina

1. SQLite, file binario in locale che utilizza la sintassi SQL

Per interfacciarsi con il DBMS utilizziamo il framework JDBC e le sue API. Questo framework è abbastanza particolare perché funziona con tutti i tipi di database e DBMS(SQL,NoSQL,...); questa particolarità deriva dal fatto che le classi all'interno del framework sono perlopiù astratte, quindi molte funzioni non sono state implementate. Per via di questa sua caratteristica il JDBC deve essere integrato con un JDBC Driver che permette la connessione al database ed implementa il protocollo per il trasferimento dei dati. Per integrare ed importare i Driver possiamo scaricarli manualmente (.jar file) e aggiungerli alla CLASSPATH del progetto o utilizzare dei building tool (Gradle) che automatizzano il processo. (All'interno del file build.gradle possiamo vedere i driver importati)

I passaggi per connettersi e comunicare con un DBMS/DB sono i seguenti:

1. Caricare il driver, importato con Gradle o scaricato

```
Class.forName("com.mysql.jdbc.Driver"); //metodo statico
```

2. Stabilire una connessione (viene fatto una sola volta) con comunicazione di rete(URL)

```
DriverManager.getConnection(String Url);
```

3. Creare un statement per ogni operazione da effettuare

```
Statement statement = c.createStatement();
```

4. Eseguire lo statement

```
//operazione di scrittura, ritorna il numero di record modificati
int executeUpdate(String SQL);
```

```

//operazione di lettura, ritorna un oggetto ResultSet
ResultSet executeQuery(String SQL);
ResultSet rs = statement.executeQuery("SELECT * FROM person")
// il ResultSet si comporta come un iteratore
while(rs.next()){ //andiamo alla riga successiva
    rs.getInt(1); //restituisce il valore alla riga corrente colonna 1
    rs.getString(2); //restituisce il valore alla riga corrente
        colonna 2
    rs.getString(3);
    rs.getString(4);
}
statement.executeUpdate(String SQL);

//Formattazione stringhe SQL
String sql = String.format(
"INSERT INTO person (name ,surname ,salary) VALUES ('%s' ,'%s' ,'%f')",
person.getName() ,
person.getSurName() ,
person.getSalary()
);

```

- Chiudere la connessione

```

try{
    ...
}catch(SQLException e){
    //do something
}finally{
    if(connection != null){
        statement.close();
        connection.close();
    }
}

```

Con l'utilizzo dei database e di JDBC esiste un problema di conversione tra i tipi di dato nel database con i tipi di dato di Java. Molte conversioni non sono mappate 1:1, cioè non esiste un tipo di dato in java che rispecchia le caratteristiche del dato sul database. Per esempio il FLOAT viene convertito in double, il VARCHAR in String.

JDBC type	Java type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

2.11.1 ResultSet

Abbiamo parlato di ResultSet come di un oggetto unico standard con la solita utilità di mantenere un cursore che punta alla riga corrente di un set di dati importati dal DB; in verità esistono diversi tipi di ResultSet. Ogni database mette a disposizione diversi ResultSet, impattando notevolmente sul modo di programmare.

- Forward ResultSet(Default)
Permette di "scorrere" il database solo in avanti e non permette di modificare i dati del database con metodi java (bisogna modificare sia i dati java e sia i dati del database con delle query)
- Scrollable ResultSet
Rende il database scrollabile e quindi si può "scorrere" sia in avanti che indietro

```
Statement s = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);

ResultSet rs = s.executeQuery("SELECT * FROM person");

rs.previous();           //vai al precedente
rs.relative(-5);        // vai indietro di 5 rispetto all'elemento corrente
rs.relative(7);          // vai avant di 7 rispetto all'elemento
corrente
```

```

rs.absolute(100); // vai al 100esimo record

• Updatable ResultSet
È possibile modificare il database con metodi java della classe ResultSet

Statement s = c.createStatement(
ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_UPDATABLE);

ResultSet rs = s.executeQuery("SELECT * FROM students");

while (rs.next()) {
    int grade = rs.getInt("grade");
    rs.updateInt("grade", grade + 1); /* modifica il valore sia
        della variabile java grade che del dato sul database*/
    rs.updateRow();
}

```

Un altro fattore che impatta notevolmente sul modo di programmare è la sensibilità alle modifiche:

- ResultSet.TYPE_SCROLL_INSENSITIVE
Di norma utilizzato con i Forward ResultSet, rende il resultSet insensibile alle modifiche del database fatte da applicazioni terze
- ResultSet.TYPE_SCROLL_SENSITIVE
Rende il resultSet sensibile alle modifiche del database (database live)

2.11.2 Transactions

Una transaction è una serie di azioni che vengono eseguite insieme come se fosse un'unica operazione e quindi o vengono eseguite tutte le azioni o nessuna. Il classico esempio di transaction è il trasferimento di denaro da un conto ad un altro: prima dobbiamo togliere 100 euro dal conto1 e poi dobbiamo aggiungere i 100 euro tolti dal conto1 al conto2; se una delle due azioni non va a buon fine il trasferimento non viene eseguito. Da questo esempio è chiaro che dobbiamo eseguire 2 azione come un'unica operazione.

```

// disattiviamo l'autocommit
connection.setAutoCommit(false);
// creiamo ed eseguiamo gli statement
.
.
.
.

// con il commit finale confermo ed abilito tutte le operazioni precedenti
connection.commit();

```

2.12 Functional Interfaces

La programmazione funzionale è un paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche. Con l'ascesa dei big-data e delle architetture multi-core molti linguaggi di programmazione hanno introdotto e supportato la programmazione funzionale. (Haskell is a recente purely functional language). La programmazione funzionale è molto elegante e concisa permettendo di descrivere il comportamento in poche linee di codice. Vediamo un esempio concreto: selezionare gli studenti con una determinata media.

```
public static List<Student> filterStudentsByGrade(List<Student> students,
    double average) {
    List<Student> result = new ArrayList<>();
    for (Student s : students) {
        if (s.getAverage() == average) {
            result.add(s);
        }
    }
    return result;
}
```

In caso volessimo, invece, selezionare gli studenti con una media definita in un range dovremmo sviluppare una nuova funzione, rompendo la DRY o code duplication.

```
public static List<Student> filterStudentsByGradeRange(List<Student>
    students, int low, int high) {
    List<Student> result = new ArrayList<>();
    for (Student s : students) {
        if (s.getAverage() > low && s.getAverage() < high) {
            result.add(s);
        }
    }
    return result;
}
```

Attraverso l'uso di un'interfaccia potremmo rendere il codice più flessibile e più facile da usare.

```
interface StudentPredicate {
    boolean test(Student s);
}
// ogni classe implementera' una strategia diversa
class StudentBadPredicate implements StudentPredicate {
    public boolean test(Student s) {
        return s.getAverage() <= 20;
    }
}

class StudentGoodPredicate implements StudentPredicate {
    public boolean test(Student p) {
        return s.getAverage() >= 26;
    }
}

public static List<Student> filterStudents(List<Student> students,
    StudentPredicate tester) {
    List<Student> result = new ArrayList<>();
    for (Student s : students) {
```

```

        if (tester.test(s)) {
            result.add(s);
        }
    }
    return result;
}

/* main */
goodStudents = filterStudents(students, new StudentGoodPredicate());
badStudents = filterStudents(students, new StudentBadPredicate());

```

Questo approccio però è molto prolioso perché dobbiamo dichiarare molte classi ed istanziare un notevole numero di oggetti. Per rendere il processo elegante e conciso ed utilizzare la programmazione funzionale in Java abbiamo a disposizione 2 espressioni:

1. Anonymous classes

Permettono di dichiarare ed istanziare gli oggetti nello stesso momento. La particolarità è che sono come delle classi locali ma non hanno un nome(anonime) e quindi possono essere utilizzate solo una volta

```

result = filterStudents(l, new StudentPredicate() {
    @Override
    public boolean test(Student p) {
        return p.getAverage() >= 20 && p.getAverage() <= 24;
    }
});

```

2. Lambda expressions

Un'espressione lambda è come un metodo, fornisce un elenco di parametri formali e un corpo (che può essere un'espressione o un blocco di codice). Le espressioni lambda risolvono il problema della verbosità delle classi interne permettendo una riduzione delle linee di codice da scrivere.

```

(parameters) -> expression
(parameters) -> {statements;}

() -> {}
() -> "Raoul"
() -> { return "Mario"; }

//wrong
(Integer i) -> return "Alan" + i
(String s) -> { "Iron Man"; }

//1. A boolean expression
(List<String> list) -> list.isEmpty()

//2. Creating objects
() -> new Apple(10)

//3. Consuming from an object
(Apple a) -> { System.out.println(a.getWeight()); }

//4. Select/extract a field from an object
(String s) -> s.length()

//5. Multiply two ints

```

```

(int a, int b) -> a * b

//6. Compare two objects
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())

```

Le interfacce con uno solo metodo(Comparator, Runnable, Listener,..) sono le candidate all'utilizzo delle lambda expressions per implementare i singoli metodi.

```

//anonymous classe vs lambda expressions
//using anonymous classes
Comparator<Apple> byWeight =
    new Comparator<Apple>() {
        public int compare(Apple a1, Apple a2){
            return a1.getWeight().compareTo(a2.getWeight());
        }
    };

// using lambda expressions:
Comparator<Apple> byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

```

Ritornando al nostro esempio degli studenti

```

interface StudentPredicate {
    boolean test(Student p);
}

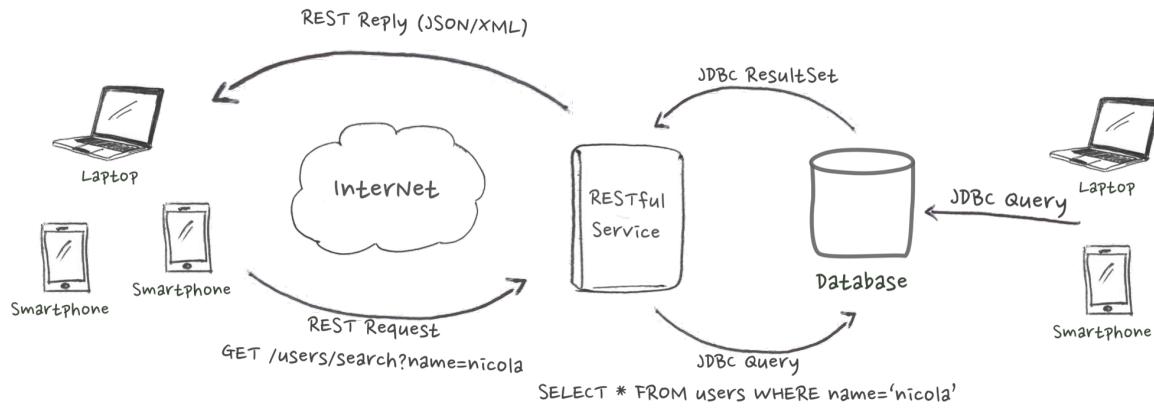
public static List<Student> filterStudents(List<Student> students,
    StudentPredicate tester) {
    List<Student> result = new ArrayList<>();
    for (Student s : students) {
        if (tester.test(s)) {
            result.add(s);
        }
    }
    return result;
}

//main
result = filterStudents(students, (Student s) -> s.getAverage() >= 20 &&
    s.getAverage() <= 24);

```

2.13 REST

I dispositivi degli utenti finali non comunicano direttamente con il database con il framwork JDBC, principalmente per una questione di sicurezza. Per questo motivo è stato introdotto il sistema REST cioè un insieme di servizi web, basati sul protocollo internet HTTP, che permettono di creare un filtro tra l'applicazione finale e i dati sensibili del progetto come struttura,database,dati,ecc.... Quindi attraverso le REST API possiamo disaccoppiare le applicazioni dai dettagli di progettazione e possiamo prevenire esposizioni rischiose del DBMS. Tutto il sistema si basa sulla comunicazione client-server del protocollo HTTP.



- REST Request(HTTP)

<https://financialmodelingprep.com/api/v3/quote/AAPL>
- REST Reply(JSON,XML)

[{"symbol": "AAPL", "name": "Apple Inc.", "price": 276.1000000, "changesPercentage": 2.88000000, "change": 7.73000000, "dayLow": 272.22000000, "dayHigh": 277.85000000, "sharesOutstanding": 4375479808, "timestamp": 1587637985}]

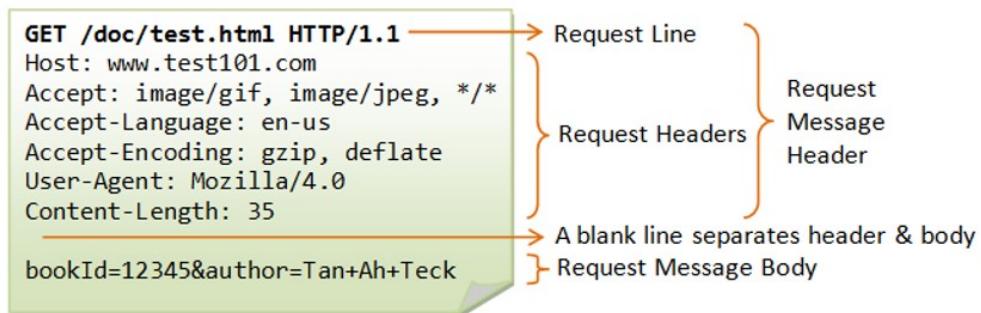
Attraverso la classe Spark di Java possiamo sviluppare un client REST capace di inviare richieste HTTP. All'interno degli applicativi, di norma, viene sviluppata solo la parte del client perché per la parte server, ci serviamo di server di open data sviluppati da terzi. (<https://www.flickr.com/services/api/>)

Il sistema REST si basa su 6 punti fondamentali:

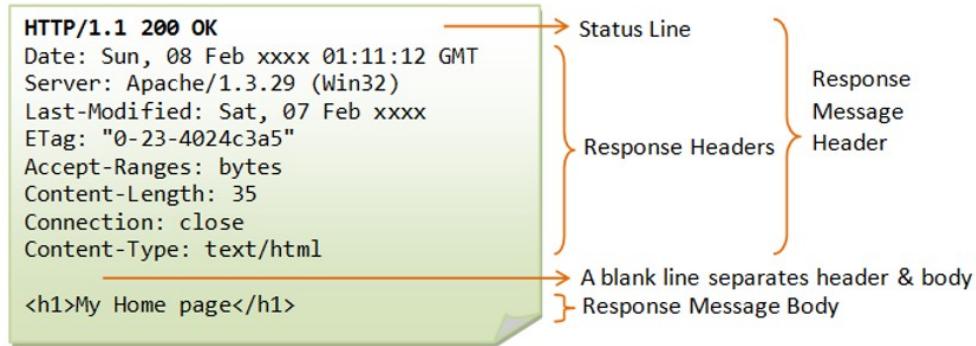
1. Messaggi HTTP

Il client manda una HTTP request al server, il quale risponde con una HTTP response

- HTTP Request



- HTTP Response



N.B. Con URI intendiamo l'indirizzo a cui mandare la richiesta.

Nonostante sia molto utile, il protocollo HTTP è molto inefficiente perché sposta enormi quantità di dati necessari alla sola comunicazione.

2. Risorse

Immagini, Video, Dati, Indirizzi sono tutte risorse web identificate da uno specifico URLs

- Foto https://api.foursquare.com/v2/photos/PHOTO_ID
- Place https://api.foursquare.com/v2/venues/VENUE_ID
- Ricerche <https://api.foursquare.com/v2/users/search>

3. Rappresentazioni delle risorse

Le più utilizzate rappresentazioni sono XML e JSON

- JSON (like Java Object)

```
{
    "ID": "1",
    "Name": "M Vaqqas",
    "Email": "m.vaqqas@gmail.com"
}
```

- XML (like HTML)

```
<Person>

    <ID>1</ID>
    <Name>M Vaqqas </Name>
    <Email> m.vaqqas@gmail.com </Email>

</Person>
```

4. Operazioni HTTP

I verbi HTTP definiscono la specifica operazione.

- **GET** Read a resource (Safe, cioè non modificano la risorsa)
GET /users/145
- **PUT** Insert/update a resource (Idempotente, cioè ha sempre lo stesso effetto indipendentemente dal numero di esecuzioni)
PUT /users/17

- **POST** Insert/update a resource (No safe, No idempotente)

POST /users/ (add a new user)

- **DELETE** Delete a resource (Idempotente)

DELETE /users/145

5. Indirizzo Risorse (URIs)

Il lavoro dell'URI è quello di identificare una risorsa o una collezione di risorse all'interno di una gerarchia.

Protocol://ServiceName/ResourceType/ResourceID Il protocollo stabilisce alcune convenzioni formali per l'URIs:

- Usa nomi plurali per le risorse
- Evita di utilizzare gli spazi
- URI è case insensitive
- Usa la notazione camelCase
- Un nome URI non cambia quasi mai
- Evita verbi(get,post,put)

6. Stateless

I servizi statelessness sono servizi che non tengono traccia dello stato dell'applicazione per tutti i client. Ciò vuol dire una richiesta non dipende da quella precedente e che ogni richiesta è indipendente. Questa caratteristica facilita notevolmente l'uso del sistema REST. Viceversa, i servizi stateful sono troppo complicati da gestire per via dell'enorme numero di client su cui viene eseguita l'applicazione. Per questo motivo la maggior parte dei servizi web sono statelessness che di fatto è diventato uno standard.

- Stateless

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/Persons/2 HTTP/1.1

- Stateful

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/NextPerson HTTP/1.1 (qual è il prossimo cliente???)

N.B. There is no excuse for not documenting your service. You should document every resource and URI for client developers.

Vediamo come sviluppare un semplice client-server.

```
package oop.rest.simple;

import static spark.Spark.get;
import static spark.Spark.port;

public class Server {

    public void run() {
        // Start embedded server at this port
        port(8080);

        // Configure resources
        get("/italian", (request, response) -> "Ciao Mondo!");
        get("/english", (request, response) -> "Hello World!");
        get("/german", (request, response) -> "Hallo Welt!");
    }
}
```

```

    public static void main(String[] args) {
        new Server().run();
    }
}

package oop.rest.simple;

import kong.unirest.Unirest;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Random;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import java.util.random.RandomGenerator;

public class Client implements Runnable {
    static Logger logger = LoggerFactory.getLogger(Client.class);
    RandomGenerator rnd = new Random();
    String[] languages = {"italian", "english", "german"};

    @Override
    public void run() {
        String url = "http://localhost:8080/" +
            languages[rnd.nextInt(languages.length)];
        String json = Unirest.get(url).asString().getBody();
        logger.info(json);
    }
}

public static void main(String[] args) {
    ScheduledExecutorService scheduler =
        Executors.newScheduledThreadPool(1);
    scheduler.scheduleAtFixedRate(new Client(), 0, 5,
        TimeUnit.SECONDS);
}
}

```

2.13.1 Object Mapper

The Jackson ObjectMapper class (com.fasterxml.jackson.databind.ObjectMapper) is the simplest way to parse JSON with Jackson. The Jackson ObjectMapper can parse JSON from a string, stream or file, and create a Java object or object graph representing the parsed JSON. Parsing JSON into Java objects is also referred to as to deserialize Java objects from JSON. The Jackson ObjectMapper can also create JSON from Java objects. Generating JSON from Java objects is also referred to as to serialize Java objects into JSON. The Jackson Object mapper can parse JSON into objects of classes developed by you, or into objects of the built-in JSON tree model explained later in this tutorial.

By the way, the reason it is called ObjectMapper is because it maps JSON into Java Objects (deserialization), or Java Objects into JSON (serialization).

```
ObjectMapper objectMapper = new ObjectMapper();

String carJson =
    "{ \"brand\" : \"Mercedes\", \"doors\" : 5 }";

try {
    Car car = objectMapper.readValue(carJson, Car.class);

    System.out.println("car brand = " + car.getBrand());
    System.out.println("car doors = " + car.getDoors());
} catch (IOException e) {
    e.printStackTrace();
}
```

2.14 Multi-Threading

In un sistema operativo, un processo é l'istanza in esecuzione di un programma. Durante l'esecuzione un processo ha a disposizione una propria area di indirizzamento di dati e codice, un proprio PID e delle risorse a disposizione.

I Thread sono, invece, dei processi leggeri e come tali sono delle entitá computazionali indipendenti cioé ogni thread ha un proprio stack, program counter e variabili locali. La sostanziale differenza tra processo e thread é che i thread, rispetto ai processi, condividono lo stesso spazio di indirizzamento. Questa caratteristica impatta notevolmente sul modo di programmare perché cambia il tipo di comunicazione; infatti i thread possono comunicare solo leggendo e scrivendo sulle variabili e risorse locali in comune che rende la comunicazione rischiosa ma velocissima. In Java, la gestione dei processi e dei thread é affidata nella maggior parte dei casi (dipende dalla specifica Java e dall'implementazione della JVM) allo scheduler del sistema operativo che mappa un thread Java come un thread di sistema. Tutti i programmi Java hanno almeno un thread: il thread del main();

Una domanda che puó sorgere é la seguente: Perché sono stati introdotti i Thread?

Esistono 3 principali motivazioni:

1. Sviluppare un'interfaccia grafica (UI) responsiva, cioè che reagisce ai vari eventi senza bloccarsi (la UI é gestita dal gestore degli eventi). In un ambiente single-thread una azione viene eseguita solo quando quella precedente é terminata; se una singola operazione derivante da un evento richiedesse molto tempo, l'interfaccia grafica rimarrebbe bloccata fino al completamento dell'operazione. In un scenario multi-thread, invece, abbiamo la possibilità di eseguire le varie operazioni in background, senza bloccare il gestore degli eventi e la UI.
2. Eseguire processi in background
3. Sfruttare i vantaggi dei sistemi multiprocessori (parallelismo)

Lo svantaggio principale della multi-programmazione é che é molto difficile per la maggior parte dei programmatori (anche quelli esperti) ed é molto tricky.

Per utilizzare i processi e le primitive di sistema in Java, possiamo utilizzare la classe Process ed i suoi metodi, grazie ai quali possiamo scrivere e leggere su standard error, input ed output.

```
Process p = Runtime.getRuntime().exec("/bin/ls -al /");
Process p = (new ProcessBuilder("/bin/ls", "-al", "/")).start();
```

Invece, per creare un thread abbiamo 3 opzioni a disposizione:

1. Estendere la classe Thread

```
Class T extends Thread {
    public void run() {
        //code here
    }
}

T t = new T();
t.start();
```

2. Implementare l'interfaccia Runnable

```
Class R implements Runnable {
    public void run() {
        //code here
    }
}

Thread t = new Thread(new R());
t.start();
```

3. Classe anonima

```
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
    }
});

class Counter implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName()
                + i);
        }
    }
}

public class Runner{
    public static void main(String[] args) {
        \\\'possiamo anche dare un nome al thread
        Thread t1 = new Thread(new Counter(), "T_A");
        Thread t2 = new Thread(new Counter(), "T_B");
        Thread t3 = new Thread(new Counter(), "T_C");
        t1.start(); t2.start(); t3.start();
    }
}
```

Quando un oggetto Thread viene creato non viene messo automaticamente in esecuzione. Per mandarlo in esecuzione dobbiamo utilizzare il metodo start(), il quale può essere invocato una sola volta sullo stesso thread (altrimenti RuntimeException). A livello di sistema sarà lo scheduler a decidere quando mandare in esecuzione il thread e per quanto tempo (dipende dal tempo in cui la CPU sarà disponibile); lo scheduler della maggior parte delle JVM è time-sliced, cioè ogni thread ha a disposizione la CPU per un determinato lasso di tempo, preemptive, cioè i thread possono essere sospesi prima della fine della loro esecuzione, priority-based, cioè esiste una coda di priorità.

Di default, un thread eredita la priorità del thread che lo ha creato; questa priorità può anche essere configurata.

```
Thread t = new Thread(new Runnable());
t.setPriority(Thread.MAX_PRIORITY); //10
t.setPriority(Thread.MIN_PRIORITY); //1
t.setPriority(Thread.NORM_PRIORITY); //5
```

Se uno scheduler è non-preemptive il primo thread mandato in esecuzione verrà eseguito all'infinito. Possiamo verificare il tipo di scheduler, mandando in esecuzione due thread e notando il comportamento.

```
public class CheckPreemption implements Runnable {
    @Override
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
        }
    }
}

public static void main(String argv[]) {
    CheckPreemption c = new CheckPreemption();
    new Thread(c, "To be").start();
}
```

```

        new Thread(c, "Not to be").start();
    }
}

```

Tutti i thread devono sempre essere "uccisi", cioè dobbiamo terminare la loro esecuzione. Se il thread padre termina, terminano anche tutti i suoi thread figlio. I thread figlio condividono le risorse con il thread padre, comprese le variabili. Al termine del thread padre, i thread figlio non saranno in grado di accedere a quelle risorse che possiede il thread padre.

```

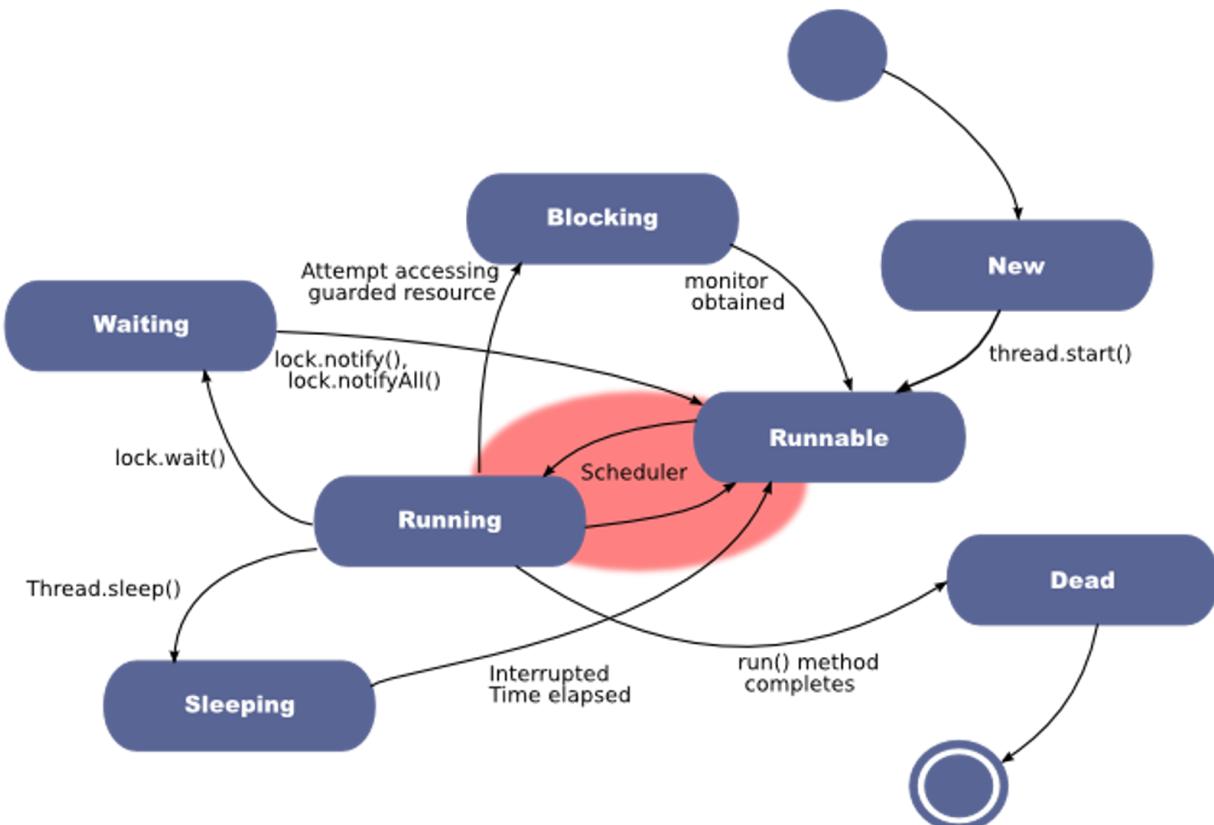
/* start children threads */
producer.start();
consumer.start();

/* wait 1/10 of second */
Thread.sleep(100);

/* gracefully shut down children threads */
producer.running = false; //inizializzata a true
consumer.running = false;

/* wait for children before exit */
producer.join();
consumer.join();

```



La transizione di stati è la seguente:

- Viene creato un nuovo thread (NEW)

- Thread.start() siamo nello stato RUNNABLE, cioè il thread è nella coda dei thread pronti
- Quando viene mandato in esecuzione dallo scheduler siamo nello stato RUNNING
- Un thread può essere interrotto durante l'esecuzione per 4 motivi:
 1. L'esecuzione è finita (stato DEAD)
 2. Il thread ha a disposizione una risorsa ma ha non operazioni da eseguire (WAITING)
 3. Il thread viene bloccato perché ha bisogno di una risorsa non disponibile (BLOCKING)
 4. Il thread viene messo a "riposo" (SLEEPING)

Attraverso dei specifici metodi Java possiamo interrompere esplicitamente l'esecuzione di un thread:

- sleep(), il thread attualmente in esecuzione interrompe l'esecuzione per la durata di sospensione specificata.

```
try {
    // Sleep for 1 second
    Thread.sleep(1000);
} catch (InterruptedException ex) {
    ...
}
```

- yield(), riporta il thread attualmente in esecuzione allo stato Runnable e consente ad altri thread di fare il loro turno.
- join(), il thread attualmente in esecuzione interrompe l'esecuzione fino al completamento del thread a cui si "joina". Solitamente utilizzato dal thread padre per aspettare la fine del thread figli.

```
Thread t = new Thread();
t.start();
t.join();

// per non aspettare figli già terminati e non notificati
t.join(5000);
// wait t for 5 seconds: if t is not finished
// then current thread is Runnable again
```

Tutti questi metodi sono metodi statici della classe Thread (Thread.method()), quindi non hanno effetto sull'istanza t ma hanno effetto sul thread che è attualmente in esecuzione.

2.14.1 Thread Synchronization

Cosa succede quando due diversi thread accedono alla stessa funzione, classe o variabile? Un problema che si verifica ogni volta che due o più thread condividono la stessa risorsa è che un thread "accede" velocemente alla risorsa prima degli altri thread modificando la risorsa e rendendo l'operazione non atomica (che dovrebbe essere atomica). Un problema tipico è quello del prelievo all'ATM; questa operazione unica può essere scomposta in 3 fasi: decidere la somma da prelevare, controllare il conto e se ci sono abbastanza soldi prelevare i soldi. Se 2 o più 3 eseguono questa operazione sullo stesso conto potrebbe succedere la seguente problematica:

Davide decide di prelevare 100\$ e verifica che il conto contenga 125\$!; Luciano entra nello stato RUNNING; Davide decide di prelevare 120\$ e verifica che il conto contenga 125\$!; Luciano ritira 120\$; Davide entra nello stato RUNNING; Davide preleva 100\$ (ha già controllato!) ma il bancomat gli dà solo 5\$

Visto che il programmatore non può controllare lo scheduler di sistema, per risolvere questo problema sono state introdotte delle primitive per garantire che le varie sotto-operazioni del prelievo non vengano divise e rendere l'operazione atomica.

Attraverso la keyword **synchronized** è possibile definire metodi ed oggetti ad accesso esclusivo, utilizzate per proteggere l'accesso alle risorse a cui si accede contemporaneamente.

```
public synchronized void doStuff() {
    System.out.println("synchronized");
}

/* is equivalent to... */

public void doStuff() {
    synchronized(this) {
        System.out.println("synchronized");
    }
}
```

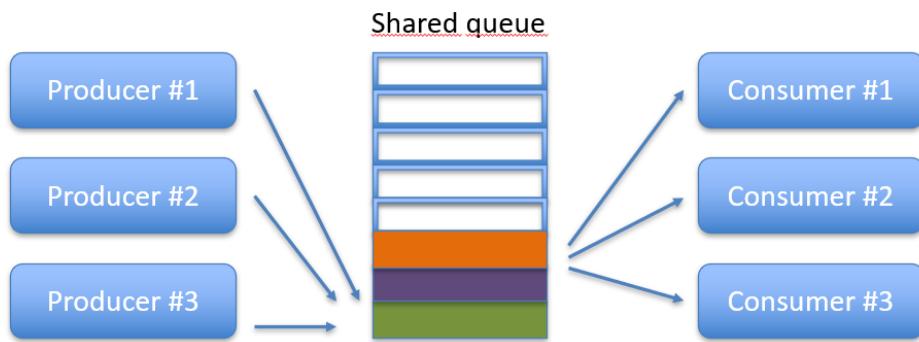
Per far ciò in Java è stato introdotto un metodo di blocco che ogni oggetto Java possiede. Questo crea una differenza notevole tra metodi synchronized statici e non statici:

- Accedere ad un metodo sincronizzato non statico significa ottenere il blocco dell'oggetto. Se un thread ottiene il blocco, tutti gli altri thread devono attendere per accedere TUTTO il codice sincronizzato fino a quando il blocco non viene rilasciato, cioè quando il primo thread esce dal metodo sincronizzato.
- Accedere ad un metodo statico sincronizzato significa ottenere il blocco della classe invece che di un oggetto. Utile quando anche la risorsa condivisa è definita statica.

Una particolarità di Java è che i thread non rilasciano il blocco in caso di sleep() e che un thread può acquisire più di un blocco.

Permane il problema che ogni volta che il blocco dell'oggetto viene acquisito da un thread, gli altri thread possono comunque accedere alla classe con altri metodi non-synchronized e quindi compito del programmatore cerca di sincronizzare tutto il codice. Per sincronizzare il codice possiamo basarci su 2 principali patterns:

- Producer-consumer pattern, dove il thread produttore spinge gli elementi in un oggetto condiviso e il thread consumatore li recupera (consuma). Un tipico esempio è la barra video di Youtube, nella quale il produttore recupera dalla rete i frame e li carica in una coda ed il consumatore legge i frame dalla coda. La problematica principale di questo pattern è che bisogna gestire la coda condivisa da tutti



consumer; per ovviare a questo problema possiamo utilizzare una thread-safe class, cioè una classe che è sicura quando deve essere acquisita da diversi thread, come ConcurrentLinkedQueue. Inoltre utilizziamo il metodo wait() ed il metodo notifyAll() per sincronizzare il codice e non bloccare i thread in attesa:

- wait() può essere chiamato solo da un blocco sincronizzato. Rilascia il blocco sull'oggetto in modo che un altro thread possa entrare e acquisire un blocco.

- Il metodo notify() invia un segnale a uno dei thread che sono in attesa nel pool di attesa dello stesso oggetto senza però poter specificare quale thread in attesa notificare. Il metodo notificationAll() è simile ma invia un segnale a tutti i thread in attesa sull'oggetto.

```

//Consumer
public class ConsumerSynchronizedWaitNotify<T> extends Consumer<T>
{
    public ConsumerSynchronizedWaitNotify(Queue<T> q) {
        super(q);
    }

    @Override
    public void run() {
        while (running) {
            synchronized (q) {
                if (!q.isEmpty()) {
                    q.poll();
                    System.out.printf("Consumer %s received %d
                                      items\n",
                                      Thread.currentThread().getName(), count);
                    count += 1;
                    q.notifyAll();
                } else {
                    try {
                        q.wait();
                    } catch (InterruptedException ignored) {
                    }
                }
            }
        }
    }
}

//Producer
public class ProducerSynchronizedWaitNotify<T> extends Producer<T>{

    public ProducerSynchronizedWaitNotify(int maxitems,T item,Queue<T>
                                         q){
        super(maxitems, item, q);
    }

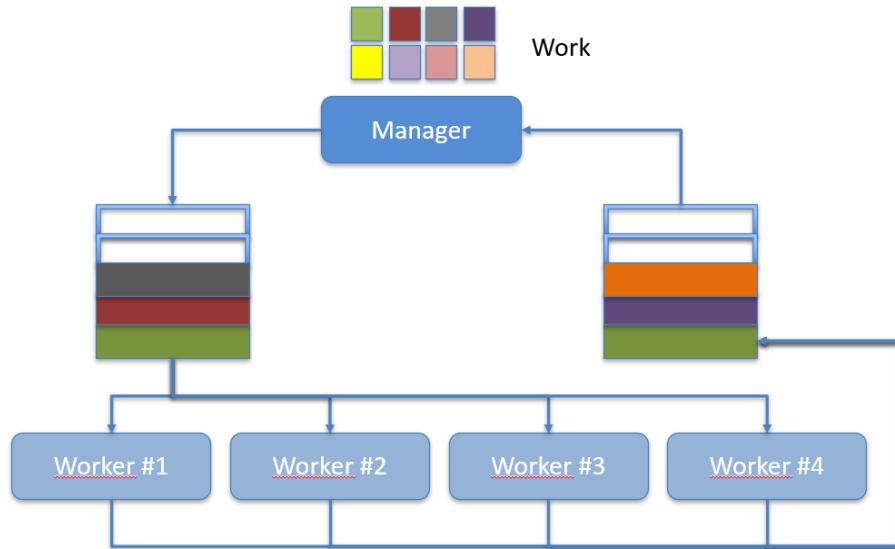
    @Override
    public void run() {
        while (running) {
            synchronized (q) {
                if (q.size() < maxitems) {
                    q.add(item);
                    System.out.printf("Producer %s pushed %d items\n",
                                      Thread.currentThread().getName(), count);
                    count += 1;
                    q.notifyAll();
                } else {

```

```
        try  {
            q.wait();
        } catch (InterruptedException ignored) {

    }
}
}
```

- Manager-worker pattern, dove un manager scomponete un'attività complessa in attività secondarie e le assegna ai thread di lavoro. La problematica principale è gestire i worker che una volta avviati non notificano la loro fine; possiamo risolvere questo problema creando una gestione asincrona ed a chiamate degli eventi dei worcker, grazie alla classe Java propertyChangeSupport.



```
public class Worker {  
    final PropertyChangeSupport support;  
    int start, range;  
    WorkerState state;  
  
    public Worker(int start, int range) {  
        this.start = start;  
        this.range = range;  
        state = WorkerState.NEW;  
        support = new PropertyChangeSupport(this);  
    }  
  
    public void addPropertyChangeListener(PropertyChangeListener l)  
    {  
        support.addPropertyChangeListener(l);  
    }  
  
    public void removePropertyChangeListener(PropertyChangeListener l)  
    {
```

```

        support.removePropertyChangeListener(l);
    }

    public void search() {
        Thread t = new Thread(() -> {
            PrimeSearcher ps = new PrimeSearcherFast();
            List<Integer> primes = new ArrayList<>();
            state = WorkerState.RUNNING;

            for (int i = start; i < (start + range); i++) {
                if (ps.isPrime(i)) {
                    primes.add(i);

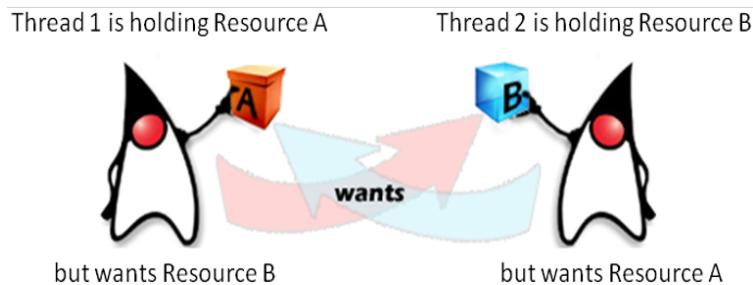
                    //cambia la propieta' e notifichiamo l'evento
                    support.firePropertyChange("progress", null,
                        (100 * (i - start)) / range);
                }
                while (state == WorkerState.PAUSED) {
                    try {
                        Thread.sleep(250);
                    } catch (InterruptedException ignored) {}
                }
            }

            state = WorkerState.COMPLETED;
            support.firePropertyChange("progress", null, 100);
            support.firePropertyChange("results", null, primes);
        });
        t.start();
    }
}

```

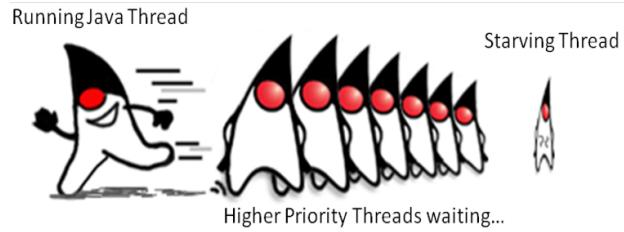
Lavorare con i thread puó risultare molto difficile e ingannevole per 3 problematiche comuni:

- Deadlock, si verifica quando due thread sono bloccati, l'uno con l'altro in attesa del blocco dell'altro. Nessuno dei due può correre finché l'altro non rinuncia al blocco, quindi aspettano per sempre.



- Livelock, si verifica quando un thread agisce in risposta all'azione di un altro thread che é anche essa una risposta all'azione di un altro thread.

- Starvation, descrive una situazione in cui un thread non è in grado di ottenere l'accesso regolare alle risorse condivise e non è in grado di fare progressi.

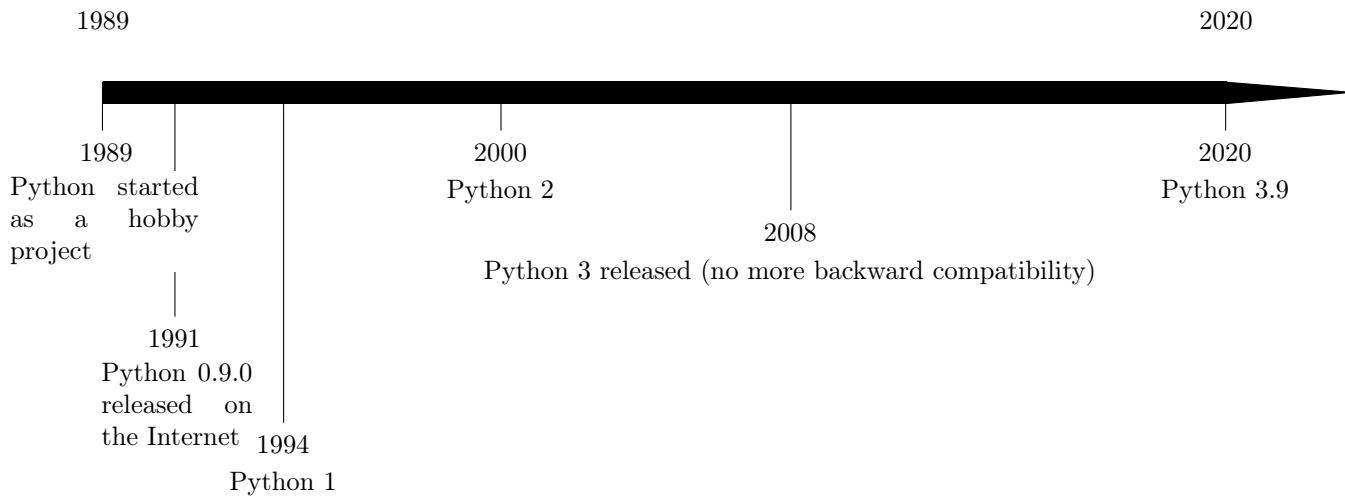


Inoltre i thread sono difficili da debuggare e impattano notevolmente sulle performance.

3 Python Programming

3.1 The Python Environment

3.1.1 Timeline



Guido van Rossum (nato il 31 gennaio 1956) è un programmatore olandese meglio conosciuto come il creatore del linguaggio di programmazione Python, per il quale era il Benevolent Dictator for Life (BDFL) fino a quando non si è dimesso dalla posizione nel Luglio 2018. È rimasto membro del Python Steering Council fino al 2019 e si è ritirato dalle candidature per le elezioni del 2020.

Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered. - Guido van Rossum, August 1996

3.1.2 Advantages and disadvantages

Advantages

- Portable
- User-Friendly (Semplice)
- Open-Source and Community
- Fast Prototyping
- High-level (no need to manage system architecture or memory)
- Interpreted
- Object-Oriented
- Dynamic Typing
Non c'è bisogno di dichiarare i tipi sui riferimenti (Ma i data types esistono)
- Large Standard Library
- Used in Data Science and Web

Disadvantages

- Slow Speed (Lento)
- Not Memory Efficient (Occupava molto spazio in RAM)
- Weak in Mobile Computing (No Android)
- Database Access (way more primitive than JDBC)
- Runtime Errors (dynamically typed languages need more testing)

3.1.3 The Zen of Python

I programmatore Python esperti ti incoraggeranno ad **evitare la complessità** e puntare alla semplicità quando possibile. La filosofia della comunità di Python è contenuta nel “The Zen of Python” di Tim Peters. Puoi accedere a questo insieme di principi per scrivere bene codice inserendo **import this** nel tuo interprete.

[1]: `import this`

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

Prendiamo solo poche righe e vediamo cosa significano per i nuovi programmatore.

Beautiful is better than ugly.

I programmatore Python riconoscono che un buon codice può effettivamente essere bello. Se trovi un modo particolarmente elegante o efficiente per risolvere un problema, specialmente un problema difficile. C'è bellezza anche dentro ad un lavoro tecnico di alto livello.

Explicit is better than implicit.

È meglio essere chiari su quello che stai facendo, piuttosto che inventare un modo più breve per fare qualcosa che è difficile da capire.

Simple is better than complex.

Complex is better than complicated.

Mantieni il tuo codice semplice quando possibile, ma riconosci che a volte puoi affrontare problemi davvero difficili per i quali non ci sono soluzioni facili. In questi casi, accetta la complessità ma evita le complicazioni.

Readability counts.

Ci sono pochissimi programmi interessanti e utili in questi giorni che sono scritti e mantenuti interamente da una persona. Scrivi il tuo codice in modo che gli altri possano leggerlo il più facilmente possibile e in modo che tu sarai in grado di leggerlo e comprenderlo tra 6 mesi. Per far ciò bisogna scrivere dei buoni commenti nel tuo codice.

There should be one-- and preferably only one --obvious way to do it.

Ci sono molti modi per risolvere la maggior parte dei problemi che emergono nella programmazione. Tuttavia, la maggior parte dei problemi ha un approccio standard e consolidato. Salva la complessità per quando è necessario e risovi i problemi, nella maggior parte dei casi, nel modo più diretto possibile.

Now is better than never.

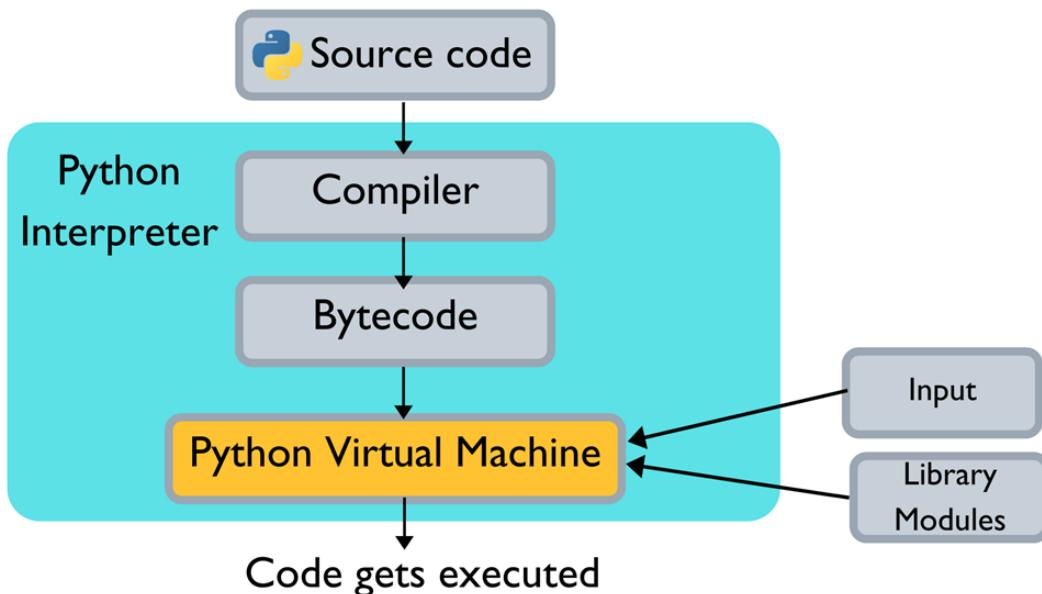
Nessuno scrive mai un codice perfetto. Se hai un'idea che vuoi implementare, scrivi del codice che funzioni. Rilascialo, lascia che sia usato da altri, e poi miglioralo costantemente.

3.1.4 Python Enhancement Proposals

Un PEP (Python Enhancement Proposal) è un documento di progettazione che fornisce informazioni o descrive una nuova funzionalità per Python o descrive i suoi processi ed il suo ambiente. Il PEP dovrebbe fornire una sintesi tecnica della caratteristica e una motivazione per l'introduzione della caratteristica. I principali PEP sono :

- PEP 0 – Index of Python Enhancement Proposals (PEPs)
- PEP 8 – Style Guide
- PEP 257 – Docstring Conventions for Python Code

3.1.5 Python Virtual Machine



Essendo anche Python un linguaggio interpretato la struttura della Virtual Machine è analoga a quella di Java.

3.1.6 Using Python

- The **Python shell** is an interface for typing Python code and executing it directly in your computer's terminal. The **IPython shell** is a much nicer version of the Python shell. It provides syntax highlighting, autocompletion, and other features.
- An IDE is a sophisticated text editor that allows you edit, run, and debug code. The most feature-rich is **PyCharm**. A good alternative is **Visual Studio Code**. Every Python installation comes with an Integrated Development and Learning Environment, which you'll see shortened to IDLE (meglio non usarlo).

- Python scripts can be run from command line.
- The Jupyter Notebook is a powerful tool for prototyping and experimenting with code, as well as visualizing data and writing nicely-formatted text. We will be using this throughout the course.

In all cases aside from Jupyter Notebooks, a python program is a readable script ready for being executed by an interpreter as represented below. Può essere eseguito da riga di comando nel seguente modo:

```
#!/usr/bin/env python
def main():
    print('Hello world!')

if __name__ == "__main__":
    main()

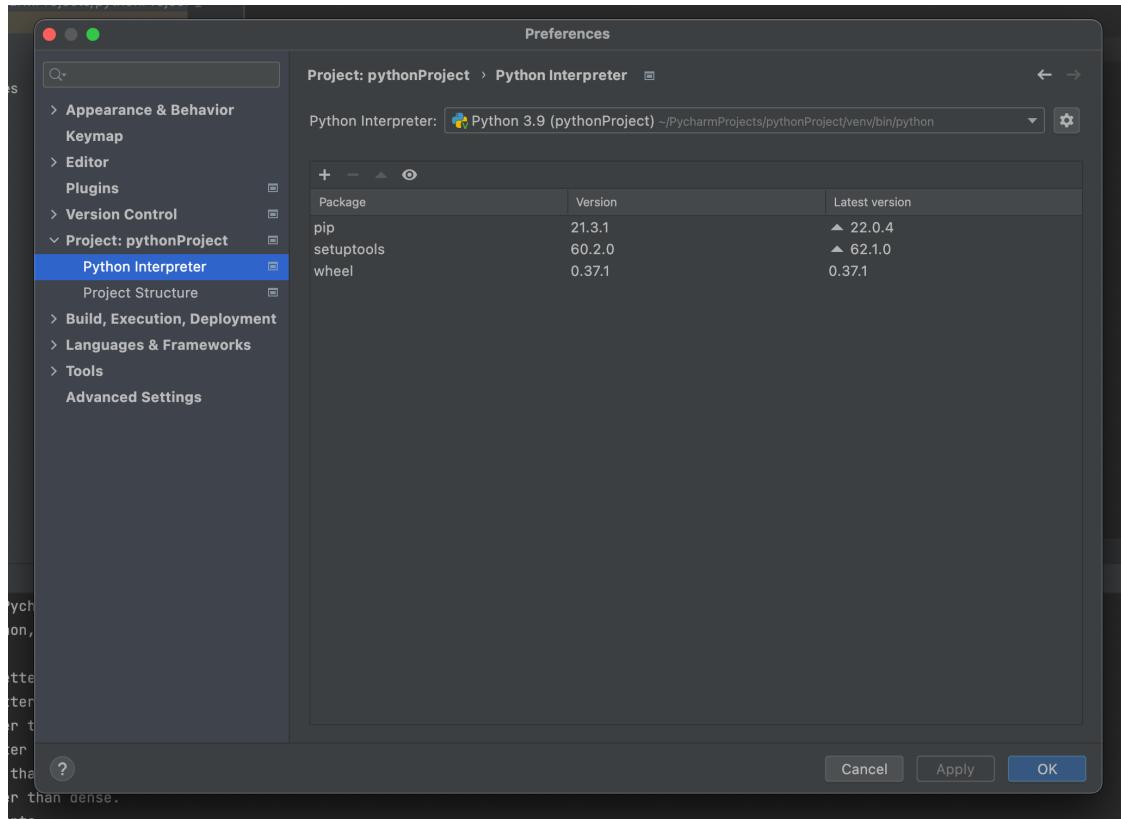
$ python script.py

# or

$ chmod 755 script.py
$ ./script.py
```

3.1.7 Libraries

The **Python Package Index** (aka PyPI) is the official third-party software repository for the Python. **pip** is a command-line program used to {install, remove, update, ...} software packages written in Python. Anaconda and PyCharm greatly simplify the management of external libraries via Virtual Environments, the *conda* package manager and a dedicated GUI shown below.



3.1.8 Which Python version am I running?

```
[121]: import sys  
sys.version
```

```
[121]: '3.9.7 (default, Sep 16 2021, 08:50:36) \n[Clang 10.0.0 ]'
```

3.2 Basic concepts

3.2.1 Main function

Python is not designed to start execution of the code from a main function explicitly. A special variable called `__name__` provides the functionality of the main function. When you run a stand-alone python script which is not referring to any other script, the value of `__name__` variable is equal to `__main__`. Tutti gli script Python vengono eseguiti dalla prima linea di codice interpretata.

```
[3]: print('[1] Hello World!')  
  
def main():  
    print('[2] Hello world!')  
  
if __name__ == '__main__':  
    main()
```

```
[1] Hello World!  
[2] Hello world!
```

3.2.2 Multi-line statements

The end of a statement is marked by a newline character. We can make a statement extend over multiple lines with the line continuation character `\`. Line continuation is implied inside parentheses `()`, brackets `[]`, and braces `{ }`.

```
[4]: a = 1 + 2 + 3 + \  
      4 + 5 + 6 + \  
      7 + 8 + 9  
  
b = (1 + 2 + 3 +  
     4 + 5 + 6 +  
     7 + 8 + 9)  
  
print('a={} b={}'.format(a, b))
```

```
a=45 b=45
```

3.2.3 Indentation

Other languages like C/C++ and Java use curly braces `{ }` to indicate the beginning and the end of blocks of code. Python uses white spaces (space or tabs) to define the block of functions. *It is mandatory to use a consistent amount of spaces (usually 4) for code blocks.* Indentazione obbligatoria!!

```
[5]: def sum(a=0.0, b=0.0):  
    """Sums two numbers.  
  
    Keyword arguments:  
    a -- the first number (default 0.0)
```

```
b -- the second number (default 0.0)
"""
return a + b

sum(4, 5)
```

[5]: 9

3.2.4 Naming rules

- Variables can only contain letters, numbers, and underscores. Variable names can start with a letter or an underscore, but can not start with a number (like C).
- Spaces are not allowed in variable names, so we use underscores instead of spaces. For example, use `student_name` instead of “student name”.
- Variable names should be descriptive, without being too long. For example `motorcycle_wheels` is better than both `wheels`, and `number_of_wheels_on_a_motorcycle`.
- Be careful about using the lowercase letter l and the uppercase letter O in places where they could be confused with the numbers 1 and 0.
- You cannot use Python keywords as variable names.

```
[6]: import keyword
keyword.iskeyword('for')
```

[6]: True

```
[7]: keyword.iskeyword('annalisa')
```

[7]: False

3.2.5 Variable Assignment

Think of a variable as a name attached to a particular object. In Python, variables need not be declared or defined in advance, as is the case in many other programming languages. To create a variable, you just assign it a value and then start using it. Assignment is done with a single equals sign (=).

```
[8]: # one variable, one value
v = 'apple.com'

# same variable, a new value (dynamic typing)
v = 1

# multiple variables, one value
x = y = z = 'same value'

# multiple variables, multiple values
x, y, z = 5, 3.2, 'Hello'
```

3.2.6 Constants

Constants are written in capital letters with underscores separating words. *Constats are only a convention and can be modified.* In fact, there are no actual compiler checks. Constants are usually declared and assigned in a separate module imported from the main file.

```
[9]: MAX_SIZE = 9000  
MAX_SIZE = 9001  
print(MAX_SIZE)
```

```
[9]: 9001
```

As of Python 3.8, there's a `typing.Final` variable annotation that will tell static type checkers (like mypy) that the variable shouldn't be reassigned. This is the closest equivalent to Java's `final`. However, it does not actually prevent reassignment! (rompe la convenzione di Python meglio non utilizzarlo)

```
[10]: from typing import Final  
  
MAX_SIZE: Final = 9000  
MAX_SIZE += 1 # warning reported by static type checker, visible in PyCharm  
print(MAX_SIZE)
```

```
[10]: 9001
```

3.2.7 Everything Is an Object

Python is an object-oriented programming language, so in Python everything is an object. There are no primitive types. Some claim erroneously that Python is a type-free language. But this is not the case! Python has types; however, the types are linked not to the variable names but to the objects themselves. Variable names are only names, references to actual objects.

```
[11]: x = 4  
print(type(x))  
  
x = 3.14159  
print(type(x))  
  
x = 3+4j  
print(type(x))  
  
x = 'hello'  
print(type(x))
```

```
[11]: <class 'int'>  
<class 'float'>  
<class 'complex'>  
<class 'str'>
```

In object-oriented programming languages, an object is an entity that contains data along with associated functionalities. Every entity has data (called attributes) and associated functionalities (called methods). These attributes and methods are accessed via the dot syntax. What is sometimes unexpected is that in Python even simple types have attached attributes and methods.

```
[12]: x = 4+3j  
print('{:.0f}+{}j'.format(x.real, x.imag))  
  
x = 4.5  
print(x.is_integer())  
  
x = 4.0  
print(x.is_integer())
```

```
[12]: 4.0+3.0j
      False
      True
```

3.3 Literals

3.3.1 Numeric literals

Numeric Literals are immutable (unchangeable). Can be only redefined (discarding the old value). Numeric literals can belong to 3 different numerical types: Integer, Float, Complex. The *math* module contains mathematical functions. The *random* module provides functions for random numbers.

sys.maxsize contains the maximum size in bytes a Python int can be. *sys.float_info* contains metadata about floats.

```
[13]: # Decimal, hex, octal, and binary representations of the same integer number
       print(100, 0x64, 0o144, 0b1100100)

       # Float literals without and with an exponent
       print(150.0, 1.5e2)

       # inf is a special Float literal
       # (2^400 is a massive number. 2^85 is close to the number of atoms in the universe)
       print(2e400)
       print(type(2e400))

       # Complex number
       print(3+14j)
```

```
[13]: 100 100 100 100
      150.0 150.0
      inf
      <class 'float'>
      (3+14j)
```

```
[14]: import math
       print(math.fabs(-3))
       print(math.sqrt(2))
       print(math.pi)

       import random
       print(random.random())
```

```
[14]: 3.0
      1.4142135623730951
      3.141592653589793
      0.6162497700409433
```

3.3.2 Boolean literals

Boolean values are the two constant objects *False* and *True*. They are used to represent truth values (other values can also be considered false or true). In numeric contexts, they behave like integers (i.e., *True* = 1, *False* = 0).

```
[15]: x = True  
y = False  
  
print(x)  
print(y)
```

```
[15]: True  
False
```

```
[16]: x = (3 > 5)  
y = (3 != 5)  
  
print(x)  
print(y)
```

```
[16]: False  
True
```

```
[17]: x = True + 0  
y = False + 0  
  
print(x)  
print(y)
```

```
[17]: 1  
0
```

Every value can be evaluated as True or False. The general rule is that any non-zero or non-empty value will evaluate to True. If you are ever unsure, you can open a Python terminal and write two lines to find out if the value you are considering is True or False.

```
[18]: if 3:  
    print('This evaluates to True.')  
else:  
    print('This evaluates to False.')
```

```
[18]: This evaluates to True.
```

```
[19]: if '':  
    print('This evaluates to True.')  
else:  
    print('This evaluates to False.')
```

```
[19]: This evaluates to True.
```

```
[20]: if 'hello':  
    print('This evaluates to True.')  
else:  
    print('This evaluates to False.')
```

```
[20]: This evaluates to True.
```

```
[21]: if None:  
      print('This evaluates to True.')  
    else:  
      print('This evaluates to False.)
```

```
[21]: This evaluates to False.
```

```
[22]: if '':  
      print('This evaluates to True.')  
    else:  
      print('This evaluates to False.)
```

```
[22]: This evaluates to False.
```

```
[23]: if 0:  
      print('This evaluates to True.')  
    else:  
      print('This evaluates to False.)
```

```
[23]: This evaluates to False.
```

3.3.3 String literals

Strings are arrays of bytes representing Unicode characters (16bit encoding). Python does not have a character data type, a single character is a string with a length of 1.

```
[24]: string = 'This is Python'  
string = "C"  
string = \  
"""This is a multiline string  
comprising more than one line of text.  
  
"""
```

3.3.4 Implicit Casting

If a single object is passed to `type()`, the function returns its type. We can use `type()` to know which class a variable belongs to and `isinstance()` to check if a variable belongs to a particular class.

```
[25]: a = 5  
b = 2.3  
c = True  
d = 'hello'  
e = 3+4j  
  
print(type(a))  
print(type(b))  
print(type(c))  
print(type(d))  
print(type(e))  
  
print(isinstance(a, int))  
print(isinstance(b, float))  
print(isinstance(c, bool))
```

```
print(isinstance(d, str))
print(isinstance(e, complex))
```

```
[25]: <class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
<class 'complex'>
True
True
True
True
True
```

In Implicit type conversion, Python automatically converts one type to another type. This process doesn't need any user involvement. Python promotes the conversion of the lower data type (int) to the higher data type (float) to avoid data loss.

```
[26]: print(type(123))
print(type(1.23))
print(type(123 + 1.23))
```

```
[26]: <class 'int'>
<class 'float'>
<class 'float'>
```

3.3.5 Explicit Casting

Molto più semplice rispetto a Java (non esistono wrapper) perché Python mette a disposizione le seguenti funzioni built-in (non c'è bisogno di librerie esterne):

- int() - constructs an integer number from an integer literal, a float literal, or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals
- bool() - constructs a boolean from a numeric literals.

```
[124]: def show(n):
    print(type(n), n)

show(int(1))
show(int(2.8))
show(int('2'))

show(float(1))
show(float('4.2'))
show(float(4.2))

show(str('abc'))
show(str(4.2))
show(str(2))

show(bool('1'))
```

```
show(bool(''))
show(bool(0.2))
```

```
[124]: <class 'int'> 1
<class 'int'> 2
<class 'int'> 2
<class 'float'> 1.0
<class 'float'> 4.2
<class 'float'> 4.2
<class 'str'> abc
<class 'str'> 4.2
<class 'str'> 2
<class 'bool'> True
<class 'bool'> False
<class 'bool'> True
```

Operators, for example `+`, behave in different manners when applied to different types.

```
[126]: a = 4.2
b = 2.4
print(a + b)

a = '4.2'
b = '2.4'
print(a + b)

a = 4.2
b = '2.4'
# produces an error!
# print(a + b)
```

```
[126]: 6.6
4.22.4
```

3.4 Sequences

Sequences are a generic term for an *ordered set* which means that the order in which we input the items will be the same when we access them. Six different types of sequences are supported: strings, lists, tuples, byte sequences, byte arrays, and range objects.

```
[149]: #sequence = [1,2,3,4,5,5]
sequence = 'abcdefghijkl'
```

3.4.1 Operations (for Sequences)

The operator `(+)` is used to concatenate the second element to the first (concatenation).

```
[150]: print(sequence + sequence)
```

```
[150]: abcdefghilabcdefghijkl
```

The operator `(*)` is used to repeat a sequence n number of times (repeat).

```
[151]: print(sequence * 3)
```

```
[151]: abcdefghilabcdefghilabcdefghil
```

Membership operators (`in`) and (`not in`) are used to check whether an item is present in the sequence or not (membership). They return True or False.

```
[152]: print('b' in sequence)
```

```
[152]: True
```

All the sequences in Python can be sliced (slicing). The slicing operator can take out a part of a sequence from the sequence.

```
[153]: print(sequence[1:])
```

```
[153]: bcd
```

3.4.2 Functions (for Sequences)

The `len()` function is very handy when you want to know the length of the sequence.

```
[154]: print(len(sequence))
```

```
[154]: 10
```

The `min()` and `max()` functions are used to get the minimum value and the maximum value from the sequences respectively.

```
[155]: print(min(sequence))
print(max(sequence))
```

```
[155]: a
1
```

The `index()` method searches an element in the sequence and returns the index of the first occurrence.

```
[156]: print(sequence.index('b'))
```

```
[156]: 1
```

The `count()` method counts the number of times an element has occurred in the sequence.

```
[157]: print(sequence.count('b'))
```

```
[157]: 1
```

3.5 Strings

3.5.1 Strings

Strings are a group of characters written inside a single or double-quotes. Python does not have a character type so a single character inside quotes is also considered as a string. Strings are immutable in nature so we can reassign a variable to a new string but we can't make any changes in the string.

```
[38]: print("This is a string.")
print('This is also a string.')
print('I told my friend, "Python is my favorite language!"')
print("The language 'Python' is named after Monty Python, not the snake.")
```

```
[38]: This is a string.  
This is also a string.  
I told my friend, "Python is my favorite language!"  
The language 'Python' is named after Monty Python, not the snake.
```

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. We can only assign different values to the same reference (i.e., the old object is discarded). We cannot delete or remove characters from a string. But deleting the string entirely is possible using the `del` keyword.

```
[39]: name = 'python'  
# name[2] = 'a'  
# TypeError: 'str' object does not support item assignment
```

3.5.2 Accessing characters

Individual characters can be accessed using *indexing*, *negative indexing*, and *slicing*. Index starts both from 0 and -1. Access a character out of index range raises `IndexError`. Using not-integer index raises `TypeError`. Concerning negative indexing, the index of -1 refers to the last item, -2 to the second last item and so on. We can also access a range of items in a string by using the slicing operator :(colon). (Questo vale per tutte le sequenze non solo per le stringhe)

```
[40]: name = 'ooprogramming'  
# indexing  
print(name[0])  
print(name[3])  
  
# negative indexing  
print(name[-1])  
print(name[-2])  
  
# slicing  
print(name[0:3])  
print(name[5:-2])  
print(name[3:])  
print(name[:])
```

```
[40]: o  
r  
g  
n  
oo  
op  
grammi  
rogramming  
ooprogramming
```

3.5.3 Combining Strings

Explicit casting is required when mixing numeric literals and string literals. Alternatively, use string formatting techniques.

```
[160]: age = 26  
# print('Happy ' + age + 'th Birthday!')  
# TypeError: must be str, not int  
print('Happy ' + str(age) + 'th Birthday!')
```

```
print('Happy {}th Birthday!'.format(age))
print(f'Happy {age}th Birthday!')
```

[160]: Happy 26th Birthday!
Happy 26th Birthday!
Happy 26th Birthday!

3.5.4 Formatting Strings

```
[162]: # re-arranging the order of arguments
print('{1} {0}'.format('nicola', 'bicocchi'))

# padding up to 10 spaces
print('{:10} {:10}'.format('nicola', 'bicocchi'))

# specifying 4 digits precision
print('{:.4f} {:.4f}'.format(1 / 3, 2 / 3))

# padding up to 8 spaces, 6 digits precision
print('{:08.6f} {:08.6f}'.format(1 / 3, 2 / 3))

# padding up to 8 spaces, 1 digit precision
print('{:08.1f} {:08.1f}'.format(1 / 3, 2 / 3))
```

[162]: bicocchi nicola
nicola bicocchi
0.3333 0.6667
0.333333 0.666667
000000.3 000000.7

3.5.5 Dealing with whitespaces

```
[43]: name = ' python '
print('\'{}\''.format(name.rstrip()))
print('\'{}\''.format(name.lstrip()))
print('\'{}\''.format(name.strip()))

name = 'python'
print('\'{}\''.format(name.rjust(10)))
print('\'{}\''.format(name.ljust(10)))
print('\'{}\''.format(name.center(10)))
```

[43]: ' python'
'python '
'python'
' python'
'python '
' python '

3.5.6 Dealing with cases

```
[44]: name = 'Ada Lovelace'  
print(name.upper())  
print(name.lower())  
print(name.capitalize())  
print(name.title())  
print(name.islower())  
print(name.isupper())  
print(name.istitle())
```

```
[44]: ADA LOVELACE  
ada lovelace  
Ada lovelace  
Ada Lovelace  
False  
False  
True
```

3.5.7 Finding and replacing substrings

If you want to know where a substring appears in a string, you can use the *find()* method. The *find()* method tells you the index at which the substring begins. Note, however, that this function only returns the index of the first appearance of the substring you are looking for. If the substring appears more than once, you will miss the other substrings.

```
[45]: message = 'I like cats and dogs, but I\'d much rather own a dog.'  
dog_index = message.find('dog')  
print(dog_index)
```

```
[45]: 16
```

If you want to find the last appearance of a substring, you can use the *rfind()* function:

```
[46]: message = 'I like cats and dogs, but I\'d much rather own a dog.'  
last_dog_index = message.rfind('dog')  
print(last_dog_index)
```

```
[46]: 48
```

You can use the *replace()* function to replace any substring with another substring. To use the *replace()* function, give the substring you want to replace, and then the substring you want to replace it with. You also need to store the new string, either in the same string variable or in a new variable.

```
[47]: message = 'I like cats and dogs, but I\'d much rather own a dog.'  
message = message.replace('dog', 'snake')  
print(message)
```

```
[46]: I like cats and snakes, but I'd much rather own a snake.
```

3.5.8 Splitting and joining strings

Strings can be split into a set of substrings when they are separated by a repeated character. If a string consists of a simple sentence, the string can be split based on spaces. The *split()* function returns a list of substrings. The *split()* function takes one argument, the character that separates the parts of the string.

```
[48]: # From string to list
animals = 'dog, cat, tiger, mouse, bear'
print(animals.split(','))
print(animals.split(', '))
```

```
[48]: ['dog', ' cat', ' tiger', ' mouse', ' bear']
['dog', 'cat', 'tiger', 'mouse', 'bear']
```

```
[49]: # From list to string
# Don't do this!
animals = ['dog', 'cat', 'tiger', 'mouse', 'bear']
semicolon_separated = animals[0]
for animal in animals[1:]:
    semicolon_separated += ', ' + animal
print(semicolon_separated)
```

```
[49]: dog, cat, tiger, mouse, bear
```

```
[164]: # From list to string
# Do this!
animals = ['dog', 'cat', 'tiger', 'mouse', 'bear']
print(', '.join(animals))
```

```
[164]: dog,cat,tiger,mouse,bear
```

3.6 Flow control

3.6.1 if .. else

The if..else statement evaluates a boolean condition. If the condition is True, the body of if is executed. If the condition is False, the body of else is executed. Mandatory indentation is used to separate the blocks.

```
[51]: n = 15
if n > 0:
    print('{} > 0'.format(n))
else:
    print('{} <= 0'.format(n))
```

```
[51]: 15 > 0
```

3.6.2 if .. elif .. else

The elif is short for else if. It allows us to check for multiple conditions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, the body of else is executed. Only one block among the several if...elif...else blocks is executed according to the condition. The if block can have only one else block. But it can have multiple elif blocks.

```
[52]: n = 5
if n > 0:
    print('{} > 0'.format(n))
elif n < 0:
    print('{} < 0'.format(n))
else:
    print('{} == 0'.format(n))
```

```
[52]: 5 > 0
```

3.6.3 Comparison operators

Every if statement evaluates to *True* or *False*. *True* and *False* are Python keywords, which have special meanings attached to them. You can test a number of conditions in your if statements. The most frequently used are listed below. There is a [section of PEP 8](#) that tells us it's a good idea to put a single space on either side of all of these comparison operators.

```
[165]: a = (2,3,4)
b = (2,3,4)

# == compares objects content
print(a == b)

# is compares references (objects identity)
print(a is b)
```

```
[165]: True
False
```

```
[168]: # Some immutable objects (str, int, ...) are transparently optimized (like Strings in
        ↪Java)
a = 4.2
b = 4.2

# == compares objects content
print(a == b)

# is compares references (objects identity)
print(a is b)
```

```
[168]: True
False
```

```
[55]: 5 == 4
```

```
[55]: False
```

```
[56]: 5 == 5.0
```

```
[56]: True
```

```
[57]: 'Eric'.lower() == 'eric'.lower()
```

```
[57]: True
```

```
[58]: '5' == str(5)
```

```
[58]: True
```

```
[59]: 3 != 5
```

```
[59]: True
[60]: 'Eric' != 'eric'
[60]: True
[61]: 5 > 3
[61]: True
[62]: 3 >= 3
[62]: True
[63]: 3 < 5
[63]: True
[64]: 3 <= 5
[64]: True
[65]: vowels = 'aeiou'
      'a' in vowels
[65]: True
[66]: vowels = ['a', 'e', 'i', 'o', 'u']
      'a' in vowels
[66]: True
```

3.6.4 Logical operators

```
[67]: x = 11

# and
if x > 5 and x > 10:
    print('{} > 5 and {} > 10'.format(x, x))

# or
if x < 5 or x > 10:
    print('{} < 5 or {} > 10'.format(x, x))

# not
if not x > 10:
    print('not {} > 10'.format(x))
```

[67]: 11 > 5 and 11 > 10
 11 < 5 or 11 > 10

3.6.5 for loop

The for loop in Python is used to iterate over sequences or other iterable objects (e.g., bytearrays, buffers). Iterating over a sequence is called traversal. *char* is the variable that takes the value of each item inside the sequence on each iteration. The iteration continues until the end of the sequence is reached. The body of for loop is separated from the rest of the code using indentation.

```
[172]: string = 'python'

for c in string:
    print(c)
```

```
[172]: p
y
t
h
o
n
```

A for loop can have an optional *else* block as well. The else part is executed when the loop terminates.

```
[69]: string = 'python'

for char in string:
    print(char)
else:
    print('terminated')
```

```
[69]: p
y
t
h
o
n
terminated
```

The *break* keyword can be used to stop a for loop. In such cases, the else part is ignored. Control of the program flows to the statement immediately after the body of the loop. If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

```
[173]: string = 'python'

for char in string:
    if char == 'h':
        break
    print(char)
else:
    print('for terminated')
```

```
[173]: p
y
```

The *continue* statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

```
[71]: string = 'python'

for char in string:
    if char == 'h':
        continue
    print(char)
else:
    print('for terminated')
```

```
[71]: p
y
t
o
n
for terminated
```

3.6.6 while loop

The while loop is used to iterate as long as the test expression (condition) is true. Generally used when the number of times to iterate is unknown beforehand.

```
[72]: i = 0
n = 10
sum = 0

while i <= n:
    sum = sum + i
    i += 1

print('sum={}'.format(sum))
```

```
[72]: sum=55
```

While loops can also have an optional else block. The else part is executed when the loop terminates. The while loop can be terminated with a break statement. In such cases, the else part is ignored.

```
[73]: i = 0
n = 10
sum = 0

while i <= n:
    sum = sum + i
    i += 1
else:
    print('sum={}'.format(sum))
```

```
[73]: sum=55
```

3.6.7 pass

The pass statement is a null statement. The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when the pass is executed. It results in no operation (NOP). (Serve per dare una momentanea implementazione a funzioni/oggetti che dovranno essere sviluppati in futuro)

```
[176]: for val in 'python':
    pass

def function(args):
    pass

class Example:
    pass
```

3.7 Functions

3.7.1 General Syntax

Functions much improve code reuse. Functions, in fact, can be used and reused. A general function looks something like this:

```
[75]: def function_name(arg_1, arg_2):
    pass

function_name('a', 'b')
```

3.7.2 Passing parameters

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. [Python Tutor](#) could be of much help for understanding how these examples work.

```
[76]: def change_list(numbers):
    numbers.extend([4, 5, 6])
    return

numbers = [1, 2, 3]
print(numbers)
change_list(numbers)
print(numbers)
```

```
[76]: [1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

An example where argument is being passed by reference and the reference is being overwritten inside the called function (i.e., the reference COPY is overwritten). The parameter numbers is local to the function. Changing numbers within the function does not affect the caller.

```
[77]: def change_list(numbers):
    numbers = [4, 5, 6]
    return

numbers = [1, 2, 3]
print(numbers)
change_list(numbers)
print(numbers)
```

```
[77]: [1, 2, 3]
[1, 2, 3]
```

3.7.3 Default Arguments

Function arguments can have default values. We can provide a default value to an argument by using the assignment operator (`=`). Any number of arguments in a function can have a default value. Once we have a default argument, all the arguments to its right must also have default values.

```
[178]: def greet(name, msg='Good morning!'):
    print('Hello {}, {}'.format(name, msg))

greet('Bruce', 'How are you doing?')
greet('Kate')
```

```
[178]: Hello Bruce, How are you doing?
Hello Kate, Good morning!
```

3.7.4 Keyword Arguments

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. We can mix positional arguments with keyword arguments during a function call. We must keep in mind that *keyword arguments must follow positional arguments*.

```
[180]: # 2 keyword arguments (in order)
greet(name = 'Bruce', msg = 'How do you do?')

# 2 keyword arguments (out of order)
greet(msg = 'How do you do?', name = 'Bruce')

# 1 positional, 1 keyword argument
greet('Bruce', msg = 'How do you do?')

# greet(name='Bruce', 'How do you do?')
# SyntaxError: positional argument follows keyword argument
```

```
[180]: Hello Bruce, How do you do?
Hello Bruce, How do you do?
Hello Bruce, How do you do?
```

L'uso di keyowrd posizionali e valori di default è tipico delle funzioni complesse usate nella data science, in cui abbiamo un parametro obbligatorio ed una serie di argomenti facoltativi.

```
[181]: def generate_chart(data, lines=None, chart_type=None, borders=None, shadows=None):
    print('{} lines={}, chart_type={}, borders={}, shadows={}'.format(
        data, lines, chart_type, borders, shadows))

generate_chart([1,2,3])
generate_chart([1,2,3], lines=3)
generate_chart([1,2,3], lines=3, shadows=4)
generate_chart([1,2,3], shadows=4, lines=2)
```

```
[181]: [1, 2, 3] lines=None, chart_type=None, borders=None, shadows=None
[1, 2, 3] lines=3, chart_type=None, borders=None, shadows=None
[1, 2, 3] lines=3, chart_type=None, borders=None, shadows=4
[1, 2, 3] lines=2, chart_type=None, borders=None, shadows=4
```

3.7.5 Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments. In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a for loop to retrieve all the arguments back.

```
[183]: def greet(*args):
    # names is a tuple
    for arg in args:
        print('Hello', arg)

if __name__ == '__main__':
    greet('Monica', 'Luke', 'Steve', 'John', 'Sven')
```

```
[183]: Hello Monica
Hello Luke
Hello Steve
Hello John
Hello Sven
```

```
[82]: def adder(num_1, num_2, *args):
    sum = num_1 + num_2

    for arg in args:
        sum = sum + arg
    return sum

print(adder(1, 2))
print(adder(1, 2, 3, 4, 5))
```

```
[183]: 3
15
```

3.7.6 Arbitrary keyword arguments

Python also provides a syntax for accepting an arbitrary number of keyword arguments. The syntax looks like below. The third argument has two asterisks in front of it, which tells Python to collect all remaining key-value arguments in the calling statement. This argument is commonly named *kwargs*. La funzione **items** trasforma il dizionario in una lista di tuple. We see in the output that these key-values are stored in a dictionary. We can loop through this dictionary to work with all of the values that are passed into the function:

```
[185]: def example_function(arg_1, arg_2, **kwargs):
    print(arg_1)
    print(arg_2)
    for key, value in kwargs.items():
        print('{} = {}'.format(key, value))

#example_function('a', 'b')
example_function('a', 'b', kwarg_1='abc', kwarg_2='def')
```

```
[185]: a  
b  
kward_1 = abc  
kward_2 = def
```

Funzione più generica possibile in Python (prende un numero di parametri arbitrario senza nome ed un numero di parametri arbitrario con nome):

```
[186]: def example_function(*args, **kwargs):  
    for arg in args:  
        print(arg)  
    for key, value in kwargs.items():  
        print('{} = {}'.format(key, value))  
  
example_function()  
example_function('a', 'b', 'c')  
example_function('a', 'b', 'c', kward_1='abc', kward_2='def', kward_3='ghi')  
example_function(kward_1='abc', kward_2='def', kward_3='ghi')
```

```
[186]: a  
b  
c  
a  
b  
c  
kward_1 = abc  
kward_2 = def  
kward_3 = ghi  
kward_1 = abc  
kward_2 = def  
kward_3 = ghi
```

3.7.7 Lambda expressions

Small anonymous functions can be created with the *lambda* keyword. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition.

```
[85]: import math  
  
def sqrt(x):  
    return math.sqrt(x)  
  
def log(x):  
    return math.log(x)  
  
def process(items, function):  
    for item in items:  
        print(item, function(item))  
  
process([1,2,3], sqrt)  
process([1,2,3], lambda x : math.sqrt(x))
```

```
[85]: 1 1.0
2 1.4142135623730951
3 1.7320508075688772
1 1.0
2 1.4142135623730951
3 1.7320508075688772
```

```
[86]: # one argument
f = lambda x: x + 1
f(2)
```

```
[86]: 3
```

```
[87]: # two arguments
f = lambda x, y: x + y
f(2, 3)
```

```
[87]: 5
```

```
[88]: # direct call
(lambda x, y: x + y)(2, 3)
```

```
[88]: 5
```

```
[89]: f = lambda first, last: 'Full name: {} {}'.format(first.title(), last.title())
f('anna', 'pannoccchia')
```

```
[89]: 'Full name: Anna Pannoccchia'
```

Funzione lambda che stabilisce il metodo di ordinamento delle liste di tuple.

```
[90]: pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[0])
print(pairs)
pairs.sort(key=lambda pair: pair[1])
print(pairs)
```

```
[90]: [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

3.7.8 Returning multiple values

Python allows various ways for returning multiple values.

```
[91]: # using a tuple
def g(x):
    y0 = x + 2
    y1 = x * 3
    y2 = y0 - y1
    return (y0, y1, y2)

g(3)
```

```
[91]: (5, 9, -4)
```

```
[92]: # using a dictionary
def g(x):
    y0 = x + 2
    y1 = x * 3
    y2 = y0 - y1
    return {'y0': y0, 'y1': y1, 'y2': y2}

g(3)
```

```
[92]: {'y0': 5, 'y1': 9, 'y2': -4}
```

```
[189]: # using a class
class ReturnValue:
    def __init__(self, y0, y1, y2):
        self.y0 = y0
        self.y1 = y1
        self.y2 = y2

    #def __repr__(self):
    #    return str(self.__dict__)

def g(x):
    y0 = x + 2
    y1 = x * 3
    y2 = y0 - y1
    return ReturnValue(y0, y1, y2)

r = g(3)
print(r)
```

```
[189]: <__main__.ReturnValue object at 0x7fb608f18190>
```

```
[191]: # using dataclass (only 3.7+)
# check https://www.youtube.com/watch?v=uRVVyl9uaZc for more details
from dataclasses import dataclass

@dataclass
class ReturnValue:
    y0: int
    y1: int
    y2: int

def g(x):
    y0 = x + 2
    y1 = x * 3
    y2 = y0 - y1
    return ReturnValue(y0, y1, y2)

r = g(3)
print(r)
print('{} {} {}'.format(r.y0, r.y1, r.y2))
```

```
[191]: ReturnValue(y0=5, y1=9, y2=-4)
      5 9 -4
```

3.8 Decorators

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure. Decorators are usually called before the definition of a function you want to decorate.

3.8.1 Inner Functions

It is possible to define functions inside other functions. Such functions are called inner functions.

```
[95]: def parent():
    def first_child():
        print('first_child() function')

    def second_child():
        print('second_child() function')

    print('parent() function')
    first_child()
    second_child()

parent()
```

```
[95]: parent() function
       first_child() function
       second_child() function
```

Returning Functions From Functions

Functions are objects and can be used as return values.

```
[96]: def parent(n):
    def first_child():
        return 'Hi, I am Emma'

    def second_child():
        return 'Hi, I am Liam'

    if n == 1:
        return first_child
    else:
        return second_child

# reference for a function object
f = parent(1)
print(f())

# direct function call
print(parent(2)())
```

```
[96]: Hi, I am Emma
       Hi, I am Liam
```

3.8.2 Decorators as wrappers

```
[2]: def my_decorator(func):
    def wrapper():
        print('Something is happening before the function is called.')
        func()
        print('Something is happening after the function is called.')
    return wrapper

def say_whee():
    print('Whee!')

#say_whee()

f = my_decorator(say_whee)
f()

# more concise
#my_decorator(say_whee)()
```

```
[2]: Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

3.8.3 Pie (@) syntax

The way we decorated `say_whee()` above is a little clunky. Python allows to use decorators in a simpler way with the @ symbol, sometimes called the *pie syntax*. The following example does the exact same thing as the first decorator example:

```
[3]: def my_decorator(func):
    def wrapper():
        print('Something is happening before the function is called.')
        func()
        print('Something is happening after the function is called.')
    return wrapper

@my_decorator
def say_whee():
    print('Whee!')

say_whee()
```

```
[3]: Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

3.8.4 Passing arguments

```
[99]: def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

```

@do_twice
def say(name):
    print(name)

# say('ciao')
# TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given

```

The solution is to use `args` and `*kwargs` in the inner wrapper function. Then it will accept an arbitrary number of positional and keyword arguments.

```

[100]: def do_twice(func):
        def wrapper_do_twice(*args, **kwargs):
            func(*args, **kwargs)
            func(*args, **kwargs)
        return wrapper_do_twice

@do_twice
def say(name, surname):
    print("Hello! " + name + " " + surname)

say('Nicola', 'Bicocchi')

```

```
[100]: Hello! Nicola Bicocchi
Hello! Nicola Bicocchi
```

3.8.5 Returning arguments

```

[101]: def do_twice(func):
        def wrapper_do_twice(*args, **kwargs):
            func(*args, **kwargs)
            func(*args, **kwargs)
        return wrapper_do_twice

@do_twice
def say(name):
    print('Creating greeting')
    return '{}!!'.format(name)

print(say('hello'))

```

```
[101]: Creating greeting
Creating greeting
None
```

To fix this, you need to make sure the wrapper function returns the return value of the decorated function.

```

[102]: def do_twice(func):
        def wrapper_do_twice(*args, **kwargs):
            func(*args, **kwargs)
            return func(*args, **kwargs)
        return wrapper_do_twice

@do_twice

```

```

def say(name):
    print('Creating greeting')
    return name + '!'

print(say('hello'))

```

[101]: Creating greeting
Creating greeting
hello!

3.9 Exceptions

3.9.1 Definition

When something goes wrong an exception is raised. For example, if you try to divide by zero, `ZeroDivisionError` is raised or if you try to access a nonexistent key in a dictionary, `KeyError` is raised. In Python non c'è la differenza tra eccezioni check e uncheck, tutte le eccezioni di Python sono uncheck e quindi non vengono viste durante l'interpretazione del codice (a compile time).

[7]: empty_dict = {}

Uncomment to see the traceback
empty_dict['key']

3.9.2 try - except

If you know that a block of code can fail in some manner, you can use `try-except` structure to handle potential exceptions in a desired way.

[8]: # Let's try to open a file that does not exist
file_name = 'not_existing.txt'

try:
 with open(file_name, 'r') as my_file:
 print('File is successfully open')

except FileNotFoundError as e:
 print(e)

[8]: [Errno 2] No such file or directory: 'not_existing.txt'

If you don't know the type of exceptions that a code block can possibly raise, you can use `Exception` which catches all exceptions. In addition, you can have multiple `except` statements.

[12]: def calculate_division(var1, var2):
 result = 0.0
 try:
 result = var1 / var2
 except ZeroDivisionError as e:
 print(e)
 except TypeError as e:
 print(e)
 return result

print(calculate_division(3, 0))

```
print(calculate_division(3, '0'))
```

```
[12]: division by zero
0.0
unsupported operand type(s) for /: 'int' and 'str'
0.0
```

3.9.3 Delegation

try-except can be also in outer scope:

```
[106]: def calculate_division(var1, var2):
         return var1 / var2

     try:
         calculate_division(3, 0)
     except ZeroDivisionError as e:
         print(e)
     except TypeError as e:
         print(e)
```

```
[106]: division by zero
```

```
[13]: def calculate_division(var1, var2):
        return var1 / var2

    def process(var1, var2):
        # other computations
        return calculate_division(var1, var2)

    try:
        process(3, 0)
    except ZeroDivisionError as e:
        print(e)
    except TypeError as e:
        print(e)
```

```
[13]: division by zero
```

3.9.4 Raising exceptions

We can use the `raise` keyword to throw an exception if a condition occurs. The statement can be complemented with a custom exception. Using standard exceptions is nevertheless preferred. Refer to <https://docs.python.org/3/library/exceptions.html> for the full taxonomy of exceptions.

```
[108]: def calculate_division(var1, var2):
         result = 0.0

         try:
             result = var1 / var2
         except ZeroDivisionError:
             raise ValueError('Zero-division error')
         except TypeError:
             raise ValueError('Type error')
```

```

        return result

try:
    calculate_division(2, '3')
except ValueError as e:
    print(e)

```

[108]: Type error

3.9.5 try - except - else

The optional *else* clause is executed if and when control flows off the end of the try clause. Control *flows off the end* except in the case of an exception or the execution of a return, continue, or break statement.

```
[109]: def calculate_division(var1, var2):
        return var1 / var2
```

```
# Don't do this!
exception_occurred = False
try:
    calculate_division(1, 0)
except ZeroDivisionError:
    exception_occurred = True
except TypeError:
    exception_occurred = True

if not exception_occurred:
    print('All went well!')
else:
    print('Something happened!')
```

[110]: Something happened!

```
# Do this!
try:
    calculate_division(1, 0)
except ZeroDivisionError:
    print('Something happened!')
except TypeError:
    print('Something happened!')
else:
    print('All went well!')
```

[111]: Something happened!

3.9.6 try - except - finally

For scenarios where you want to do something always, even when there are exceptions. A *finally* clause is always executed before leaving the try statement, whether an exception has occurred or not. When an exception has occurred in the try clause and has not been handled by an except clause (or it has occurred in a except or else clause), it is re-raised after the finally clause has been executed. The finally clause is also executed *on the way out* when any other clause of the try statement is left via a break, continue or return statement.

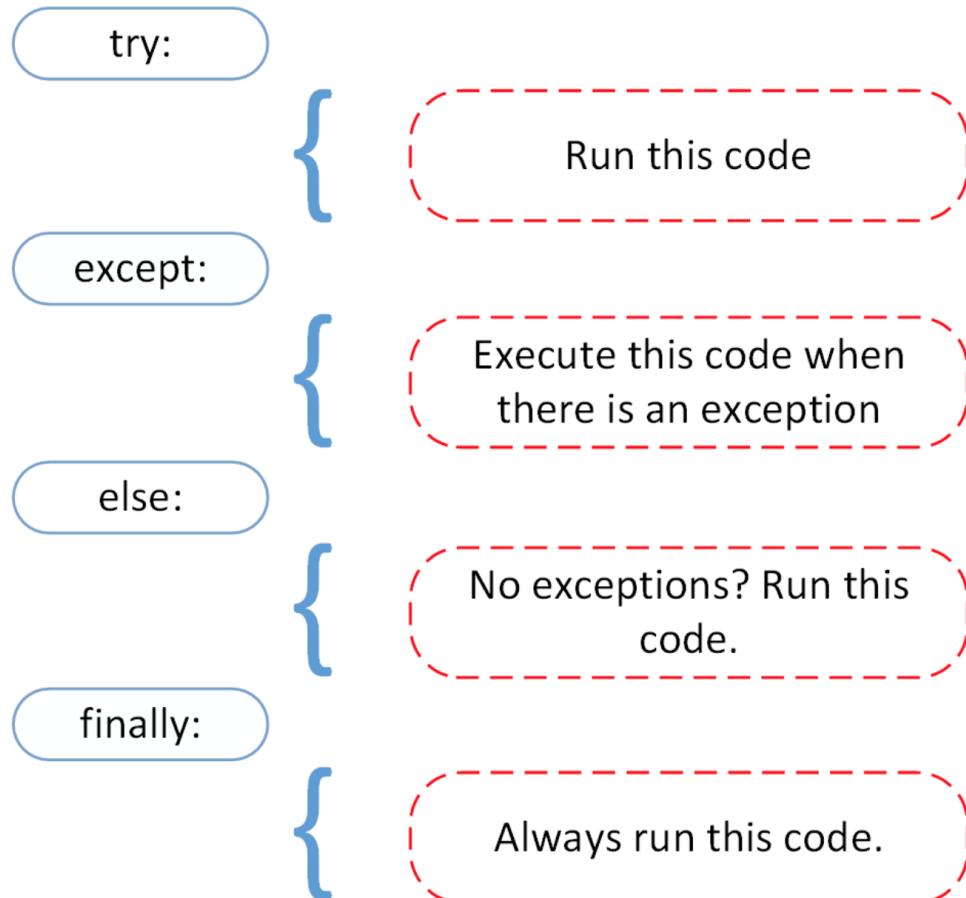
You can also have `try-except-else-finally` structure. In cases where exception is not raised inside `try`, `else` will be executed before `finally`. If there is an exception, `else` block is not executed.

```
[15]: def calculate_division(var1, var2):
        return var1 / var2

    try:
        calculate_division(1, 0)
    except ZeroDivisionError:
        print('Something happened!')
    except TypeError:
        print('Something happened!')
    else:
        print('All went well!')
    finally:
        print('Always do it!')
```

```
[15]: Something happened!
      Always do it!
```

3.9.7 Summarizing



3.10 File I/O

3.10.1 Working with paths

```
[113]: import os

current_file = os.path.realpath('01 - Python Basics.ipynb')
print('file: {}'.format(current_file))

current_dir = os.path.dirname(current_file)
print('directory: {}'.format(current_dir))

data_dir = os.path.join(current_dir, 'resources')
print('data: {}'.format(data_dir))
```

```
file: /Volumes/GoogleDrive/My
Drive/Unimore/Didattica/ooprogramming/python/slides/01 - Python Basics.ipynb
directory: /Volumes/GoogleDrive/My
Drive/Unimore/Didattica/ooprogramming/python/slides
data: /Volumes/GoogleDrive/My
Drive/Unimore/Didattica/ooprogramming/python/slides/resources
```

3.10.2 Checking paths

```
[114]: print('exists: {}'.format(os.path.exists(current_dir)))
print('is file: {}'.format(os.path.isfile(current_dir)))
print('is directory: {}'.format(os.path.isdir(current_dir)))
```

```
exists: True
is file: False
is directory: True
```

3.10.3 Reading files

The `with` statement is for obtaining a `context manager` that will be used as an execution context for the commands inside the `with`. Context managers guarantee that certain operations are done when exiting the context.

In this case, the context manager guarantees that `file_path.close()` is implicitly called when exiting the context. This is a way to make developers life easier: you don't have to remember to explicitly close the file you opened nor be worried about an exception occurring while the file is open. Unclosed files maybe a source of a resource leak. Thus, prefer using `with open()` structure when working with I/O.

```
[115]: # Don't do this!
file_path = os.path.join(data_dir, 'cars.txt')
simple_file = open(file_path, 'r')

for line in simple_file:
    print(line.strip())
simple_file.close() # This has to be called explicitly
```

```
BMW, M3, 120
Toyota, Supra, 130
Nissan, GTR, 140
```

```
[116]: # Do this!
file_path = os.path.join(data_dir, 'cars.txt')

with open(file_path, 'r') as simple_file:
    for line in simple_file:
        print(line.strip())
```

```
BMW, M3, 120
Toyota, Supra, 130
Nissan, GTR, 140
```

3.10.4 Writing files

```
[117]: new_file_path = os.path.join(data_dir, 'new_file.txt')

with open(new_file_path, 'w') as my_file:
    my_file.write('This is my first file that I wrote with Python.')
```

Now go and check that there is a new_file.txt in the data directory. After that you can delete the file by:

```
[118]: if os.path.exists(new_file_path): # make sure it's there
    os.remove(new_file_path)
```

3.11 Comments

As you begin to write more complicated code, you will have to spend more time thinking about how to code solutions to the problems you want to solve. Once you come up with an idea, you will spend a fair amount of time troubleshooting your code, and revising your overall approach.

Comments allow you to write within your program. In Python, any line that starts with a pound (#) symbol is ignored by the Python interpreter.

```
[119]: # This line is a comment.
print('This line is not a comment, it is code.')
```

```
This line is not a comment, it is code.
```

3.11.1 Docstring

Docstring is a short for documentation string. Python docstrings are the string literals that appear right after the definition of a function, method, class, or module. Triple quotes are used. The docstring is available as the `doc` attribute of the function. *Although optional, documentation is a key programming practice.*

```
[120]: def greet(name):
    """This function greets to the
    person passed in as a parameter
    """
    print('Hello! ' + name)

print(greet.__doc__)
```

```
This function greets to the
person passed in as a parameter
```

3.11.2 What makes a good comment?

- *It is short and to the point, but a complete thought.* Most comments should be written in complete sentences.
- It explains your thinking, so that when you return to the code later you will understand how you were approaching the problem. It also helps others working with your code to understand your approach.
- It explains particularly difficult sections of code in detail.

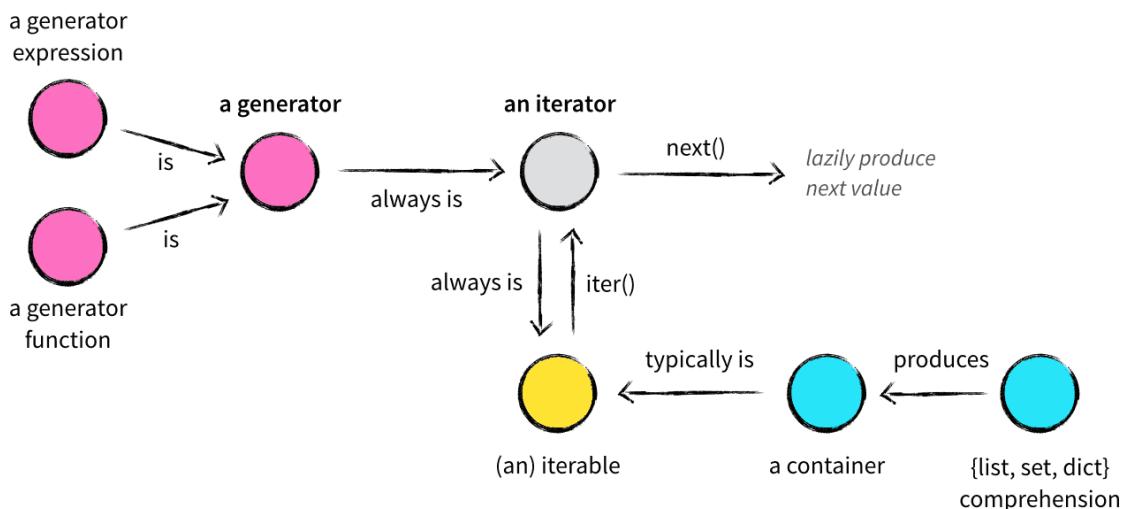
3.11.3 When should you write comments?

- When you have to think about code before writing it.
- When you are likely to forget later exactly how you were approaching a problem.
- When there is more than one way to solve a problem.
- When others are unlikely to anticipate your way of thinking about a problem.

Writing good comments is one of the clear signs of a good programmer. If you have any real interest in taking programming seriously, start using comments **now**.

3.12 Data Structures

3.12.1 General Concepts



3.12.2 Containers

Containers are data structures holding elements, and that support membership tests. They are data structures that live in memory, and typically hold all their values in memory, too. In Python, some well known examples are:

- **list**, **deque**, ...
- **set**, **frozensets**, ...
- **dict**, **defaultdict**, **OrderedDict**, **Counter**, ...
- **tuple**, **namedtuple**, ...
- **str**

Containers are easy to grasp, because you can think of them as real life containers: a box, a cupboard, a house, a ship, etc.

Technically, an object is a container when it can be asked whether it contains a certain element. You can perform such membership tests on lists, sets, tuples, dictionaries, strings:

```
[3]: assert 1 in [1, 2, 3]      # lists
assert 4 not in [1, 2, 3]
assert 1 in {1, 2, 3}        # sets
assert 4 not in {1, 2, 3}
assert 1 in (1, 2, 3)       # tuples
assert 4 not in (1, 2, 3)
```

```
[96]: d = {1: 'foo', 2: 'bar', 3: 'qux'}
assert 1 in d
assert 4 not in d
assert 'foo' not in d # 'foo' is not a _key_ in the dict
```

```
[97]: s = 'foobar'
assert 'b' in s
assert 'x' not in s
assert 'foo' in s # a string "contains" all its substrings
```

3.12.3 Iterables

As said, most containers are also iterable. But many more things are iterable as well. Examples are open files, open sockets, etc. Where containers are typically finite, an iterable may just as well represent an infinite source of data.

An iterable is any object, not necessarily a data structure, that can return an iterator (with the purpose of returning all of its elements). That sounds a bit awkward, but there is an important difference between an iterable and an iterator. Take a look at this example:

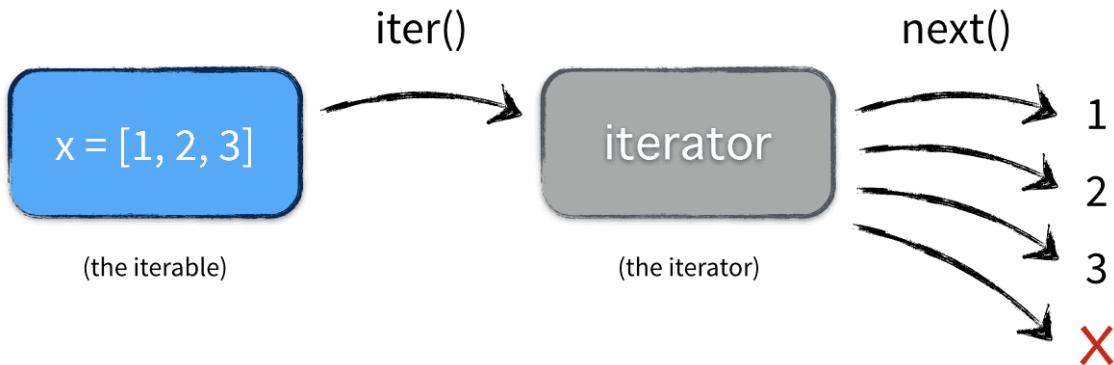
```
[99]: x = [1, 2, 3]
y = iter(x)
z = iter(x)
print(next(y))
print(next(y))
print(next(z))
print(type(x))
print(type(y))
```

```
[99]: 1
2
1
<class 'list'>
<class 'list_iterator'>
```

Here, x is the iterable, while y and z are two individual instances of an iterator, producing values from the iterable x. Both y and z hold state, as you can see from the example. In this example, x is a data structure (a list), but that is not a requirement. Finally, when you write:

```
[106]: x = [1, 2, 3]
for elem in x:
    print(elem)
```

```
[106]: 1
2
3
```



3.12.4 Iterables and for loops

```
[102]: # string
string = 'Hello'
for c in string:
    print(c)
```

```
[102]: H
e
l
l
o
```

```
[2]: # list
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

```
[2]: 1
2
3
```

```
[3]: # tuple
numbers = (1, 2, 3)
for n in numbers:
    print(n)
```

```
[3]: 1
2
3
```

```
[4]: # set
numbers = {1, 2, 3}
for n in numbers:
    print(n)
```

```
[4]: 1
2
3
```

```
[5]: # dictionary
map = {1 : 'a', 2 : 'b', 3 : 'c'}
for k, v in map.items():
    print(k, v)
```

```
[5]: 1 a
2 b
3 c
```

3.12.5 Iterators

So what is an iterator then? It's a stateful helper object that will produce the next value when you call `next()` on it. Any object that has a `__next__()` method is therefore an iterator. How it produces a value is irrelevant. An iterator can be viewed as a value factory. Each time you ask it for "the next" value, it knows how to compute it because it holds internal state.

There are countless examples of iterators. All of the `itertools` functions return iterators.

```
[104]: # Some produce infinite sequences:
from itertools import count
counter = count(start=13)
print(next(counter))
print(next(counter))
```

```
[104]: 13
14
```

```
[105]: # Some produce infinite sequences from finite sequences:
from itertools import cycle
colors = cycle(['red', 'white', 'blue'])
print(next(colors))
print(next(colors))
print(next(colors))
print(next(colors))
```

```
[105]: red
white
blue
red
```

3.12.6 Functions acting on iterables

- `list`, `tuple`, `set`, `dict`: construct a list, tuple, set, or dictionary from the content of an iterable
- `sorted`: return a list of the sorted content of an iterable
- `any`: returns True if `bool(item)` was True for any item in the iterable
- `all`: returns True if `bool(item)` was True for all items in the iterable
- `sum`: sum the content of an iterable
- `max`: return the largest value in an iterable
- `min`: return the smallest value in an iterable

```
[6]: print(list('La fiducia è bene'))
print(tuple('Il controllo è meglio'))
print(sorted('nicola'))
print(sorted(((0,1), (2,3), (0, 0), (1,0))))
```

```

print(any((0, None, [], 1)))
print(all(([1, (0, 1), True, []])))
print(sum([1, 2, 3]))
print(max((5, 8, 9, 0)))
print(min((5, 8, 9, 0)))
print(max('hello'))
print(min('hello'))

```

```

[6]: ['L', 'a', ' ', 'f', 'i', 'd', 'u', 'c', 'i', 'a', ' ', 'è', ' ', 'b', 'e', 'n',
'e']
('I', 'l', ' ', 'c', 'o', 'n', 't', 'r', 'o', 'l', 'l', 'o', ' ', 'è', ' ', 'm',
'e', 'g', 'l', 'i', 'o')
['a', 'c', 'i', 'l', 'n', 'o']
[(0, 0), (0, 1), (1, 0), (2, 3)]
True
False
6
9
0
o
e

```

3.13 List

Implementation: resizable array, **Mutable:** yes, **Insertion order:** yes, **Allows duplicates:** yes

Lists are based on resizable arrays. They are mutable and thus can be altered after creation. They can grow and shrink by adding and removing objects as needed. It's also possible to change any object stored in any slot. Items can be of different types (integer, float, string etc.). A list can also have another list as an item. Lists keep insertion order and have a definite count. The elements in a list are indexed according to a definite sequence starting from 0. A list is created by placing all the items inside square brackets [], separated by commas.

Since lists are collection of objects, it is good practice to give them a plural name. If each item in your list is a car, call the list ‘cars’. This gives you a straightforward way to refer to the entire list (‘cars’), and to a single item in the list (‘car’).

```

[121]: # empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed data types
my_list = [1, 'Hello', 3.4]

# nested list
my_list = [['mouse', 'cat', 'dog'], [8, 4, 6]]

```

3.13.1 Accessing elements

As seen for all sequences, elements of a list can be accessed via both indexing and slicing. The index must be an integer. Trying to access indices out of valid bounds raises *IndexError*.

```
[9]: my_list = list('source code')
print(my_list)

# Indexing
print(my_list[0])
print(my_list[4])

# Nested indexing
my_list = ['Happy', [2, 0, 1, 5]]
print(my_list[0][1])
print(my_list[1][3])

# print(my_list[9999])
# IndexError: list index out of range

# print(my_list[4.0])
# TypeError: list indices must be integers or slices, not float
```

```
[9]: ['s', 'o', 'u', 'r', 'c', 'e', ' ', 'c', 'o', 'd', 'e']
s
c
a
5
```

```
[10]: my_list = list('source code')

# Negative indexing
print(my_list)
print(my_list[-1])
print(my_list[-2])
print(my_list[-3])
```

```
[10]: ['s', 'o', 'u', 'r', 'c', 'e', ' ', 'c', 'o', 'd', 'e']
e
d
o
```

```
[129]: my_list = list('source code')

# Slicing
# elements from beginning to 4th (excluded)
print(my_list[0:3])

# elements from beginning to the last one (excluded)
print(my_list[: -1])

# elements from 6th to end
print(my_list[5:])

# elements from beginning to end (shallow copy)
print(my_list[:])
```

```
[129]: ['s', 'o', 'u']
['s', 'o', 'u', 'r', 'c', 'e', ' ', 'c', 'o', 'd']
['e', ' ', 'c', 'o', 'd', 'e']
['s', 'o', 'u', 'r', 'c', 'e', ' ', 'c', 'o', 'd', 'e']
```

3.13.2 Changing elements

Lists are mutable, meaning their elements can be changed unlike for strings or tuples. We can use the assignment operator (`=`) to change an item or a range of items. It is possible to change entire slices eventually.

```
[11]: my_list = [2, 4, 6, 8]
print(my_list)

# change the 1st item
my_list[0] = 1
print(my_list)

# replace a slice with another slice (even of different length)
my_list[1:3] = [3, 5, 7]
print(my_list)
```

```
[11]: [2, 4, 6, 8]
[1, 4, 6, 8]
[1, 3, 5, 7, 8]
```

3.13.3 Adding elements

We can add one item at the end of a list using the `append()` method. We can add one item in a specific position using the `insert()` method. We can add a (flat) group of items using the `extend()` method.

```
[14]: # Appending and Extending lists
my_list = []
my_list.append('hello')
print(my_list)
my_list.append([9, 11])
print(my_list)
my_list.insert(0, 1)
print(my_list)
my_list.extend([17, 19])
print(my_list)
# extending in arbitrary positions
my_list[0:0] = [1, 2, 3]
print(my_list)
```

```
[14]: ['hello']
['hello', [9, 11]]
[1, 'hello', [9, 11]]
[1, 'hello', [9, 11], 17, 19]
[1, 2, 3, 1, 'hello', [9, 11], 17, 19]
```

3.13.4 Removing elements

We can delete one or more items from a list using the keyword `del`. `del` can even delete the list entirely.

```
[16]: # Deleting list items
my_list = list('source code')

# delete one item
del my_list[1]
print(my_list)

# delete multiple items
del my_list[0:6]
print(my_list)

# delete entire list
del my_list
# print(my_list)
# NameError: name 'my_list' is not defined
```

```
[16]: ['s', 'u', 'r', 'c', 'e', ' ', 'c', 'o', 'd', 'e']
['c', 'o', 'd', 'e']
```

We can also use the *remove()* method to remove the given item or *pop()* method to remove an item at the given index. The *pop()* method removes and returns the last item if an index is not provided. Useful for implementing stacks or queues. We can also use the *clear()* method to empty a list.

```
[16]: my_list = list('source code')

my_list.remove('d')
print(my_list)

print(my_list.pop(1))
print(my_list)

print(my_list.pop())
print(my_list)

my_list.clear()
print(my_list)
```

```
[16]: ['s', 'o', 'u', 'r', 'c', 'e', ' ', 'c', 'o', 'e']
o
['s', 'u', 'r', 'c', 'e', ' ', 'c', 'o', 'e']
e
['s', 'u', 'r', 'c', 'e', ' ', 'c', 'o']
[]
```

3.13.5 Sorting

We can also sort a list, in either order.

```
[17]: students = ['lucy', 'bernie', 'aaron', 'cody']
students.sort()
print(students)

# reverse sort
students.sort(reverse=True)
```

```

print(students)

# sorts the list using the last letter of the string
students.sort(key=lambda x: x[-1], reverse=True)
print(students)

# reverse sort
students.reverse()
print(students)

```

```
[17]: ['aaron', 'bernie', 'cody', 'lucy']
['lucy', 'cody', 'bernie', 'aaron']
['lucy', 'cody', 'aaron', 'bernie']
['bernie', 'aaron', 'cody', 'lucy']
```

Keep in mind that `sort()` modifies the list. You can not recover the original order. If you want to display a list in sorted order, while preserving the original order, you can use the `sorted()` function.

```
[128]: students = ['bernice', 'aaron', 'cody']
print(sorted(students, reverse=True, key=lambda x : x[-1]))
print(students)
```

```
[128]: ['cody', 'aaron', 'bernice']
['bernice', 'aaron', 'cody']
```

3.13.6 Other List operations

```
[122]: my_list = list('abc')
print(my_list + ['a', 'b', 'c'])
```

```
[122]: ['a', 'b', 'c', 'a', 'b', 'c']
```

```
[123]: my_list = list('abc')
print(my_list * 3)
```

```
[123]: ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

```
[125]: my_list = list('abc')
# length
print(len(my_list))
```

```
[125]: 3
```

```
[126]: # membership
print('a' in my_list)
print('a' not in my_list)
```

```
[126]: True
False
```

```
[124]: # iteration (for-each)
for i in my_list:
    print(i)
```

```
# iteration by index
length = len(my_list)
for i in range(length):
    print(my_list[i])
```

```
[124]: a
b
c
a
b
c
```

3.14 Tuple

Implementation: records, **Mutable:** no, mutable items, **Insertion order:** yes, **Allows duplicates:** yes

A tuple is a collection of objects much like a list. The sequence of values stored in a tuple can be of any type, and they are indexed by integers. Tuples are immutable. A tuple cannot change once it has been assigned. Eventually, we can change its internal items, if they are mutable (e.g., a list contained in a tuple). A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

```
[23]: # empty tuple
my_tuple = ()
print(my_tuple)

# tuple with integers
my_tuple = (1, 2, 3, 3)
print(my_tuple)

# tuple with mixed datatypes
my_tuple = (1, 'Hello', 3.4)
print(my_tuple)

# nested tuple
my_tuple = ('mouse', [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

```
[23]: ()
(1, 2, 3, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
```

Creating a tuple with one element is a bit tricky. Having one element within parentheses is not enough. A trailing comma to indicate that it is a tuple is required.

```
[24]: # This actually creates a string
my_tuple = ('hello')
print(type(my_tuple))

# Creating a tuple having one element
my_tuple = ('hello',)
print(type(my_tuple))
```

```
# Parentheses are optional
my_tuple = 'hello',
print(type(my_tuple))
```

```
[24]: <class 'str'>
<class 'tuple'>
<class 'tuple'>
```

3.14.1 Accessing elements

As seen for all sequences, elements of a tuple can be accessed via both indexing and slicing. The index must be an integer. Trying to access indices out of valid bounds raises *IndexError*.

```
[25]: my_tuple = tuple('source code')

# indexing
print(my_tuple[0])
print(my_tuple[5])

# negative indexing
print(my_tuple[-1])
print(my_tuple[-2])

# slicing
print(my_tuple[:6])
print(my_tuple[-4:])
```

```
[25]: s
e
e
d
('s', 'o', 'u', 'r', 'c', 'e')
('c', 'o', 'd', 'e')
```

3.14.2 Changing elements

Unlike lists, tuples are immutable. Elements of a tuple cannot be changed once they have been assigned. However, if the element is itself a mutable data type like list, its nested items can be changed. We can also assign a tuple to different values (reassignment).

```
[132]: my_tuple = (4, 2, 3, [6, 5])
print(my_tuple)

# my_tuple[1] = 9
# TypeError: 'tuple' object does not support item assignment

# However...
my_tuple[3][0] = 5
my_tuple[3].append(5)
print(my_tuple)

# Tuples can provide the illusion of mutability by being re-assigned (as seen for
# strings)
my_tuple = ('n', 'i', 'c', 'o', 'l', 'a')
```

```
print(my_tuple)
```

```
[132]: (4, 2, 3, [6, 5])
(4, 2, 3, [5, 5, 5])
('n', 'i', 'c', 'o', 'l', 'a')
```

We cannot change the elements in a tuple. It means that we cannot add or remove items from a tuple. Deleting a tuple entirely, however, is possible using the keyword `del`.

```
[27]: my_tuple = ('n', 'i', 'c', 'o', 'l', 'a')

# del my_tuple[3]
# TypeError: 'tuple' object doesn't support item deletion

del my_tuple

# print(my_tuple)
# NameError: name 'my_tuple' is not defined
```

3.14.3 Other Tuple operations

```
[136]: my_tuple = tuple('abc')
print(my_tuple + ('a', 'b', 'c'))
```

```
[136]: ('a', 'b', 'c', 'a', 'b', 'c')
```

```
[137]: my_tuple = tuple('abc')
print(my_tuple * 3)
```

```
[137]: ('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
```

```
[138]: my_tuple = tuple('abc')
# length
print(len(my_tuple))
```

```
[138]: 3
```

```
[139]: # membership
print('a' in my_tuple)
print('a' not in my_tuple)
```

```
[139]: True
False
```

```
[142]: # iteration (for-each)
for i in my_tuple:
    print(i)

# iteration by index
length = len(my_tuple)
for i in range(length):
    print('{} -> {}'.format(i, my_tuple[i]))
```

```
[142]: a  
b  
c  
0 -> a  
1 -> b  
2 -> c
```

3.14.4 Advantages of Tuple over List

Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list:

- Conventionally, use tuples for heterogeneous data types and lists for homogeneous data types.
- Since tuples are immutable, iterating through tuples is faster.
- If you have data that doesn't change, implementing it as a tuple guarantees write-protection.
- Tuples containing immutable items can be used as keys for a dictionaries (requiring immutability). Most python objects (booleans, integers, floats, strings, and tuples) are immutable. Lists, sets, and dictionaries are mutable.

```
[148]: d = { (0, 3, 0, 1) : 'test_01'}  
  
d = { (1, 2.3, 'a', True) : 'test_01'}  
  
# d = { (1, 2.3, 'a', []) : 'test_01'}  
# TypeError: unhashable type: 'list'  
  
# d = { {1, 2, 3} : 'test_01'}  
# TypeError: unhashable type: 'set'  
  
# d = { {1 : 'a'} : 'test_01'}  
# TypeError: unhashable type: 'dict'
```

3.15 Set

Implementation: hash table, **Mutable:** yes, **Insertion order:** no, **Allows duplicates:** no

A set is an unordered collection data type, mutable, which do not allow duplicate elements. Set items must be immutable (e.g., lists, sets, dictionaries are not allowed). Questo perché in caso di elemento mutabile all'interno ed in caso di modifica di questo elemento mutabile l'hash dell'elemento non sarebbe più lo stesso (il set non sa come individuare l'elemento).

The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. Sets are based on a data structure known as a hash table. Because of this, sets do not retain insertion order and cannot be accessed by index.

A set is created by placing all the items inside curly braces {}, separated by comma, or by using the built-in `set()` function. It can have any number of items and they may be of different types as long as they are immutable.

```
[31]: # set of integers  
my_set = {1, 4, 8, 4}  
print(my_set)  
  
# set of mixed types  
my_set = {1.0, 'Hello', (1, 2, 3)}  
print(my_set)
```

```
# including mutable items
# my_set = {1, 2, [3, 4]}
# TypeError: unhashable type: 'list'

# set from list
my_set = set([1, 2, 3, 2])
print(my_set)
```

[31]: {8, 1, 4}
{1.0, (1, 2, 3), 'Hello'}
{1, 2, 3}

[32]: # Distinguish set and dictionary while creating empty sets

```
# initialize a with {}
a = {}
print(type(a))

# initialize a with set()
a = set()
print(type(a))
```

[32]: <class 'dict'>
<class 'set'>

3.15.1 Accessing elements

Since Sets are unordered, indexing has no meaning. We cannot access or change an element of a set using indexing or slicing. The Set data type does not support it.

[33]: my_set = {99.2, 1.4, 3.6, 77.6}

```
for item in my_set:
    print(item)

# my_set[0] = 2
# TypeError: 'set' object does not support item assignment
```

[33]: 99.2
1.4
3.6
77.6

3.15.2 Adding elements

Sets are mutable. We can add a single element using the `add()` method, and multiple elements using the `update()` method. The `update()` method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

[34]: # initialize my_set

```
my_set = {1, 2}
print(my_set)

# add an element
```

```

my_set.add(3)
print(my_set)

# add multiple elements
my_set.update([2, 3, 4])
print(my_set)

```

```
[34]: {1, 2}
{1, 2, 3}
{1, 2, 3, 4}
```

3.15.3 Removing elements

A particular item can be removed from a set using the methods `discard()` and `remove()`. The only difference between the two is that the `discard()` function don't do anything if the element is not present in the set. On the other hand, the `remove()` function will raise an exception in such a case.

```

[19]: # initialize my_set
my_set = {'italian', 'english', 'chinese', 'spanish', 'german', 'french'}
print(my_set)

# remove an element (present)
my_set.discard('german')
my_set.remove('french')
print(my_set)

# remove an element (not present)
my_set.discard('german')
#my_set.remove('french')
# KeyError: 'french'
print(my_set)

```

```
[19]: {'french', 'italian', 'chinese', 'english', 'spanish', 'german'}
{'italian', 'chinese', 'english', 'spanish'}
{'italian', 'chinese', 'english', 'spanish'}
```

```

[149]: # managing eventual exceptions
my_set = {'italian', 'english', 'chinese', 'spanish', 'german', 'french'}
try:
    my_set.remove('french')
    my_set.remove('french')
    my_set.remove('french')
except KeyError as e:
    print('[warn] missing key', e)

```

```
[149]: [warn] missing key 'french'
```

3.15.4 Other Set operations

Sets are mostly used for:

- removing duplicates from lists
- checking membership efficiently
- operations on ensembles (union, intersection, difference and symmetric difference, ...)

```
[37]: # removing duplicates
my_list = [1, 2, 3, 4, 1, 2, 3]
print(my_list)
my_list = list(set(my_list))
print(my_list)
```

```
[37]: [1, 2, 3, 4, 1, 2, 3]
[1, 2, 3, 4]
```

```
[38]: # checking membership efficiently
my_set = {1, 2, 3, 4, 5}
print(1 in my_set)
print(6 in my_set)
```

```
[38]: True
False
```

Sets are significantly faster than lists when it comes to determining if an object is present in the set (as in `x in s`), but are slower than lists when it comes to iterating over their contents.

```
[20]: # search performance in lists depends on elements position
# search performance in sets DO NOT depend on elements position

import timeit
t = timeit.timeit(lambda: 'a' in {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'}, number=1000000)
print('t={:.3f}s'.format(t))

t = timeit.timeit(lambda: 'i' in {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'}, number=1000000)
print('t={:.3f}s'.format(t))

t = timeit.timeit(lambda: 'a' in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'], number=1000000)
print('t={:.3f}s'.format(t))

t = timeit.timeit(lambda: 'i' in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'], number=1000000)
print('t={:.3f}s'.format(t))
```

```
[20]: t=0.099s
t=0.102s
t=0.086s
t=0.215s
```

```
[24]: # search performance in lists is linear to the number of elements
# search performance in sets DOES NOT DEPEND ON the number of elements

import timeit
import random

items = 100
test_list = list(range(items))
```

```

test_set = set(range(items))

t = timeit.timeit(lambda: random.randint(0, items - 1) in test_set, number=100000)
print('t={:.3f}s'.format(t))

t = timeit.timeit(lambda: random.randint(0, items - 1) in test_list, number=100000)
print('t={:.3f}s'.format(t))

```

[24]: t=0.120s
t=0.184s

```

[41]: # operations on ensembles
a = {1, 2, 3, 4, 5, 6}
b = {5, 6, 7, 8}

print(a.issubset(b))
print(a.isdisjoint(b))
print(a.union(b))
print(a.intersection(b))
print(a.difference(b))
print(a.symmetric_difference(b))

#pip install matplotlib-venn
from matplotlib_venn import venn2
venn2([a, b])

```

[41]: False
False
{1, 2, 3, 4, 5, 6, 7, 8}
{5, 6}
{1, 2, 3, 4}
{1, 2, 3, 4, 7, 8}

[41]: <matplotlib_venn._common.VennDiagram at 0x7fce97eaf7c0>

3.15.5 Frozenset

Frozenset has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets. Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets being immutable and hashable can be used as keys into a dictionary. Frozensets can be created using the *frozenset()* function.

```

[42]: a = frozenset([1, 2, 3, 4])
b = frozenset([3, 4, 5, 6])

# ensemble-related methods work fine
print(a.issubset(b))
print(a.isdisjoint(b))
print(a.union(b))
print(a.intersection(b))
print(a.difference(b))
print(a.symmetric_difference(b))

# mutability-related methods fail

```

```
# s1.add(5)
# AttributeError: 'frozenset' object has no attribute 'add'

# s1.remove(3)
# AttributeError: 'frozenset' object has no attribute 'remove'
```

[42]: False
False
frozenset({1, 2, 3, 4, 5, 6})
frozenset({3, 4})
frozenset({1, 2})
frozenset({1, 2, 5, 6})

3.16 Dictionary

Implementation: hash table, **Mutable:** yes, **Insertion order:** no, **Allows duplicates:** no in keys, yes in values

A dictionary is an unordered collection of *key:value* pairs. Each key is unique and immutable and has a value associated with it. The values associated with a key can be any object. Dictionaries are unordered and mutable. Dictionaries can have any number of pairs. The order in which pairs are added to a dictionary is not maintained and has no meaning.

[158]: # Creating a Dictionary with Integer Keys
dict = {
 1: 'Geeks',
 2: 'For',
 3: 'Geeks'
}
print(dict)

Creating a Dictionary with Mixed keys
dict = {
 'Name': 'Geeks',
 1: [1, 2, 3, 4]
}
print(dict)

Creating a Dictionary representing a Person
dict = {
 'Name': 'Mario',
 'Lastname': 'Rossi',
 'Age' : 33,
}
print(dict)

[158]: {1: 'Geeks', 2: 'For', 3: 'Geeks'}
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
{'Name': 'Mario', 'Lastname': 'Rossi', 'Age': 33}

3.16.1 Accessing elements

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets // or with the *get()* method. If we use the square brackets //, *KeyError* is raised in case a key is not found. On the other hand, the *get()* method returns *None* if the key is not found.

```
[162]: # get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

print(my_dict.get('name'))
print(my_dict['name'])

print(my_dict.get('address'))
try:
    print(my_dict['address'])
except KeyError as e:
    print('KeyError: {}'.format(e))
```

```
[162]: Jack
Jack
None
KeyError: 'address'
```

3.16.2 Changing elements

We can change the value of existing items using the assignment operator (`=`). If the key is already present, then the existing value gets updated. In case the key is not present, a new (key: value) pair is added to the dictionary.

```
[172]: my_dict = {'name': 'Loris', 'lastname': 'Batacchi', 'age': 32}

my_dict['age'] = 33
print(my_dict)
```

```
[172]: {'name': 'Loris', 'lastname': 'Batacchi', 'age': 33}
```

3.16.3 Adding elements

```
[173]: my_dict = {'name': 'Loris', 'lastname': 'Batacchi', 'age': 32}

my_dict['address'] = 'Sant'Arcangelo di Romagna'
print(my_dict)
```

```
[173]: {'name': 'Loris', 'lastname': 'Batacchi', 'age': 32, 'address': "Sant'Arcangelo
di Romagna"}
```

3.16.4 Removing elements

We can remove a particular item in a dictionary by using the `pop()` method. This method removes an item with the provided key and returns its value. The `popitem()` method, instead, can be used to remove and return the last (key, value) item. All the items can be removed at once, using the `clear()` method. We can also use the `del` keyword to remove individual items or the dictionary itself.

```
[47]: squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
print(squares)

# remove a particular item, returns its value
print('removed =', squares.pop(4))
print(squares)
```

```

# remove the last item (since Python 3.7+), return (key,value)
print('removed =', squares.popitem())
print(squares)

# remove an item using del
del squares[1]
print(squares)

# remove all items
squares.clear()
print(squares)

```

[47]: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
removed = 16
{1: 1, 2: 4, 3: 9, 5: 25}
removed = (5, 25)
{1: 1, 2: 4, 3: 9}
{2: 4, 3: 9}
{}

3.16.5 Other Dictionary Operations

We can test if a key is in a dictionary using the keyword *in*. Notice that *the membership test is only for the keys and not for the values*. We can iterate through both keys and key:value pairs using a for loop.

[48]: squares = {1: 1, 2: 4, 3: 9}
membership tests for key only
print(1 in squares)
print(9 in squares)

[48]: True
False

[174]: squares = {1: 1, 2: 4, 3: 9}
Iterating through keys
for key in squares:
 print(key, squares[key])

[174]: 1 1
2 4
3 9

[175]: squares = {1: 1, 2: 4, 3: 9}
Iterating through keys
for key in squares.keys():
 print(key, squares[key])

[175]: 1 1
2 4
3 9

```
[176]: squares = {1: 1, 2: 4, 3: 9}

# Iterating through values
for v in squares.values():
    print(v)
```

```
[176]: 1
        4
        9
```

```
[52]: squares = {1: 1, 2: 4, 3: 9}
print(list(squares.items()))

# Iterating through keys and values using unpacking
for k, v in squares.items():
    print(k, v)
```

```
[52]: [(1, 1), (2, 4), (3, 9)]
1 1
2 4
3 9
```

3.16.6 OrderedDict

In the words of Raymond Hettinger, core Python developer and coauthor of OrderedDict, the class was specially designed to keep its items ordered, whereas the new implementation of dict was designed to be compact and to provide fast iteration:

The current regular dictionary is based on the design I proposed several years ago. The primary goals of that design were compactness and faster iteration over the dense arrays of keys and values. Maintaining order was an artifact rather than a design goal. The design can maintain order but that is not its specialty.

In contrast, I gave collections.OrderedDict a different design (later coded in C by Eric Snow). The primary goal was to have efficient maintenance of order even for severe workloads such as that imposed by the lru_cache which frequently alters order without touching the underlying dict. Intentionally, the OrderedDict has a design that prioritizes ordering capabilities at the expense of additional memory overhead and a constant factor worse insertion time.

It is still my goal to have collections.OrderedDict have a different design with different performance characteristics than regular dicts. It has some order specific methods that regular dicts don't have (such as a move_to_end() and a popitem() that pops efficiently from either end). The OrderedDict needs to be good at those operations because that is what differentiates it from regular dicts. (Source)

3.17 Comprehension Expressions

3.17.1 Unpacking

Python provides some syntactic tricks for working with iterables: unpacking and enumerating. Writing clean, readable code leads to bug-free algorithms that are easy to understand. Furthermore, these tricks will also facilitate the use of other features, like comprehension-statements.

```
[177]: # assigning a list to variables using unpacking
my_list = [7, 9, 11]
x, y, z = my_list
```

```

grades = [('Ashley', 28, 27), ('Brad', 23, 26), ('Cassie', 25, 28)]

# for loop without unpacking
for entry in grades:
    print(entry)

# for loop with unpacking
for name, grade_1, grade_2 in grades:
    print(name, grade_1, grade_2)

```

[177]:

```

('Ashley', 28, 27)
('Brad', 23, 26)
('Cassie', 25, 28)
Ashley 28 27
Brad 23 26
Cassie 25 28

```

3.17.2 Enumerating

The `enumerate()` function adds a counter to an iterable and returns it. The returned object is an enumerate object. You can convert enumerate objects to lists and tuples using `list()` and `tuple()` functions respectively.

[1]:

```

brands = ['Nikola', 'Rimac', 'Polestar']
e = enumerate(brands)
print(list(e))
print(type(e))

```

[1]:

```

[(0, 'Nikola'), (1, 'Rimac'), (2, 'Polestar')]
<class 'enumerate'>

```

[2]:

```

brands = ['Nikola', 'Rimac', 'Polestar']
for index, brand in enumerate(brands):
    print(index, brand)

```

[2]:

```

0 Nikola
1 Rimac
2 Polestar

```

[3]:

```

# identify indices of None elements (without enumeration)
none_indices = []
index = 0
for item in [2, None, -10, None, 4, 8]:
    if item is None:
        none_indices.append(index)
    index += 1
print(none_indices)

```

[3]:

```

[1,3]

```

[4]:

```

# identify indices of None elements (with enumeration)
none_indices = []
for index, item in enumerate([2, None, -10, None, 4, 8]):
    if item is None:

```

```
    none_indices.append(index)
print(none_indices)
```

[4]: [1,3]

3.17.3 Generators

A generator is a special kind of iterator, which stores the instructions for how to generate each of its members, in order, along with its current state of iterations. A generator does not store its elements in memory but generates each member, one at a time, as requested via iteration.

[185]: `import collections.abc`

```
issubclass(types.GeneratorType, collections.abc.Iterator)
```

[185]: True

[186]: `# start: 0 (default, included) # stop: 3 (excluded) # step: 1 (default)`
`for i in range(3):`
 `print(i)`

[186]: 0
1
2

[58]: `# start: 12 (included) # stop: 15 (excluded) # step: 1 (default)`
`for i in range(12, 15):`
 `print(i)`

[58]: 12
13
14

[59]: `# start: 1 (included) # stop: 10 (excluded) # step: 3`
`for i in range(0, 10, 2):`
 `print(i)`

[59]: 0
2
4
6
8

3.17.4 Differences between generators and normal function

- Generator function contains one or more yield statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

```
[2]: # A simple generator function
# Generator function contains yield statements
def my_gen():
    n = 1
    print('This is printed first')
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

```
[8]: g = my_gen()
next(g)
next(g)
next(g)
#next(g)
```

```
[2]: This is printed first
This is printed second
This is printed at last
```

```
[8]: 3
```

```
[9]: # Using for loop
for item in my_gen():
    print(item)
```

```
[9]: This is printed first
1
This is printed second
2
This is printed at last
3
```

```
[24]: def rev_str(my_str):
        length = len(my_str)
        for i in range(length - 1, -1, -1):
            yield my_str[i]

# For loop to reverse the string
for char in rev_str("hello"):
    print(char)
```

```
[24]: o
l
l
e
h
```

```
[11]: def fibonacci(k):
    l = [0,1]
    n = 0
    while n < k:
        l.append(sum(l))
        l.pop(0)
        n += 1
    yield l[0]

for f in fibonacci(3):
    print(f)
```

```
[11]: 1
1
2
```

3.17.5 Generator Comprehension

A generator comprehension is a single-line specification for defining a generator. It is absolutely essential to learn this syntax in order to write simple and readable code.

```
(<expression> for <var> in <iterable> [if <condition>])
```

Specifies the general form for a generator comprehension. This produces a generator, whose instructions for generating its members are provided within the parenthetical statement.

```
[34]: example_gen = ('hello'[i] for i in range(len('hello') - 1, -1, -1))
for item in example_gen:
    print(item)
```

```
[34]: o
l
l
e
h
```

```
[32]: example_gen = (i/2 for i in [0, 9, 21, 32] if i < 30)
for item in example_gen:
    print(item)
```

```
[32]: 0.0
4.5
10.5
```

```
[61]: example_gen = ((i, i**2, i**3) for i in range(5))
for item in example_gen:
    print(item)
```

```
[61]: (0, 0, 0)
(1, 1, 1)
(2, 4, 8)
(3, 9, 27)
(4, 16, 64)
```

```
[62]: example_gen = (('even' if i % 2 == 0 else 'odd') for i in range(4))
for item in example_gen:
    print(item)
```

```
[61]: even
odd
even
odd
```

3.17.6 Consuming generators

We can feed a generator to any function accepting iterators. For instance, we can feed it to the built-in sum function, which sums the contents of an iterable. This computes the sum of the sequence of numbers without ever storing the full sequence of numbers in memory.

You must redefine the generator if you want to iterate over it again. Defining a generator requires very few resources, so this is not a point of concern.

```
[47]: gen = (i**2 for i in range(10))
# computes the sum of a generator
print(sum(gen))

# computes the sum of ... nothing!
# the generator has already been consumed!
print(sum(gen))
```

```
[47]: 285
0
```

```
[48]: gen = (i**2 for i in range(3))

try:
    print(next(gen))
    print(next(gen))
    print(next(gen))
    print(next(gen))
except StopIteration:
    print('StopIteration')
```

```
[48]: 0
1
4
StopIteration
```

A generator comprehension can be specified directly as an argument to a function.

```
[14]: # providing generator expressions as arguments to functions that operate on iterables
print(sum(i for i in range(101)))
print(list(i**2 for i in range(10)))
print(any(i < 6 for i in [6, 6, 7, 7, 8, 8]))
print(all(i < 16 for i in [6, 6, 5, 7, 8, 8]))
print(', '.join(str(i) for i in [10, 200, 4000, 800]))
```

```
[14]: 5050
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
False
True
10, 200, 4000, 800
```

3.17.7 List & Tuple Comprehensions

Using generator comprehensions to initialize lists is so useful that Python actually reserves a specialized syntax for it, known as the list comprehension. A list comprehension is a syntax for constructing a list, which exactly mirrors the generator comprehension syntax:

```
[<expression> for <var> in <iterable> [if <condition>]]
```

```
[15]: # creating a list using a comprehension expression
print(list(i**2 for i in range(5)))

# creating a tuple using a comprehension expression
print(tuple(i**2 for i in range(5)))
```

```
[15]: [0, 1, 4, 9, 16]
(0, 1, 4, 9, 16)
```

```
[67]: import math
print([math.sqrt(i) for i in range(5)])
```

```
[67]: [0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
```

```
[68]: l = [' Andrew ', ' luCY ', '     george', ' LUCAS ']
l = [s.strip().lower() for s in l]
print(l)
```

```
[68]: ['andrew', 'lucy', 'george', 'lucas']
```

```
[69]: # Select all words containing the char 'o' OR 'O'
word_collection = ['Python', 'Like', 'You', 'Mean', 'It']

# without list comprehension (4 lines)
out = []
for word in word_collection:
    if 'o' in word.lower():
        out.append(word)
print(out)

# with list comprehension (1 line)
out = [word for word in word_collection if 'o' in word.lower()]
print(out)
```

```
[69]: ['Python', 'You']
['Python', 'You']
```

```
[51]: # Transform a string in a list of numbers 0 or 1
# 1 if letter is 'o'
# 0 otherwise
```

```

word = 'Hello. How Are You?'

# without comprehension
out = []
for i in word:
    if i == 'o':
        out.append(1)
    else:
        out.append(0)
print(out)

# with comprehension
out = [(1 if letter == 'o' else 0) for letter in word]
print(out)

```

[51]: [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]

3.17.8 Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary. Dictionary comprehension consists of an expression pair (key: value) followed by a for statement inside curly braces {}.

```

[71]: # Without Dictionary Comprehension
squares = {}
for n in range(6):
    squares[n] = n**2
print(squares)

# With Dictionary Comprehension
squares = {n: n**2 for n in range(6)}
print(squares)

```

[71]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

```

[52]: words = ['mother', 'spaceship', 'is', 'lifting', 'off']
too_common = ['a', 'is', 'not', 'and', 'off']
d = {word: hash(word) for word in words if word not in too_common}

# pprint stands for pretty print
import pprint
pprint.pprint(d)

```

[52]: {'lifting': -1658600787297277306,
'mother': 6051358449637906722,
'spaceship': -4001180469585927950}

3.18 Itertools

Python has an `itertools` module, which provides a core set of fast, memory-efficient tools for creating iterators. The majority of these functions create generators, thus we will have to iterate over them in order to show their use (it is not possible to print them).

There are three built-in functions, `range()`, `enumerate()`, and `zip()`, that belong to the `itertools` module, but they are so useful that they do not need to be imported. *It is essential that range, enumerate, and zip become tools that you are comfortable with.*

3.18.1 `range()`

`range()` allows user to generate a series of numbers within a given range. Depending on how many arguments user is passing to the function, user can decide where that series of numbers will begin and end as well as how big the difference will be between one number and the next one.

```
[73]: print(range(10))
      print(list(range(10)))
```

```
[73]: range(0, 10)
      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[74]: print(range(5, 10))
      print(list(range(5, 10)))
```

```
[74]: range(5, 10)
      [5, 6, 7, 8, 9]
```

```
[75]: print(range(0, 10, 3))
      print(list(range(0, 10, 3)))
```

```
[75]: range(0, 10, 3)
      [0, 3, 6, 9]
```

3.18.2 `enumerate()`

When dealing with iterators, we also get a need to keep a count of iterations. Python eases the programmers' task by providing a built-in function `enumerate()` for this task. The `enumerate()` function adds a counter to an iterable and returns it in a form of enumerate object. This enumerate object can then be used directly in for loops or be converted into a list of tuples using `list()` method.

```
[76]: print(enumerate(['apple', 'banana', 'cat', 'dog']))
      print(list(enumerate(['apple', 'banana', 'cat', 'dog'])))
```

```
[76]: <enumerate object at 0x7fce97fae240>
      [(0, 'apple'), (1, 'banana'), (2, 'cat'), (3, 'dog')]
```

```
[77]: my_list = ['apple', 'banana', 'cat', 'dog']
      for i, item in enumerate(my_list):
          print(i, item)
```

```
[77]: 0 apple
      1 banana
      2 cat
      3 dog
```

3.18.3 `zip()`

The purpose of `zip()` is to map multiple containers so that they can be used as a single entity.

```
[78]: names = ['Angie', 'Brian', 'Cassie', 'David']
exam_1_scores = [90, 82, 79, 87]
exam_2_scores = [95, 84, 72, 91]

print(zip(names, exam_1_scores, exam_2_scores))
print(list(zip(names, exam_1_scores, exam_2_scores)))

for name, grade_1, grade_2 in zip(names, exam_1_scores, exam_2_scores):
    print(name, grade_1, grade_2)
```

```
[78]: <zip object at 0x7fce97faeb00>
[('Angie', 90, 95), ('Brian', 82, 84), ('Cassie', 79, 72), ('David', 87, 91)]
Angie 90 95
Brian 82 84
Cassie 79 72
David 87 91
```

3.19 Sorting Iterables

3.19.1 Using `sort()` and `sorted()`

Both `list.sort()` and `sorted()` can be used for sorting. Both support a `key` parameter to specify a function (or other callable) to be called on each element prior to making comparisons needed for ordering.

The value of the key parameter should be a function (or other callable) that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

```
[53]: # sorting tuples
car_tuples = [
    ('BMW', 'M2', 200),
    ('Rimac', 'Concept One', 300),
    ('Fiat', '500E', 100),
]

car_dict = [
    {'brand' : 'BMW', 'model' : 'M2', 'speed' : 200},
    {'brand' : 'Rimac', 'model' : 'Concept One', 'speed' : 300},
    {'brand' : 'Fiat', 'model' : '500E', 'speed' : 100},
]

# sort by brand
print(sorted(car_tuples, key=lambda car : car[2], reverse=True))
print(sorted(car_dict, key=lambda car : car['brand'], reverse=True))
```

```
[53]: [('Rimac', 'Concept One', 300), ('BMW', 'M2', 200), ('Fiat', '500E', 100)]
[{'brand': 'Rimac', 'model': 'Concept One', 'speed': 300}, {'brand': 'Fiat',
'model': '500E', 'speed': 100}, {'brand': 'BMW', 'model': 'M2', 'speed': 200}]
```

3.19.2 The operator module

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The `operator` module has `itemgetter()`, `attrgetter()`, and a `methodcaller()` function. `itemgetter()` fetches items from an iterable, while `attrgetter()` fetches attributes from objects.

```
[80]: import operator
print(sorted(car_tuples, key=operator.itemgetter(0)))
print(sorted(car_tuples, key=operator.itemgetter(1)))
```

```
[80]: [('BMW', 'M2', 200), ('Fiat', '500E', 100), ('Rimac', 'Concept One', 300)]
[('Fiat', '500E', 100), ('Rimac', 'Concept One', 300), ('BMW', 'M2', 200)]
```

3.19.3 Sorting on multiple levels

The `operator` module functions allows multiple levels of sorting. For example, to sort by grade then by age:

```
[81]: import operator
print(sorted(car_tuples, key=operator.itemgetter(0, 1)))
print(sorted(car_tuples, key=operator.itemgetter(1, 2)))
```

```
[81]: [('BMW', 'M2', 200), ('Fiat', '500E', 100), ('Rimac', 'Concept One', 300)]
[('Fiat', '500E', 100), ('Rimac', 'Concept One', 300), ('BMW', 'M2', 200)]
```

3.20 Copying Iterables

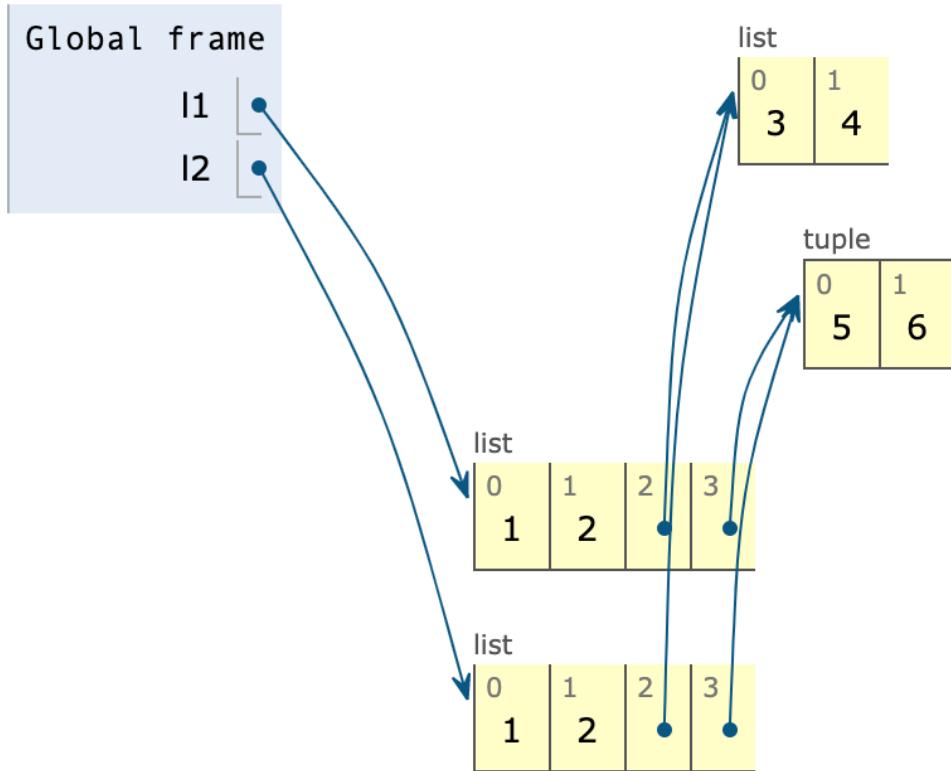
3.20.1 Shallow copy

The easiest way to copy a list (or most built-in mutable collections) is to use the built-in constructor for the type itself. For lists and other mutable sequences, the shortcut `l2 = l1[:]` also makes a copy.

```
[82]: l1 = [1, 2, [3, 4], (5, 6)]
l2 = list(l1)
l3 = l1[:]
print(l2 == l1)
print(l3 == l1)
print(l2 is l1)
print(l3 is l1)
```

```
[82]: True
True
False
False
```

However, these approaches produce shallow copies (i.e., the outermost container is duplicated, but the copy is filled with references to the same items held by the original container). This saves memory and causes no problems if all the items are immutable. But if there are mutable items, this may lead to unpleasant surprises. `pythontutor` allows for visualizing code as the examples below show. <https://www.pythontutor.com/>

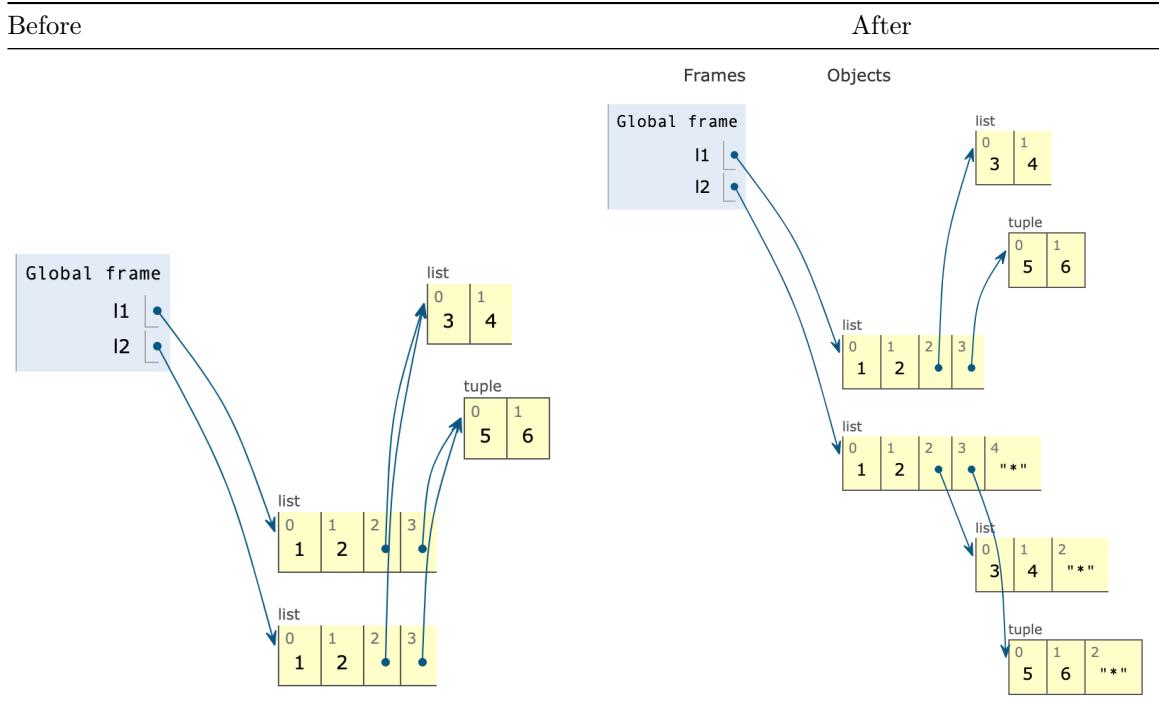


What happens when innermost iterable containers are modified?

```
[83]: l1 = [1, 2, [3, 4], (5, 6)]
l2 = list(l1)
print('l1:', l1)
print('l2:', l2)

# the copy (l2) gets modified
l2.append('*')
l2[2] = [3, 4, '*']
l2[3] = (5, 6, '*')
print('l1:', l1)
print('l2:', l2)
```

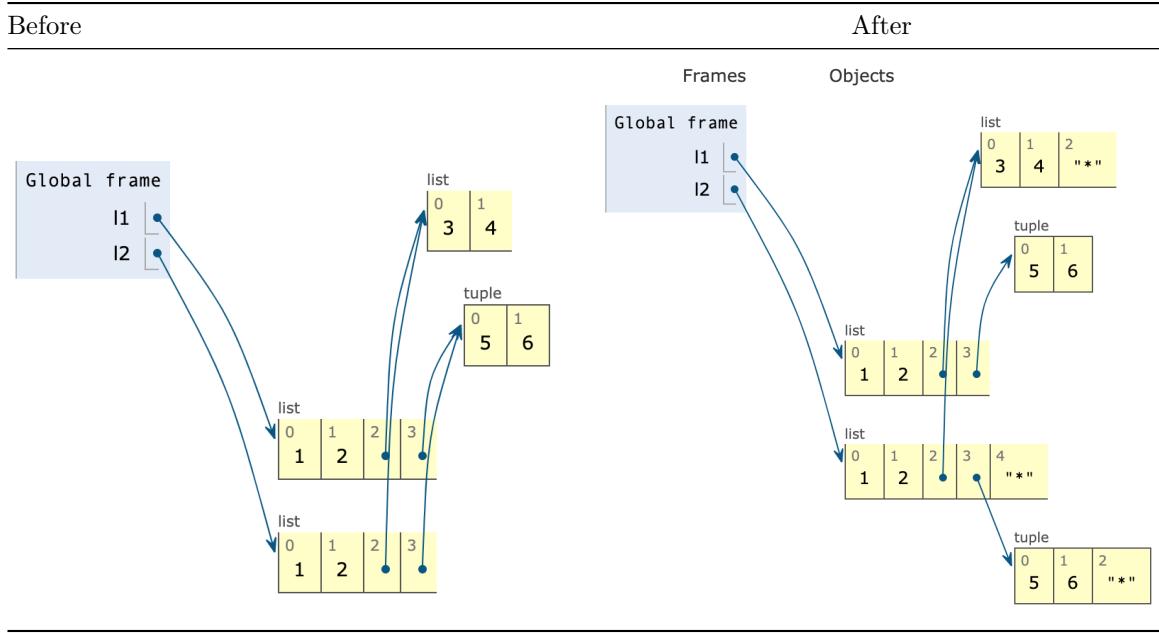
```
[14]: l1: [1, 2, [3, 4], (5, 6)]
l2: [1, 2, [3, 4], (5, 6)]
l1: [1, 2, [3, 4], (5, 6)]
l2: [1, 2, [3, 4, '*'], (5, 6, '*'), '*']
```



```
[84]: l1 = [1, 2, [3, 4], (5, 6)]
l2 = list(l1)
print('l1:', l1)
print('l2:', l2)

# the copy (l2) gets modified
l2.append('*')
l2[2].append('*')
l2[3] = (5, 6, '*')
print('l1:', l1)
print('l2:', l2)
```

```
[84]: l1: [1, 2, [3, 4], (5, 6)]
l2: [1, 2, [3, 4], (5, 6)]
l1: [1, 2, [3, 4, '*'], (5, 6)]
l2: [1, 2, [3, 4, '*'], (5, 6, '*'), '*']
```



3.20.2 Deep copy

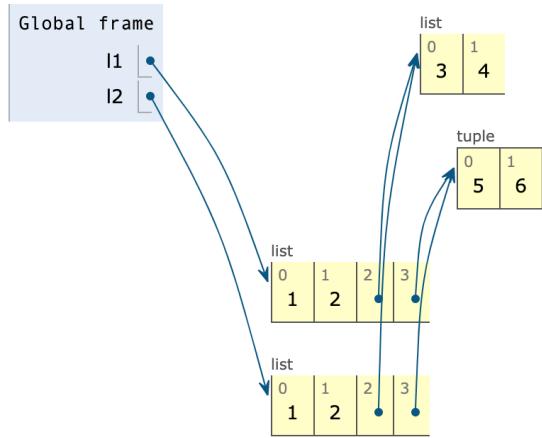
Deep copies (completely separate copies) can be obtained by making use of the *copy* module.

```
[85]: # shallow copy
l1 = [1, 2, [3, 4], (5, 6)]
l2 = list(l1)
print(l2 == l1)
print(l2 is l1)

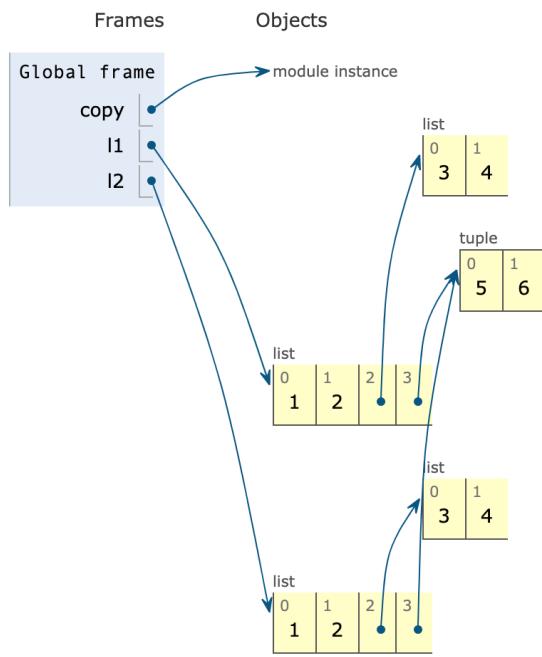
# deep copy
import copy
l1 = [1, 2, [3, 4], (5, 6)]
l2 = copy.deepcopy(l1)
print(l2 == l1)
print(l2 is l1)
```

```
[85]: True
False
True
False
```

Shallow Copy

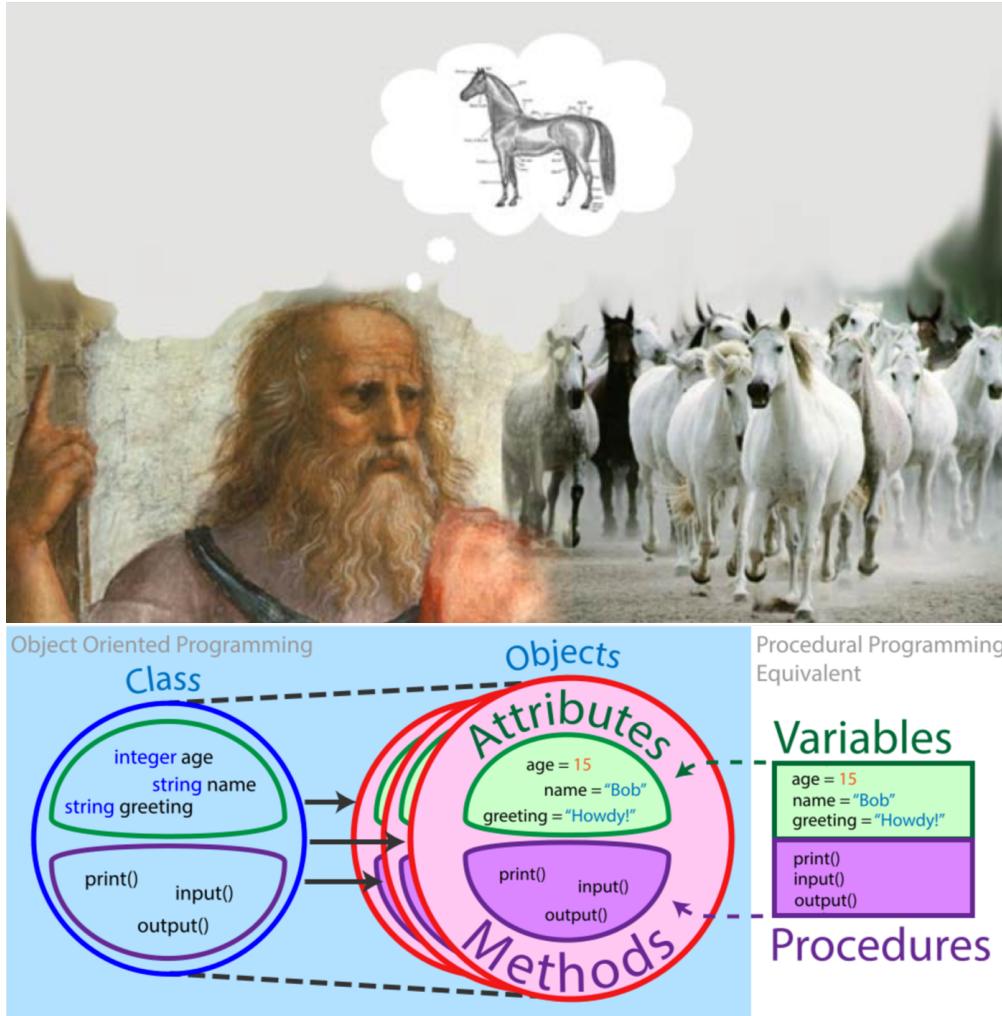


Deep Copy



3.21 Classes

3.21.1 General concepts



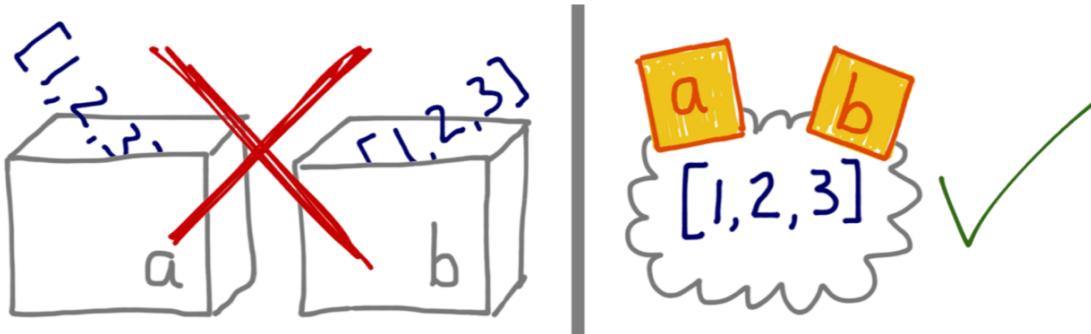
A *class* is a body of code that defines the *attributes* and *methods* required to model a class of objects. You can model something from the real world, such as a rocket ship or a car, or you can model something from a virtual world such as a rocket in a game, or a set of physical laws for a game engine.

An *attribute* is a piece of information, a property of a class of objects. In code, an attribute is a variable defined inside a class. A *method* is an action that is defined within a class. In code, they are functions defined inside a class.

An *object* is a particular instance of a class. Every object has assigns specific values to its attributes (variables). You can have as many objects as you want for any one class.

3.21.2 Object references

It is worth noticing that the usual *variables as boxes* metaphor actually hinders the understanding of reference variables in OO languages. Python variables are like reference variables in Java, so it's better to think of them as labels attached to objects instead of boxes containing data.



As an example, considering *a* and *b* as two separate boxes might obstruct the understanding of the following example. The item 4 is apparently added only to *a*. However, given the fact the *a* and *b* are only names referring to the same object, the additional element can be accessed using both *a* and *b*.

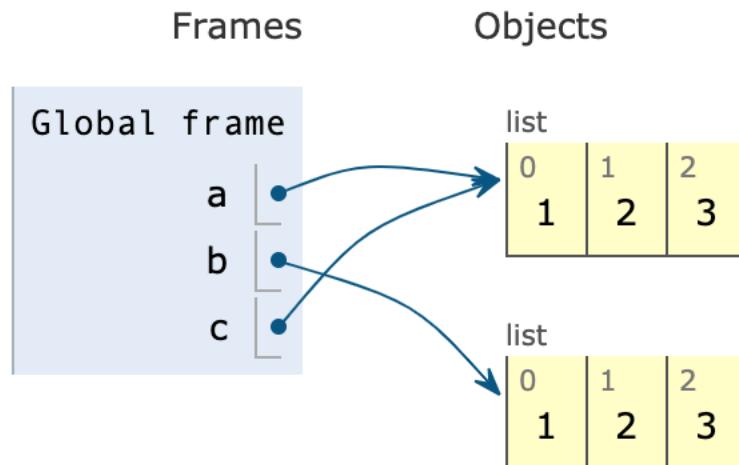
```
[1]: a = [1, 2, 3]
b = a
a.append(4)
print(a)
print(b)
print(a is b)
```

```
[1]: [1, 2, 3, 4]
[1, 2, 3, 4]
True
```

3.21.3 Object identity, type, internal state

Every object has an *identity*, a *type* and an *internal state*. An object identity is unique and never changes once it has been created. The *id()* function returns an integer representing its identity. The *type()* function provides its type (i.e., its class). The internal state is expressed by the state of internal variables. The *==* operator compares the internal state of objects (the data they hold), while the *is* operator compares their identities.

```
[2]: a = [1, 2, 3]
b = [1, 2, 3]
c = a
```



```
[3]: print('id(a):', id(a)) # identity
      print('id(b):', id(b))
      print('id(c):', id(c))

      print('type(a):', type(a)) # type
      print('type(b):', type(b))
      print('type(c):', type(c))

      print(a) # internal state
      print(b)
      print(c)
```

```
[3]: id(a): 140227319999168
      id(b): 140227319999872
      id(c): 140227319999168
      type(a): <class 'list'>
      type(b): <class 'list'>
      type(c): <class 'list'>
      [1, 2, 3]
      [1, 2, 3]
      [1, 2, 3]
```

```
[4]: print('a is b:', a is b) # identity comparison
      print('a is c:', a is c)

      print('isinstance(a, list):', isinstance(a, list)) # type comparison
      print('isinstance(b, float):', isinstance(b, float))

      print('a == b:', a == b) # internal state comparison
      print('a == c:', a == c)
```

```
[4]: a is b: False  
      a is c: True  
      isinstance(a, list): True  
      isinstance(b, float): False
```

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

3.21.4 Class definition

Classes are a way of combining information and behavior. Attributes are the properties (the information part) defining any specific class of objects. They are defined inside the `__init__()` method of the class.

Method names starting and ending with two underscores are special conventionally considered special methods. The `__init__()` method is one of these. It is called automatically when you create an object from your class (it's the constructor). The `__init__()` method lets you make sure that all relevant attributes are set to their proper values when an object is created from the class, before the object is used.

The `self` keyword refers to the current object that you are working with. When you are writing a class, it lets you refer to certain attributes from any other part of the class. Basically, all methods in a class need the `self` object as their first argument, so they can access any attribute that is part of the class.

To actually use a class, you have to create an *object* which is an instance of a *class*. Every object has a copy of each of the class's variables, and it can perform actions that are defined for the class (it can call class' methods). To access an object's attributes or methods, you give the name of the object and then use *dot notation*.

```
[5]: class Car:
    # constructor
    def __init__(self, brand, model, speed=100):
        # instance attributes
        self.brand = brand
        self.model = model
        self.speed = speed

    # methods
    def speed_up(self, step=1):
        self.speed += step

    def speed_down(self, step=1):
        self.speed -= step

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3')
    print('Brand={}, Model={}, Speed={}'.format(m3.brand, m3.model, m3.speed))
    m3.speed_up(10)
    print('Brand={}, Model={}, Speed={}'.format(m3.brand, m3.model, m3.speed))

    punto = Car('Fiat', 'Punto', 110)
    print('Brand={}, Model={}, Speed={}'.format(punto.brand, punto.model, punto.speed))

    print(id(m3), type(m3))
    print(id(punto), type(punto))
```

```
[5]: Brand=Bmw, Model=M3, Speed=100
Brand=Bmw, Model=M3, Speed=110
Brand=Fiat, Model=Punto, Speed=110
140333828574128 <class '__main__.Car'>
140333828571728 <class '__main__.Car'>
```

3.21.5 Constructor method

Constructors are generally used for creating (instantiating) objects. The task of constructors is to assign values to the object attributes. In Python the `__init__` method is used as constructor and it is always called when an object is created. *Unlike Java, you cannot define multiple constructors. However, to reach the same goal, you can define default values for the parameters.*

```
[49]: class Car:
    # constructor
    def __init__(self, brand, model, speed=0):
        # instance attributes
        self.brand = brand
        self.model = model
        self.speed = speed

    # methods
    def speed_up(self, step = 1):
        self.speed += step

    def speed_down(self, step = 1):
        self.speed -= step

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3', 70)
    print('Brand={}, Model={}, Speed={}'.format(m3.brand, m3.model, m3.speed))

m3 = Car('Bmw', 'M3')
print('Brand={}, Model={}, Speed={}'.format(m3.brand, m3.model, m3.speed))
```

```
[49]: Brand=Bmw, Model=M3, Speed=70
Brand=Bmw, Model=M3, Speed=0
```

3.21.6 Garbage collector

In addition to id, type, and internal state every Python object has a reference count. The reference count is incremented whenever the object is referenced, and it is decremented whenever an object is dereferenced. If an object's reference count goes to 0, the memory for the object is automatically freed by the *garbage collector*. La funzione `getrefcount()` crea una copia dell'oggetto passato come parametro e quindi il suo reference count aumenta di 1.

```
[51]: # calls garbage collection explicitly
import gc
gc.collect()

# counts references
import sys
print('[1, 2, 3] references =', sys.getrefcount([1, 2, 3]))

a = [1, 2, 3]
print('[1, 2, 3] references =', sys.getrefcount(a))
```

```
[51]: [1, 2, 3] references = 1
[1, 2, 3] references = 2
```

3.21.7 Destructor method

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed as in C++ because Python has a garbage collector that handles memory automatically. However, the `__del__()` method is used as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

```
[7]: class Car:
    # constructor
    def __init__(self, brand, model, speed=0):
        # instance attributes
        self.brand = brand
        self.model = model
        self.speed = speed

    # destructor
    def __del__(self):
        print('Object destroyed!')

    # methods
    def speed_up(self):
        self.speed += 1

    def speed_down(self):
        self.speed -= 1

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3')
    # m3 = None
    del m3
```

```
[7]: Object destroyed!
```

3.21.8 Class attributes

Outside the `__init__()` method we can define class attributes. We can access the class attributes using `__class__.attribute`. Class attributes are shared among all objects which are instances of the same class.

```
[9]: class Car:
    # class attribute
    wheels = 4

    # constructor
    def __init__(self, brand, model, speed=0):
        # instance attributes
        self.brand = brand
        self.model = model
        self.speed = speed

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3')
    tsla = Car('Tesla', 'Models S')
```

```

print('Brand={}, Model={}, Wheels={}'.format(m3.brand, m3.model, Car.wheels))
print('Brand={}, Model={}, Wheels={}'.format(tsla.brand, tsla.model, tsla.
→__class__.wheels))

Car.wheels = 2

print('Brand={}, Model={}, Wheels={}'.format(m3.brand, m3.model, Car.wheels))
print('Brand={}, Model={}, Wheels={}'.format(tsla.brand, tsla.model, tsla.
→__class__.wheels))

```

[9]:

```

Brand=Bmw, Model=M3, Wheels=4
Brand=Tesla, Model=Models S, Wheels=4
Brand=Bmw, Model=M3, Wheels=2
Brand=Tesla, Model=Models S, Wheels=2

```

3.21.9 @classmethod

Methods annotated with `@classmethod` have the access to the state of the class as they take a class parameter that points to the class and not the object instance. They can modify a class state (i.e., class attributes) that would apply across all the instances of the class. Generally used to create *factory methods*. Factory methods return brand new objects (similarly to a constructors).

[60]:

```

class Car:
    # class attribute
    wheels = 4

    def __init__(self, brand, model, speed=0):
        self.brand = brand
        self.model = model
        self.speed = speed

    @classmethod
    def __more_wheels__(cls):
        cls.wheels += 1

    @classmethod
    def __less_wheels__(cls):
        cls.wheels -= 1

if __name__ == '__main__':
    Car.__more_wheels__()
    m3 = Car('Bmw', 'M3')
    print('Brand={}, Model={}, Wheels={}'.format(m3.brand, m3.model, Car.wheels))

```

[60]:

```

Brand=Bmw, Model=M3, Wheels=5

```

3.21.10 @staticmethod

Methods annotated with `@staticmethod` can't access or modify the class state (i.e., class attributes). They are present in a class because it makes sense for these methods to be present in class but do not actually interact with the class. This annotation is generally used to create utility classes (e.g. math).

[61]:

```

class Car:
    def __init__(self, brand, model, speed=0):

```

```

    self.brand = brand
    self.model = model
    self.speed = speed

@staticmethod
def within_limits(speed, maxspeed):
    return speed < maxspeed

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3')
    print(Car.within_limits(75, 90))

```

[61]: True

3.21.11 String representations

`__repr__()`, and `__str__()` return a string representation of the object.

If `__repr__()` is defined, and `__str__()` is not, the object will behave as though `__str__()=__repr__()`. This means that almost every object you implement should have a functional `__repr__()` method that's usable for understanding the object.

Implementing `__str__()` is optional: do that if you need a “pretty print” functionality (for example, used by a report generator). `__repr__()` is internally called by the `repr()` built-in function.

```

[11]: class Car:
    def __init__(self, brand, model, speed=0):
        self.brand = brand
        self.model = model
        self.speed = speed

    def __repr__(self):
        return 'Brand={}, Model={}, Speed={}'.format(self.brand, self.model, self.
→speed)

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3', 120)
    print(m3)
    print(repr(m3))

```

[11]: Brand=Bmw, Model=M3, Speed=120
Brand=Bmw, Model=M3, Speed=120

```

[62]: class Car:
    def __init__(self, brand, model, speed=0):
        self.brand = brand
        self.model = model
        self.speed = speed

    def __repr__(self):
        return 'Brand={}, Model={}, Speed={}'.format(self.brand, self.model, self.
→speed)

    def __str__(self):

```

```

        return 'This is a {} model {} going at {}km/h.'.format(self.brand, self.model, self.speed)

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3', 120)
    print(repr(m3))
    print(m3)

```

[62]: Brand=Bmw, Model=M3, Speed=120
This is a Bmw model M3 going at 120km/h.

3.21.12 self.__dict__

A simple and straightforward way for overriding the `__repr__` method is to return the `__dict__` attribute which is a dictionary listing all the attributes associated with their values.

```

[14]: class Car:
        def __init__(self, brand, model, speed=0):
            self.brand = brand
            self.model = model
            self.speed = speed

        def __repr__(self):
            return str(self.__dict__)

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3', 120)
    print(m3)

```

[14]: {'brand': 'Bmw', 'model': 'M3', 'speed': 120}

3.21.13 One class, many objects

```

[12]: import random
import pprint

class Car:
    def __init__(self, brand, model, speed=0):
        self.brand = brand
        self.model = model
        self.speed = speed

    def __repr__(self):
        return str(self.__dict__)

if __name__ == '__main__':
    cars = [
        Car('BMW', 'M3'),
        Car('Fiat', 'Punto'),
        Car('Porsche', 'GT3'),
        Car('Lancia', 'Beta')
    ]

```

```
pprint pprint(cars)
```

```
[12]: [ {'brand': 'BMW', 'model': 'M3', 'speed': 0},  
       {'brand': 'Fiat', 'model': 'Punto', 'speed': 0},  
       {'brand': 'Porsche', 'model': 'GT3', 'speed': 0},  
       {'brand': 'Lancia', 'model': 'Beta', 'speed': 0}]
```

3.21.14 Docstrings

Docstring is a short for documentation string. Python docstrings are the string literals that appear right after the definition of a function, method, class, or module. Triple quotes are used. The docstring is available as the `__doc__` attribute.

```
[14]: class Car:  
    """A simple class for representing a car  
    with a brand, a model name, and a speed  
    """  
    def __init__(self, brand, model, speed=0):  
        self.brand = brand  
        self.model = model  
        self.speed = speed  
  
    if __name__ == '__main__':  
        m3 = Car('Bmw', 'M3')  
        print(Car.__doc__)  
        print(m3.__class__.__doc__)  
        print(m3.__doc__)
```

```
[14]: A simple class for representing a car  
with a brand, a model name, and a speed
```

```
A simple class for representing a car  
with a brand, a model name, and a speed
```

```
A simple class for representing a car  
with a brand, a model name, and a speed
```

3.22 Encapsulation, Inheritance, Polymorphism

3.22.1 Encapsulation

Encapsulation allows to restrict access to methods and variables. This prevents data from unwanted modifications. In Python, we denote private attributes using underscores as the prefix such as `self._varname` or `self.__varname`. Getters and setters have to be used along with this approach.

```
[17]: class Car:  
    def __init__(self, brand, model, licence):  
        self._brand = brand  
        self._model = model  
        self._licence = licence  
  
    def get_brand(self):  
        return self._brand
```

```

def set_brand(self, brand):
    self._brand = brand

def get_model(self):
    return self._model

def set_model(self, model):
    self._model = model

def get_licence(self):
    return self._licence

def set_licence(self, licence):
    if len(licence) != 7:
        raise ValueError("licence must be LLNNNLL")
    if not all(x.isalpha() for x in licence[0:2]):
        raise ValueError("licence must be LLNNNLL")
    if not all(x.isalpha() for x in licence[-2:]):
        raise ValueError("licence must be LLNNNLL")
    if not all(x.isnumeric() for x in licence[2:5]):
        raise ValueError("licence must be LLNNNLL")
    self._licence = licence

if __name__ == '__main__':
    # warning: the constructor do not apply controls!!
    m3 = Car('Bmw', 'M3', 'ABCDEFG')

    # the set_licence method works fine
    m3.set_licence('AA333TT')

    # warning: should not be done but still working
    m3._licence = 'fake!'

    print(m3.get_licence())

```

[17]: fake!

```

[19]: # This version implements controls inside the constructor as well
class Car:
    def __init__(self, brand, model, licence):
        self.set_brand(brand)
        self.set_model(model)
        self.set_licence(licence)

    def get_brand(self):
        return self._brand

    def set_brand(self, brand):
        self._brand = brand

    def get_model(self):
        return self._model

```

```

def set_model(self, model):
    self._model = model

def get_licence(self):
    return self._licence

def set_licence(self, licence):
    if len(licence) != 7:
        raise ValueError("licence must be LLNNNLL")
    if not all(x.isalpha() for x in licence[0:2]):
        raise ValueError("licence must be LLNNNLL")
    if not all(x.isalpha() for x in licence[-2:]):
        raise ValueError("licence must be LLNNNLL")
    if not all(x.isnumeric() for x in licence[2:5]):
        raise ValueError("licence must be LLNNNLL")
    self._licence = licence

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3', 'GY433WE')
    m3.set_licence('FY335YT')
    print(m3.get_licence())

# warning: should not be done but still working
m3._licence = 'fake!'

```

[19]: FY335YT

The use of getters and setters is discouraged in Python because they break the external interface of the class. Referring either to obj.brand or obj.get_brand() requires updating external dependencies. Alternatively, if encapsulation is required, *properties* can be used. Properties allows for encapsulation while maintaining intact the external interface of the class.

```

[20]: class Car:
    def __init__(self, brand, model, licence):
        self.brand = brand
        self.model = model
        self.licence = licence

    @property
    def licence(self):
        print('property.getter')
        return self._licence

    @licence.setter
    def licence(self, value):
        print('property.setter')
        if len(value) != 7:
            raise ValueError("licence must be LLNNNLL")
        if not all(isinstance(x, str) for x in value[0:2]):
            raise ValueError("licence must be LLNNNLL")
        if not all(isinstance(x, str) for x in value[-2:]):
            raise ValueError("licence must be LLNNNLL")
        if not all(x.isnumeric() for x in value[2:5]):

```

```

        raise ValueError("licence must be LLNNNLL")
    self._licence = value

@licence.deleter
def licence(self):
    print('property.deleter')
    del self._licence

if __name__ == '__main__':
    m3 = Car('Bmw', 'M3', 'GY455AI')
    m3.licence = 'FY335YT'
    m3.licence

```

[20]: property.setter
property.setter
property.getter

3.22.2 Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class. The class which get inherited is called *base class* or *parent class*. The class which inherits the other class is called *child class* or *derived class*. Derived classes inherit all attributes and methods from a base class. Furthermore they can:

- * add attributes and methods
- * redefine existing methods

Python offers a *super()* function which allows us to access the super class. The *super()* function is usually used in constructurs of derived classes for initializing the inherited portion of the object.

[20]: # base class

```

class Car:
    def __init__(self, brand, model, speed=0):
        self.brand = brand
        self.model = model
        self.speed = speed

    def speed_up(self):
        self.speed += 1

    def speed_down(self):
        self.speed -= 1

    def __repr__(self):
        return str(self.__dict__)

# derived class
class ECar(Car):
    def __init__(self, brand, model, speed=0, battery_level=0):
        super().__init__(brand, model, speed)
        self.battery_level = battery_level

    def charge(self):
        self.battery_level += 1

    def discharge(self):
        self.battery_level -= 1

```

```

if __name__ == '__main__':
    tsla = ECar('Tesla', 'ModelX')
    tsla.speed_up()
    tsla.charge()
    print(tsla)

```

[20]: {'brand': 'Tesla', 'model': 'ModelX', 'speed': 1, 'battery_level': 1}

3.22.3 Multilevel Inheritance

We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python. *In multilevel inheritance, features of both the base class and the derived class are inherited into the new derived class.*

```

[21]: class Base:
        pass

class Derived1(Base):
        pass

class Derived2(Derived1):
        pass

```

3.22.4 Multiple Inheritance

A class can be derived from more than one base class in Python, similarly to C++. This is called multiple inheritance. *In multiple inheritance, the features of all the base classes are inherited into the derived class.* The syntax for multiple inheritance is similar to single inheritance. (In Python non c'è il concetto di interfaccia e di classe astratta)

```

[22]: class Base1:
        pass

class Base2:
        pass

class MultiDerived(Base1, Base2):
        pass

```

3.22.5 Method Resolution Order

Every class in Python is derived from the *object* class. It is the most base type in Python. Technically, all classes, either built-in or user-defined, are derived classes and all objects are instances of the object class.

```

[23]: # issubclass compares two types
print(issubclass(float, object))
print(issubclass(str, object))

# isinstance compares an object and a type
print(isinstance(5.5, object))
print(isinstance("Hello", object))

```

```
[23]: True  
True  
True  
True
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in *depth-first, left-right fashion* without searching the same class twice. In the above example of MultiDerived class the search order is [MultiDerived, Base1, Base2, object]. This order is also called linearization of MultiDerived class and the set of rules used to find this order is called Method Resolution Order (MRO).

MRO of a class can be viewed as the `__mro__` attribute or the `mro()` method. The former returns a tuple while the latter returns a list. More detailed examples can be found [here](#).

```
[24]: MultiDerived.__mro__
```

```
[24]: (__main__.MultiDerived, __main__.Base1, __main__.Base2, object)
```

```
[25]: MultiDerived.mro()
```

```
[25]: [__main__.MultiDerived, __main__.Base1, __main__.Base2, object]
```

3.22.6 Polymorphism

Using inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that has been inherited from the parent class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as *method overriding*. Method overriding eventually leads to polymorphic behaviours (same method behaving in different ways depending on the object it is actually called on).

```
[76]: # base class  
class Car:  
    def __init__(self, brand, model, speed=0):  
        self.brand = brand  
        self.model = model  
        self.speed = speed  
  
    def speed_up(self):  
        self.speed += 1  
  
    def speed_down(self):  
        self.speed -= 1  
  
    def __repr__(self):  
        return str(self.__dict__)  
  
# derived class  
class ECar(Car):  
    def __init__(self, brand, model, speed=0, battery_level=0):  
        super().__init__(brand, model, speed)  
        self.battery_level = battery_level  
  
    # overridden methods  
    def speed_up(self):  
        self.speed += 2
```

```

def speed_down(self):
    self.speed -= 2

# additional methods
def charge(self):
    self.battery_level += 1

def discharge(self):
    self.battery_level -= 1

if __name__ == '__main__':
    cars = [Car('BMW', 'M3', 20), ECar('Tesla', 'ModelX', 20)]
    for car in cars:
        car.speed_up()
    print(cars)

```

[76]: [<{'brand': 'BMW', 'model': 'M3', 'speed': 21}, {'brand': 'Tesla', 'model': 'ModelX', 'speed': 22, 'battery_level': 0}]

3.23 Informal Interfaces

In certain circumstances, you may not need the strict rules of a formal interface. Python's dynamic nature allows to implement informal interfaces. *An informal interface is a class that defines methods that can be overridden, but there's no strict enforcement.*

```

[27]: class InformalCarInterface:
        def speed_up(self):
            pass

        def speed_down(self):
            pass

```

InformalCarInterface defines the two methods `speed_up()` and `speed_down()`. These methods are defined but not implemented. The implementation will occur once you create concrete classes that inherit from InformalCarInterface.

To use your interface, you must create a concrete class. A concrete class is a subclass of the interface that provides an implementation of the interface's methods.

Such informal interfaces are fine for small projects where only a few developers are working on the source code. However, as projects get larger and teams grow, this could lead to developers spending countless hours looking for hard-to-find logic errors in the codebase!

```

[28]: class Car(InformalCarInterface):
        def speed_up(self):
            pass

        def speed_down(self):
            pass

```

```

[29]: class ECar(InformalCarInterface):
        def speed_up(self):
            pass

```

```
def speed_down(self):
    pass
```

3.24 Sorting user-defined objects

3.24.1 Using sort() and sorted()

Both `list.sort()` and `sorted()` have a `key` parameter to specify a function (or other callable) to be called on each list element prior to making comparisons.

The value of the key parameter should be a function (or other callable) that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

```
[21]: # sorting objects
class Car:
    def __init__(self, brand, model, speed=0):
        self.brand = brand
        self.model = model
        self.speed = speed

    def __repr__(self):
        return str(self.__dict__)

cars = [
    Car('BMW', 'M2', 200),
    Car('Rimac', 'Concept One', 300),
    Car('Fiat', '500E', 100),
]

# using sort
cars.sort(key=lambda car: car.speed, reverse=True)
print(cars)

# using sorted (list is not modified)
print(sorted(cars, key=lambda car: car.brand))
```

```
[21]: [{'brand': 'Rimac', 'model': 'Concept One', 'speed': 300}, {'brand': 'BMW', 'model': 'M2', 'speed': 200}, {'brand': 'Fiat', 'model': '500E', 'speed': 100}]
[{'brand': 'BMW', 'model': 'M2', 'speed': 200}, {'brand': 'Fiat', 'model': '500E', 'speed': 100}, {'brand': 'Rimac', 'model': 'Concept One', 'speed': 300}]
```

3.24.2 The operator module

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The `operator` module has `itemgetter()`, `attrgetter()`, and a `methodcaller()` function. The `operator` module functions allows multiple levels of sorting.

```
[31]: import operator
print(sorted(cars, key=operator.attrgetter('brand')))
```

```
[31]: [{'brand': 'BMW', 'model': 'M2', 'speed': 200}, {'brand': 'Fiat', 'model': '500E', 'speed': 100}, {'brand': 'Rimac', 'model': 'Concept One', 'speed': 300}]
```

```
[32]: import operator
print(sorted(cars, key=operator.attrgetter('brand', 'speed')))
```

```
[32]: [{'brand': 'BMW', 'model': 'M2', 'speed': 200}, {'brand': 'Fiat', 'model': '500E', 'speed': 100}, {'brand': 'Rimac', 'model': 'Concept One', 'speed': 300}]
```

3.25 Modules and classes

3.25.1 Modules

Python allows developers to save classes in another file and import them. This has the advantage of isolating your classes into files that can be used in different programs. *As you use your classes repeatedly, the classes become more reliable over time.* When you save a class into a separate file, that file is called a *module*. You can have any number of classes in a single module.

Modules should have *short, lowercase names*. If you want to have a space in the module name, use an underscore. *Class names* should be written in *CamelCase*, with an initial capital letter and any new word capitalized. There should be no underscores in your class names.

```
[22]: # Save as car.py
class Car:
    def __init__(self, brand, model, speed=0):
        self.brand = brand
        self.model = model
        self.speed = speed

    def speed_up(self):
        self.speed += 1

    def speed_down(self):
        self.speed -= 1

    def __repr__(self):
        return str(self.__dict__)

class ECar(Car):
    def __init__(self, brand, model, speed=0, battery_level=0):
        super().__init__(brand, model, speed)
        self.battery_level = battery_level

    # overridden methods
    def speed_up(self):
        self.speed += 2

    def speed_down(self):
        self.speed -= 2

    # additional methods
    def charge(self):
        self.battery_level += 1

    def discharge(self):
        self.battery_level -= 1
```

```
[34]: from resources.car import Car, ECar

car_names = [('BMW', 'M3'), ('Porsche', 'GT3'), ('Lancia', 'Beta')]
cars = [ Car(brand, model, 0) for brand, model in car_names ]
for car in cars:
    print(car)

car_names = [('Tesla', 'Model X'), ('Rimac', 'Concept Two'), ('Volvo', 'Polestar 1')]
cars = [ ECar(brand, model, 0, 100) for brand, model in car_names ]
for car in cars:
    print(car)
```

```
[34]: {'brand': 'BMW', 'model': 'M3', 'speed': 0}
{'brand': 'Porsche', 'model': 'GT3', 'speed': 0}
{'brand': 'Lancia', 'model': 'Beta', 'speed': 0}
{'brand': 'Tesla', 'model': 'Model X', 'speed': 0, 'battery_level': 100}
{'brand': 'Rimac', 'model': 'Concept Two', 'speed': 0, 'battery_level': 100}
{'brand': 'Volvo', 'model': 'Polestar 1', 'speed': 0, 'battery_level': 100}
```

3.25.2 Importing classes

There are several ways to import modules and classes, and each has its own merits.

from module_name import ClassName The first one is straightforward, and is used quite commonly. It allows you to use the class names directly in your program, so you have very clean and readable code. This can be a problem, however, if the names of the classes you are importing conflict with names that have already been used.

```
[35]: from resources.car import Car, ECar

c0 = Car('BMW', 'M2')
print(c0)

c1 = ECar('Rimac', 'Concept One')
print(c1)
```

```
[35]: {'brand': 'BMW', 'model': 'M2', 'speed': 0}
{'brand': 'Rimac', 'model': 'Concept One', 'speed': 0, 'battery_level': 0}
```

import module_name The second one prevents name conflicts. However, it requires to use the module name each time a class or a function is used.

```
[36]: import resources.car

c0 = resources.car.Car('BMW', 'M2')
print(c0)

c1 = resources.car.ESCar('Rimac', 'Concept One')
print(c1)
```

```
[36]: {'brand': 'BMW', 'model': 'M2', 'speed': 0}
{'brand': 'Rimac', 'model': 'Concept One', 'speed': 0, 'battery_level': 0}
```

import module_name as alias When you are importing a module, you are free to choose any name you want for the module. This approach is often used to shorten the name of the module, so you don't have to type

a long module name before each class name that you want to use. Beware, do not shorten names too much because it might hamper readability.

```
[37]: import resources.car as c

c0 = c.Car('BMW', 'M2')
print(c0)

c1 = c.ECar('Rimac', 'Concept One')
print(c1)
```

```
[37]: {'brand': 'BMW', 'model': 'M2', 'speed': 0}
{'brand': 'Rimac', 'model': 'Concept One', 'speed': 0, 'battery_level': 0}
```

*from module_name import ** This is not recommended, for a couple reasons. First of all, you may have no idea what all the names of the classes and functions in a module are (possible naming conflicts). Also, you may be importing way more code into your program than you need. If you really need all the functions and classes from a module, just import the module and use the `module_name.ClassName` syntax in your program.

```
[38]: from resources.car import *

c0 = Car('BMW', 'M2')
print(c0)

c1 = ECar('Rimac', 'Concept One')
print(c1)
```

```
[38]: {'brand': 'BMW', 'model': 'M2', 'speed': 0}
{'brand': 'Rimac', 'model': 'Concept One', 'speed': 0, 'battery_level': 0}
```

3.25.3 Importing functions

You can use modules to store a set of functions you want available in different programs as well, even if those functions are not attached to any class. To do this, you save the functions into a file, and then import that file just as you saw in the last section.

```
[39]: # Save as multiplying.py
def double(x):
    return 2*x

def triple(x):
    return 3*x

def quadruple(x):
    return 4*x
```

```
[40]: from resources.multiplying import double, triple

print(double(5))
print(triple(5))
```

```
[40]: 10
15
```

```
[41]: import resources.multiplying

print(resources.multiplying.double(5))
print(resources.multiplying.triple(5))
```

```
[41]: 10
15
```

```
[42]: import resources.multiplying as m

print(m.double(5))
print(m.triple(5))
```

```
[42]: 10
15
```

```
[43]: from resources.multiplying import *

print(double(5))
print(triple(5))
```

```
[43]: 10
15
```

3.25.4 PEP8 guidelines

Imports should always be placed at the top of the file. This lets anyone who works with your program see what modules are required for the program to work. Your import statements should be in a predictable order:

- The first imports should be *standard Python modules* such as `sys`, `os`, and `math`.
- The second set of imports should be *third-party libraries* such as `pygame` and `requests`.

The names of modules should be on separate lines:

```
[44]: # ok
import sys
import os

# not ok
import sys, os
```

The names of classes can be on the same line:

```
[45]: from resources.car import Car, ECar
```

3.26 NumPy

3.26.1 What is NumPy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library providing a multi-dimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. This guide summarizes the official [documentation](#).

3.26.2 Why is NumPy Fast?

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just *behind the scenes* in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

- Vectorized code is more concise and easier to read.
- Vectorized code results in more *Pythonic* code. Without vectorization, code would be littered with inefficient and difficult to read for loops.
- Fewer lines of code generally means fewer bugs.
- The code more closely resembles standard mathematical notation (making it easier to code mathematical constructs).

3.26.3 Array initialization

3.26.4 Features

At the core of the NumPy package, is the *ndarray* object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:
* NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
* The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
* NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using built-in sequences

3.26.5 Constructor

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of non-negative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension. We can initialize numpy arrays from nested Python lists, and access elements using square brackets. You can read about other methods of array creation in the [documentation](#).

```
[5]: import numpy as np

a = np.array([1, 2, 3])                      # Create a rank 1 array
print(type(a))                                # Prints "<class 'numpy.ndarray'>"
print(a)                                       # Prints "(3,)"
print(a.shape)                                 # Prints "(3,)""

b = np.array([[1,2,3],[4,5,6]])               # Create a rank 2 array
print(type(b))                                # Prints "<class 'numpy.ndarray'>"
print(b)                                       # Prints "[[1 2 3]
                                                [4 5 6]]"
print(b.shape)                                 # Prints "(2, 3)"
```

```
[5]: <class 'numpy.ndarray'>
[1 2 3]
(3,)
<class 'numpy.ndarray'>
[[1 2 3]
 [4 5 6]]
(2, 3)
```

3.26.6 Alternative constructors

```
[17]: import numpy as np  
a = np.zeros((2,3)) # Create an array of all zeros  
print(a)
```

```
[17]: [[0. 0. 0.]  
 [0. 0. 0.]]
```

```
[16]: b = np.ones((2,3)) # Create an array of all ones  
print(b)
```

```
[16]: [[1. 1. 1.]  
 [1. 1. 1.]]
```

```
[18]: c = np.full((2,3), 7) # Create a constant array  
print(c)
```

```
[18]: [[7 7 7]  
 [7 7 7]]
```

```
[21]: d = np.eye(3) # Create an identity matrix  
print(d)
```

```
[21]: [[1. 0. 0.]  
 [0. 1. 0.]  
 [0. 0. 1.]]
```

```
[23]: e = np.random.random((2,3)) # Create an array filled with random values  
print(e)
```

```
[23]: [[0.4977616 0.42545686 0.40178286]  
 [0.08753894 0.5372359 0.53117326]]
```

```
[25]: f = np.arange( 0, 240, 5 )  
print(f)
```

```
[25]: [ 0  5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85  
 90 95 100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175  
 180 185 190 195 200 205 210 215 220 225 230 235]
```

```
[26]: # number of items might be unpredictable  
g = np.arange(0.24, 0.76, 0.09)  
print(g)
```

```
[26]: [0.24 0.33 0.42 0.51 0.6 0.69]
```

```
[28]: # number of items known a priori (third parameter)  
h = np.linspace(0.24, 0.76, 6)  
print(h)
```

```
[28]: [0.24 0.344 0.448 0.552 0.656 0.76 ]
```

```
[10]: # Create an empty array with the same shape as x
i = np.zeros_like(h)
print(i)
```

```
[10]: [0. 0. 0. 0. 0.]
```

```
[11]: # Create an empty array with the same shape as x
l = np.ones_like(h)
print(l)
```

```
[11]: [1. 1. 1. 1. 1.]
```

3.26.7 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. You can read all about numpy datatypes in the [documentation](#).

```
[32]: import numpy as np

x = np.array([1, 2])          # Let numpy choose the datatype
print(x.dtype)                # Prints "int64"

x = np.array([1.0, 2.0])       # Let numpy choose the datatype
print(x.dtype)                # Prints "float64"

x = np.array([1.6, 2.1], dtype=np.int64)  # Force a particular datatype
print(x.dtype)                  # Prints "int64"
print(x)
```

```
[32]: int64
float64
int64
[1 2]
```

3.27 Array attributes

- `ndarray.ndim` the number of axes (dimensions) of the array.
- `ndarray.shape` the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension.
- `ndarray.size` the total number of elements of the array. This is equal to the product of the elements of `shape`.
- `ndarray.dtype` an object describing the type of the elements in the array. One can create or specify `dtype`'s using standard Python types. Additionally NumPy provides types of its own such as `numpy.int32`, `numpy.int16`, and `numpy.float64`.
- `ndarray.itemsize` the size in bytes of each element of the array. For example, an array of elements of type `float64` has itemsize 8 (=64/8), while one of type `complex32` has itemsize 4 (=32/8). It is equivalent to `ndarray.dtype`.

```
[13]: import numpy as np
a = np.random.random((5, 5))
print(a.ndim)
print(a.shape)
```

```
print(a.size)
print(a.dtype)
print(a.itemsize)
```

```
[13]: 2
(5, 5)
25
float64
8
```

3.27.1 Indexing and slicing

3.27.2 Integer indexing

Integer array indexing: when you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using data from another array. If you want to know more you should read the [documentation](#).

```
[16]: import numpy as np

# Create the following rank 2 array with shape (3, 2)
# [[1 2]
#  [3 4]
#  [5 6]]
a = np.array([[1,2], [3, 4], [5, 6]])

# Single elements
print(a[0,1])
print(a[1,1])

# Multiple elements (2)
print(a[[0, 1], [1, 1]])  # Prints "[2 4]"

# Multiple elements (3)
print(a[[0, 1, 2], [1, 1, 0]])  # Prints "[2 4 5]"

# Second element of each row
print(a[[0, 1, 2], np.array([1])])  # Prints "[2 4 6]"

# Second element of each row
print(a[np.arange(3), np.array([1])])  # Prints "[2 4 6]"
```

```
[16]: 2
4
[2 4]
[2 4 5]
[2 4 6]
[2 4 6]
```

3.27.3 Boolean indexing

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
[35]: import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])
print(a)

print(a > 2)      # Prints "[[False False]
                   #           [ True  True]
                   #           [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
print(a[a > 2])  # Prints "[3 4 5 6]"
```

```
[35]: [[1 2]
       [3 4]
       [5 6]]
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
```

3.27.4 Mutating elements with indexing

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix.

```
[38]: import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9]])
print(a)

# Create an array of indices
b = np.array([0, 2, 0])

# Select one element from each row of a using the indices in b
print(a[np.arange(3), b])  # Prints "[ 1  6  7]"

# Mutate one element from each row of a using the indices in b
a[np.arange(3), b] = -1
print(a)

a[a < 0] = -2
print(a)
```

```
[38]: [[1 2 3]
       [4 5 6]
       [7 8 9]]
[[1 6 7]
 [[-1  2  3]
  [ 4  5 -1]
  [-1  8  9]]
 [[-2  2  3]
  [ 4  5 -2]
  [-2  8  9]]]
```

3.27.5 Slicing

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array.

```
[14]: import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
# [ 5  6  7  8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
# [6 7]]
b = a[:2, 1:3]
print(b)

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])    # Prints "2"
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])    # Prints "77"
```

```
[14]: [[2 3]
       [6 7]]
2
77
```

You can also *mix integer indexing with slice indexing*. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

```
[15]: import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
# [ 5  6  7  8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10 ] (3,)"
```

```
print(col_r2, col_r2.shape) # Prints "[[ 2]
# [ 6]
# [10]] (3, 1)"
```

```
[15]: [5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[ 2  6 10] (3,)
[[ 2]
 [ 6]
[10]] (3, 1)
```

3.28 Array math

3.28.1 Basic functions

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module. You can find the full list of mathematical functions provided by numpy in the [documentation](#).

```
[19]: import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
print(x)
print(y)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
#print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
#print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
#print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2           0.33333333]
#  [ 0.42857143   0.5         ]]
print(x / y)
#print(np.divide(x, y))
```

```
[19]: [[1. 2.]
 [3. 4.]]
[[5. 6.]
 [7. 8.]]
```

```
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
[[0.2          0.33333333]
 [0.42857143  0.5        ]]
```

3.28.2 Dot product

Dot function is for computing inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. *dot* is available both as a standalone function and as an instance method of array objects.

```
[20]: import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11,12])

# Vector / vector product; scalar value
print(v.dot(w))
#print(np.dot(v, w))

# Matrix / vector product; rank 1 array
print(x.dot(v))
#print(np.dot(x, v))

# Matrix / matrix product; rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
#print(np.dot(x, y))
```

```
[20]: 219
[29 67]
[[19 22]
 [43 50]]
```

3.28.3 Computations on arrays

Numpy provides many useful functions for performing computations on arrays.

```
[21]: import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
print(x)

print(np.sqrt(x))           # Elementwise square root; produces the array

print(np.sum(x))           # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))   # Compute sum of each column; prints "[4 6]"
```

```

print(np.sum(x, axis=1)) # Compute sum of each row; prints "[3 7]"
print(np.prod(x))        # Compute sum of all elements; prints "24"
print(np.prod(x, axis=0)) # Compute sum of each column; prints "[3 8]"
print(np.prod(x, axis=1)) # Compute sum of each row; prints "[2 12]"

```

```
[21]: [[1. 2.]
       [3. 4.]]
      [[1.          1.41421356]
       [1.73205081 2.          ]]
      10.0
      [4. 6.]
      [3. 7.]
      24.0
      [3. 8.]
      [ 2. 12.]
```

3.28.4 Reshaping

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operations is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object.

```
[22]: import numpy as np

x = np.array([[1,2], [3,4], [5,6]])
print(x)
print(x.shape)

# Transpose x
print(x.T)
print(x.T.shape)
```

```
[22]: [[1 2]
       [3 4]
       [5 6]]
      (3, 2)
      [[1 3 5]
       [2 4 6]]
      (2, 3)
```

Note that taking the transpose of a rank 1 array does nothing

```
[23]: import numpy as np

v = np.array([1,2,3])

print(v)      # Prints "[1 2 3]"
print(v.shape)

print(v.T)   # Prints "[1 2 3]"
print(v.T.shape)
```

```
[23]: [1 2 3]
(3,)
[1 2 3]
(3,)
```

Using `reshape()` will give a new shape to an array without changing the data. Just remember that when you use the reshape method, the array you want to produce needs to have the same number of elements as the original array. If you start with an array with 12 elements, you'll need to make sure that your new array also has a total of 12 elements.

```
[24]: import numpy as np

x = np.arange(24)
print(x)

# two-dimensions
print(x.reshape(6, 4))

# another two-dimensions arrangment
print(x.reshape(2, 12))

# three-dimensions
print(x.reshape(2, 3, 4))
```

```
[24]: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
[[ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]]
[[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

 [[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]]
```

Numpy provides many more functions for manipulating arrays; you can see the full list in the [documentation](#).