primo

February 19, 2020

1 Gli oggetti in python

Tutto è un oggetto in python. Questi oggetti hanno sempre 3 elementi: * identità * tipologia * valore

1.1 Identità

Informazione numerica di tipo immutabile, rimane fino alla morte dell'oggetto.

1.2 Tipologia

Categoria che determina la natura di un oggetto; quali valori possono essere assegnati ad un oggetto, altre caratteristiche dell'oggetto. Gli oggetti che fanno parte di un certo tipo si chiamano ISTANZE di quell'oggetto.

2 Valore

Il valore è un dato o un insieme di dati mantenuto all'interno dell'oggetto, quando questi lo prevede. Un oggetto che può modificare il suo valore durante il suo ciclo di vita si dice Mutabile, diversamente si definisce Immutabile. Tale caratteristica è data dal *tipo* di oggetto.

2.1 Literal

Forma *letterale* di un oggetto: python possiede una serie di dati che sono predefiniti nel linguaggio e la cui definizione del valore dipende dal modo letterale con cui sono stati inseriti. Se scrivo 20 python lo interpreterà come un integer perchè questo è il modo letterale con cui lo abbiamo inserito; tra apici invece definiremo un oggetto stringa, senza bisogno di dichiararlo!

3 Variabili

Per riferirci agli oggetti non useremo mai la sua identità, ma lo assegneremo un *nome* ad un oggetto.

~~~ a = 20 mia_lista = [1, 2, 3]

Esistono delle regole per definire i nomi validi: * lettere o numeri o caratteri Unicode o underscore * NON può iniziare con un numero * NON può essere una **parola riservata**

I nomi associati agli oggetti vengono definiti *variabili*. In python una variabile non è una variabile, ma un nome che punta ad una variabile. Non è fortemente tipizzata.
~~~

~~~ a = 20 a = “andrea”

Questa cosa implica la **condivisione** dell’ID.

```
a = 20
b = a
```

In questo caso a e b puntano allo stesso oggetto, ma non sono lo stesso oggetto!

Molto importante è il concetto di *reference count*. Tutti gli oggetti in python hanno un contatore di riferimenti: questo implica che l’oggetto venga distrutto SOLO quando il suo reference count è pari a 0. Il *Garbage Collector* ripulisce tutto e libera la memoria (tutto in runtime).

Il *Garbage Collector* ripulisce tutto e libera la memoria (tutto in runtime).

4 Callable Objects

Gli oggetti chiamabili sono quelli in cui usiamo argomenti all’interno delle parentesi tonde. Esempio le funzioni!

La funzione `print` ne è un esempio!

Altre funzioni utili sono `id(oggetto)` oppure `type(oggetto)`.

5 Gli Attributi

Gli oggetti in python hanno un valore ed una identità; possono avere anche una serie di *attributi*.

Gli attributi sono degli oggetti riferiti da un oggetto perchè ne specificano ulteriori caratteristiche (possono essere dati o funzioni). Nel caso in cui siano funzioni vengono chiamati metodi.

Per richiamarli si una il punto: nome oggetto.nome attributo!

Quando l’attributo è una funzione dobbiamo invocarlo essendo un oggetto Callable.

[5]:
```
"python".upper()
```

[5]:
```
'PYTHON'
```

[7]:
```
x = "python"
x.upper()
```

[7]:
```
'PYTHON'
```

6 BASIC data types

- tipi numerici (anche boolean)
- stringhe
- operatori *espressioni

None è un basic data type che ha una sola istanza... None
~~~

6.1 tipi numerici

- integer
- floating poin
- boolean Sono tutti Immutabili!

`a = 3` ed `a = 4`

a non ha modificato l'oggetto, ma punta ad un oggetto diverso, prima il numero 3 e poi il numero 4!

Dalla versione 3.6 possiamo dividere le cifre di un numero attraverso l'underscore: 10_000_000.

Possiamo usare 3 literal per esprimere un numero: * Binario – 0b10011001 * ottale 0o1635 * esadecimale 0x1F8A

I boolean sono sottoinsiemi di interi che hanno solo 2 valori: True o False; True=1 e False=0.

I Floating point vengono introdotti con un punto decimale di separazione.

Possiamo usare anche l'annotazione esponenziale!

[13]:
```
2e4
```

[13]:
```
20000.0
```

[14]:
```
2e-4
```

[14]:
```
0.0002
```

Naturalmente python supporta anche i numeri **immaginari**.

6.2 Stringhe

Le STRINGHE sono sequenze di elementi: una sequenza indica un insieme *ordinato* di elementi. Tale caratteristica ci permetterà di intervenire facilmente sugli elementi che compongono la stringa.

Ogno elemento di una stringa deve appartenere al set UNICODE.

Le stringhe in python sono oggetti immutabili, possiamo copiare parte di una stringa in un'altra stringa.

Possiamo usare apici singoli o doppi, purchè siano simmetricamente definiti.

Una stringa vuota non ha caratteri ma è comunque esistente come oggetto.

Possiamo definire stringhe su multilinea usando 3 apici singoli o 3 apici doppi.

""" questa può essere una stringa su diverse linee"""

[ ]:
```
#Abbiamo diverse tipologie di Escape (caratteri non stampabili):
#* \n andare a capo
#* \t tabulare
#* \\ inseriamo la backslash nella stringa
#* \' inseriamo l'apice nella stringa
```

```
#* \" inseriamo un doppio apice nella stringa
```

6.3 caso particolare: le F strings

- modulo formatting (ormai deprecata)
- metodo format associato alle stringhe
- f-string introdotto da python 3

[21]:
```
titolo = "Isola Misteriosa"
autore = "Giulio Verne"
```

[23]:
```
print(f"Titolo: {titolo}, Autore: {autore}")
```

```
Titolo: Isola Misteriosa, Autore: Giulio Verne
```

Questa operazione viene detta **String Interpolation**

[24]:
```
print(f"Titolo: {titolo.upper()}, Autore: {autore}")
```

```
Titolo: ISOLA MISTERIOSA, Autore: Giulio Verne
```

PEP 498 – Literal String Interpolation

7 Espressioni ed Operatori

Le operazioni eseguono operazioni su degli elementi; tali elementi possono essere di tipo diverso.

I più comuni sono gli operatori aritmetici: * addizione + * sottrazione - * moltiplicazione * * divisione floating point / * divisione intera // solo parte intera del risultato * modulo % definisce il resto di una divisione * esponenziale ** * meno unario -

operatori di assegnamento: * = assegnazione di variabile * += scorciatoia per a += equivale ad a= a+b * -=, *=, /=, //=, %=, **=

operatori di confronto (ritornano sempre un valore Booleano): * < * > * == * != * <= * >=

Attenzione alla *precedenza*: usiamo le parentesi per modificarne il comportamento.

Operatori logici (restituisco valori Booleani): * and * or * notm

A	B	A AND B
True	True	True
False	False	False
True	False	False
False	False	False

A	B	A OR B
True	True	True
False	False	False

A	B	A OR B
True	False	True
False	False	True

NOT è un operatore UNARIO: ritorna il contrario dell'espressione valutata.

In python *tutti* gli oggetti hanno un valore di verità implicitamente assegnato: sono tutti FALSE * None * False * Zero * una sequenza vuota * un dizionario vuoto in tutti gli altri casi il valore è TRUE…

Operatori su *Sequenze*

[39]:
```
s = "python programming"
```

[40]:
```
s[0]
```

[40]: 'p'

[41]:
```
s[0:6]
```

[41]: 'python'

[42]:
```
s[-1]
```

[42]: 'g'

[43]:
```
s[-2:]
```

[43]: 'ng'

[44]:
```
s[:6]
```

[44]: 'python'

[45]:
```
s[0:6:3]
```

[45]: 'ph'

~~~ [start:stop:step]

Concatenazione utilizzando l'operatore +

[48]:
```
s = "python"
```

[49]:
```
len(s)
```

[49]: 6
~~~

[50]:
```
min(s)
```

[50]:
```
'h'
```

[51]:
```
max(s)
```

[51]:
```
'y'
```

[ ]:
```

```

8 Conversioni di tipo

Interi: * usiamo la funzione int() * le stringhe diventano numeri interi (se sono numeri)

Float: * usiamo la funzione float() * stesso discorso per gli interi

Stringa: * usiamo la funzione str() * usata per i numeri che diventano elementi di stringa

Booleano: * usiamo la funzione bool() * lo applichiamo a tutti gli elementi

9 Strutture di dati in python

10 Liste, Tuple, Dizionari, Set

Una sequenza è generalmente definita come un insieme ordinato di elementi indicizzati numericamente attraverso la loro posizione; l'indice parte sempre da 0

Liste e Tuple caratteristiche: * elementi di tipo qualunche * le liste sono mutabili * le tuple sono immutabili

10.1 Liste

Sequenza di elementi qualunque, mutabili. L'instaza riguarda una classe predefinita che si chiama list.

[63]:
```
myList1 = []
myList = [10, 20, 30]
lista_vuota = list()
```

[65]:
```
myList[-1]
```

[65]:
```
30
```

Valgono per lo slicing tutti i discorsi visti per le stringhe!

[107]:
```
nido = [1, 2, ["primo", "secondo", "terzo"]]
```

[108]:
```
nido[2][1]
```

[108]:
```
'secondo'
```

modifichiamo la lista:

[109]:
```
nido[0] = "andrea"
nido[2][2] = "paese"
```

[110]:
```
nido
```

[110]:
```
['andrea', 2, ['primo', 'secondo', 'paese']]
```

[111]:
```
len(nido)
```

[111]:
```
3
```

[97]:
```
nido.insert(1, "inserito")
```

[104]:
```
nido
```

[104]:
```
['andrea',
 'inserito',
 2,
 'ultimo',
 ['primo', 'secondo', 'ancora ultimo', 'paese']]
```

[112]:
```
nido.insert(-1, "ultimo")
```

[113]:
```
nido
```

[113]:
```
['andrea', 2, 'ultimo', ['primo', 'secondo', 'paese']]
```

[114]:
```
nido[-1].append("ancora ultimo")
```

[115]:
```
nido
```

[115]:
```
['andrea', 2, 'ultimo', ['primo', 'secondo', 'paese', 'ancora ultimo']]
```

[118]:
```
del nido[0]
```

[119]:
```
nido
```

[119]:
```
[2, 'ultimo', ['primo', 'secondo', 'paese', 'ancora ultimo']]
```

[120]:
```
"ultimo" in nido
```

[120]:
```
True
```

[122]:
```
"paese" in nido[-1]
```

[122]:
```
True
```

[123]:
```
"paese" in nido
```

[123]:
```
False
```

10.2 Due nomi, una lista

[124]:
```
lista1 = [1, 2, 3]
lista2 = lista1
lista2[1] = 60
```

[125]:
```
lista1
```

[125]:
```
[1, 60, 3]
```

[126]:
```
lista2
```

[126]:
```
[1, 60, 3]
```

Puntando alla stessa lista la variazione di una determina la variazione dell'altra... se questo non fosse il comportamento che vogliamo ritorna utile l'attributo COPY...

[128]:
```
lista1 = [1, 2, 3]
lista2 = lista1.copy()
```

[129]:
```
lista2[0] = 60
```

[130]:
```
lista1
```

[130]:
```
[1, 2, 3]
```

[131]:
```
lista2
```

[131]:
```
[60, 2, 3]
```

10.3 Tuple

La tupla è una sequenza *immutabile*

Il tipo è la classe tuple()

[135]:
```
medaglie = ()
medaglie = tuple()
elenco = "oro", "argento", "bronzo"
```

[136]:
```
print(type(elenco))
```

```
<class 'tuple'>
```

[137]:
```
print(elenco)
```

```
('oro', 'argento', 'bronzo')
```

il Tuple *UNpacking*

[142]:
```
primo, secondo, terzo = elenco
```

[143]:
```
primo
```

[143]:
```
'oro'
```

[144]:
```
secondo
```

[144]:
```
'argento'
```

[145]:
```
terzo
```

[145]:
```
'bronzo'
```

10.4 Dizionario

Nei dizionari l'ordine degli elementi non è definito, si usano delle chiavi univoche che vengono associate ai rispettivi valori.

Le chiavi devono essere univoche. Gli elementi dei dizionari sono mutabili attraverso i metodi appartenenti ai dizionari.

La classe dei dizionari è la dict().

[147]:
```
mioDizionario = {}
mioDizionario = dict()
```

[148]:
```
myDict = {
    "primo": 10,
    "secondo": 20,
    "terzo": 30
}
```

[149]:
```
myDict["quarto"] = 40
```

[150]:
```
myDict
```

[150]:
```
{'primo': 10, 'secondo': 20, 'terzo': 30, 'quarto': 40}
```

[151]:
```
del myDict["secondo"]
```

[152]:
```
myDict
```

[152]:
```
{'primo': 10, 'terzo': 30, 'quarto': 40}
```

~~~ myDict.clear() # vuota il dizionario

[153]:
```
"terzo" in myDict
```

[153]:
```
True
```

[155]:
```
myDict2 = myDict.copy()
```

[156]:
```
myDict2
```

[156]:
```
{'primo': 10, 'terzo': 30, 'quarto': 40}
```

[157]:
```
myDict2["quinto"] = 50
```

[158]:
```
myDict2
```

[158]:
```
{'primo': 10, 'terzo': 30, 'quarto': 40, 'quinto': 50}
```

[159]:
```
myDict
```

[159]:
```
{'primo': 10, 'terzo': 30, 'quarto': 40}
```

[160]:
```
d1 = {10: "a"}
d2 = {20: "b"}
```

[161]:
```
d3 = {}
```

[162]:
```
d3.update(d1)
```

[163]:
```
d3
```

[163]:
```
{10: 'a'}
```

[164]:
```
d3.update(d2)
```

[165]:
```
d3
```

[165]:
```
{10: 'a', 20: 'b'}
```

10.5 SET

Si tratta di insiemi, con tutte le operazioni matematiche che abbiamo per gli insiemi…
~~~

[16]:
```
mySet = set()
```

[17]:
```
mySet1 = {1, 2, 3, 4}
```

Un set è un oggetto **mutabile**:

[18]:
```
mySet1.add(19)
print(mySet1)
```

```
{1, 2, 3, 4, 19}
```

Esiste anche la possibilità di usare dei Frozen SET...immutabili

[21]:
```
gelo = frozenset([12, 23, 43])
```

[25]:
```
23 in gelo
```

[25]:
```
True
```

[26]:
```
2 in mySet1
```

[26]:
```
True
```

Operazioni peculiari sugli insiemi: intersezione ed unione.

[27]:
```
set1 = {10,20,30,40}
set2 = {30,40,50,60}
```

[29]:
```
set1 & set2 # intersezione
```

[29]:
```
{30, 40}
```

[30]:
```
set1.union(set2) # unione
```

[30]:
```
{10, 20, 30, 40, 50, 60}
```

[31]:
```
set1 | set2 # somma logica (unione)
```

[31]:
```
{10, 20, 30, 40, 50, 60}
```

[32]:
```
set1 - set2 # differenza logica
```

[32]:
```
{10, 20}
```

[33]:
```
set1 ^ set2 # or esclusivo: al primo o al secondo ma non entrambi
```

[33]:
```
{10, 20, 50, 60}
```

11 Strutture di codice

11.1 linee di codice e blocchi di codice

Abbiamo: * linee logiche di codice (che vede python) * linee fisiche (che scriviamo nel listato)

[39]:
```
s = "python \
programming \
language"
print(s) # 4 linee fisiche ma una sola linea logica
```

```
python programming language
```

Un blocco di codice è un insieme di linee di codice raggruppate!

“‘ inp = input(“inserisci un numero:”) x = int(inp) if x < 10: s = “numero minore di 10” print(s)

11.2 Statement: istruzioni del programma

Operazioni che si richiede al codice python di eseguire:

```
s = "andrea"
```

Statement semplici e composti

“‘ if x < 10: s = “numero minore di 10” elif x == 10: s = “numero 10” else: s = ”numero maggiore di 10 print(s)

Struttura di uno statement composto: * contiene una o più clausole * ogni clausola contiene una parola chiave di python detta HEADER e termina con il carattere : * suite che contiene un blocco di codice, indentato rispetto all’HEADER

11.3 Lo statement IF

Viene utilizzato per definire una esecuzione condizionata del codice (azioni differenti rispetto al test di verità).

La clausola ELSE viene eseguita solo se tutte le precedenti danno come risultato FALSE; possiamo ometterla per fare in modo che nel caso di tutti FALSE non venga eseguito nulla!

[44]:
```
x = 10
if x > 11:
    print("il numero è maggiore di 11") # manca la clausola else
```

11.4 Lo statement WHILE

La else è facoltativa ma poco utilizzata

Se è presente la clausola ELSE si entra nel ciclo sempre: se l’espressione risulterà FALSE ci entreremo direttamente senza passare dal True.

“‘ x = 0 while x < 3: print(x) x += 1

Spesso si vuole eseguire un loop senza sapere quando la condizione sarà verificata; in questo caso utilizzeremo *break* per uscire dal loop.

Diverso è lo statement *continue* il quale non esce dal ciclo ma lo fà ripartire dall'inizio!

[2]:
```
while True:
    x = input("inserire una stringa ")
    if x == "stop":
        break
    if x < "b":
        continue
    print(x) # tutte le stringhe < b non vengono stampate perchè si ritorna al␣
↪primo if
```

```
sas
ere
era
```

11.5 Ciclo FOR

"' for target in iterator: suite else: suite

L'iterazione termina quando tutti gli elementi di iterator sono stati processati. La clausola else, poco utilizzata, solo se tutti gli elementi sono stati iterati.

[19]:
```
myList = [1, 3, 4, 5]
for i in myList:
    if i <= 4:
        print(i)
    else:
        break
else:
    print("ho iterato tutti gli elementi")
```

```
1
3
4
```

[20]:
```
for i in myList:
    if i <= 9:
        print(i)
    else:
        break
else:
    print("ho iterato tutti gli elementi")
```

```
1
3
4
```

```
5
ho iterato tutti gli elementi
```

L'iterazione sui dizionari assegna alla variabile TARGET le KEY. "' for i in myDict…

```
for i in myDict.items()
```

restituisce delle tuple chiave valore.

CONTINUE nel ciclo for avanza l'iterazione saltando l'elemento indicato:

[5]:
```
for i in [1, 3, 5, 7]:
    if i == 5:
        continue
    print(i)
```

```
1
3
7
```

11.6 La funzione RANGE

range(start, stop, step)… lo STOP è SEMPRE escluso!

11.7 List comprehension

Si tratta di un argomento avanzato ma molto importante perchè si tratta di uno strumento molto potente!

[expression for item in iterable if condition]

[9]:
```
miaLista = [1, 3, 5, 7, 8, 9, 12]
```

[10]:
```
l =[x for x in miaLista if x > 5]
print(l)
```

```
[7, 8, 9, 12]
```

[11]:
```
s = [x * 10 for x in miaLista]
print(s)
```

```
[10, 30, 50, 70, 80, 90, 120]
```

[12]:
```
p = [x*x for x in miaLista if x % 2 == 0]
print(p)
```

```
[64, 144]
```

11.8 Dict comprehension

[57]:
```
nomi = {10: "andrea", 20: "mario", 30: "anna", 40: "giuseppe"}
```

[58]:
```
mioDizionario = {k:v for k,v in nomi.items() if k > 20}
print(mioDizionario)
```

```
{30: 'anna', 40: 'giuseppe'}
```

[68]:
```
nomiDict = {k:v for k,v in nomi.items() if k <= 20}
print(nomiDict)
```

```
{10: 'andrea', 20: 'mario'}
```

[71]:
```
cambiato= {(k+30):v for k,v in nomi.items()}
print(cambiato)
```

```
{40: 'andrea', 50: 'mario', 60: 'anna', 70: 'giuseppe'}
```

12 Set comprehension

Ricordiamo che il set non può contenere elementi DUPLICATI.

[76]:
```
lista = [ 1, 5, 7, 8, 9]
```

[77]:
```
nuovo = {x for x in lista if x < 7}
print(nuovo)
```

```
{1, 5}
```

13 le Funzioni

Insieme di istruzioni con un nome, eseguita a richiesta in altre parti del programma. Una funzione è un OGGETTO, Callable Object (oggetti chiamabili).

Due cose si possono fare con una funzione: * definirla * chiamarla

13.1 Definizione di una funzione

“‘ def function_name(parameters): statements

Quando passiamo i valori ai parametri, questi si chiamano argomenti!

13.2 Parametri della funzione

[1]:
```
def myFunc(a, b):
    print(a, b)  # parametri posizionali, in chiamata sono necessari␣
↪nell'ordine dato
```

[2]:
```
def myFunc(a, b):
    print(a, b) # keyword possiamo passarli nell'ordine che vogliamo...

myFunc(b=10, a=30)
```

```
30 10
```

[3]:
```
myFunc(10, b=50) # prima i posizionali poi le keyword
```

```
10 50
```

[6]:
```
def myFunc(a, b, c=4):
    print(a, b)  #parametri opzionali possono mancare nella chiamata
```

[7]:
```
def myFunc(*args):
    print(args) # il nome è convenzionale, inserisce i valori in una tupla,␣
 ↪variabile
```

[8]:
```
def myFunc(a, b, *args):
    print(a, b, args) # i posizionali sempre prima
```

[9]:
```
myFunc(1,5, 6,8,9)
```

```
1 5 (6, 8, 9)
```

[10]:
```
def myFunc(**kwargs):
    print(kwargs)
```

[15]:
```
myFunc(andrea=1, mario=2)
```

```
{'andrea': 1, 'mario': 2}
```

13.3 Lo statement *RETURN*

[16]:
```
def sum(a, b):
    return a + b  # return può non essere seguito da una espressione
```

Se non indichiamo nessuna espressione, il ritorno sarà None…

[18]:
```
def sum(a, b):
    c = a + b # torna al chiamante il valore None...
```

[ ]:
```
## Chiamare le funzioni
```

`function_name(arguments)`

I valori vengono passati come riferimento!

[20]:
```
def myFunc(x):
    x = 10
    print(x)
```

[22]:
```
y = 20
myFunc(y)
print(y) # l'oggetto originario è immutabile e non ha modificato il suo␣
 ↪contenuto
```

```
10
20
```

[26]:
```
def myFunc(x):
    x["func"] = 10
```

[28]:
```
d = {"a": 5}
myFunc(d)
print(d) # d è un dizionario, mutabile, ergo è stato effettivamente mutato
```

```
{'a': 5, 'func': 10}
```

13.4 Funzioni come OGGETTI

[29]:
```
def sum(x, y):
    print(x + y)
```

[30]:
```
sum(10, 5) # invochiamo la funzione
```

```
15
```

[31]:
```
sum # chiamiamo l'oggetto funzione, istanza della classe function
```

[31]:
```
<function __main__.sum(x, y)>
```

13.5 Usare gli oggetti funzione

[41]:
```
def outer(x, y):
    def sum(a, b):
        return a + b
    print(sum(x, y))       # funzione nidificata
```

[42]:
```
outer(10,5)
```

```
15
```

[43]:
```
sum(19,1)
```

```
20
```

[44]:
```
def outer():
    def inner(a, b):
        print(a + b)
    return inner
```

[46]:
```
outer()
```

[46]:
```
<function __main__.outer.<locals>.inner(a, b)>
```

[47]:
```
f = outer()
f(10, 5)
```

```
15
```

[53]:
```
def somma(a,b):
    print(a+b)

def sottrai(a,b):
    print(a-b)

def myFunc(f, x, y):
    f(x,y)
```

[54]:
```
myFunc(somma, 10, 5)
myFunc(sottrai, 10, 5) # ho passato la funzione come argomento oggetto funzione
```

```
15
5
```

13.6 Namespace e Scope

Namspace:

1. mappatura di nomi ad oggetti
2. namespace multipli, a Runtime
3. organizzati in una gerarchia
4. cicli di vita differenti

Scope:

area di codice che determona il namespace da utilizzare per la risoluzione dei nomi.

Namespace: gerarchia LEGB: * local * enclosed * global * built-in

13.7 local scope

livello interno di una funzione, viene creato una volta chiamata la funzione, rimosso una volta ottenuto il ritorno della funzione stessa.

E' il primo punto dove python cerca di risolvere i nomi!

13.8 enclosed

nel caso di funzioni nidificate è il secondo punto in cui viene cercata la risoluzione de nomi.m

“‘ def outer(x): y = 20 def inner(): print(x+y)

13.9 global namespace

Tutti i nomi definiti dal livello del sorgente quando i nomi sono indicati al di fuori di tutte le funzioni.

“‘ x = 20 def miaFunc(y): print(x + y)

13.10 Built-In namespace (predefinito)

Definto direttamente dall’ambiente di runtime di python, contiene, ad esempio, la risoluzione print, list, dict, tuple, …

13.11 Global e Nonlocal

Servono ad alterare la gerarchia standard nella ricerca della risoluzione dei nomi da parte di python.

Variabile Hiding:

se utilizziamo un nome di variabile già presente a livelli di scope più alti, questa avrà la precedenza sulle altre, in un certo senso nascondendole!

Per alterare questo comportamento utilizziamo la parola *global*

“‘ x = 100 def myFunc(): global x x = 20 print(x)

In questo caso abbiamo alterato la variabile globale!

Nonlocal cerca la variabile nel namespace della funzione madre (in funzioni nidificate).m

[12]:
```
def outer():
    y = 20
    def inner():
        nonlocal y
        y = 50
        print("variabile in inner %s" %(y))
    inner()
    print("variabile in outer %s" %(y))
```

[13]:
```
outer()
```

```
variabile in inner 50
variabile in outer 50
```

14 Function Decorator (decoratori di funzione)

Un decoratore di una funzione è una funzione che prende in input una funzione, la decora con altri contenuti e restituisce il nuovo valore.

Utilizzo: modificare il comportamento di una funzione senza alterarne il codice sorgente!

[24]:
```
def myDecorator(f):
    def decorator():
        print("ho decorato")
        f()
    return decorator
```

[25]:
```
def myFunc():
    print("la funzione myFunc")
```

[26]:
```
decorata = myDecorator(myFunc)
```

[27]:
```
decorata()
```

```
ho decorato
la funzione myFunc
```

[29]:
```
## con il decoratore...
```

[30]:
```
def myDecorator(f):
    def decorator():
        print("ho decorato con il decoratore")
        f()
    return decorator
```

[31]:
```
@myDecorator
def myFunc():
    print("la funzione myFunc")
```

[32]:
```
myFunc()
```

```
ho decorato con il decoratore
la funzione myFunc
```

[33]:
```
def mioDecoratore(func_destinazione):
    def wrapper(*args):
        print("elementi prima della funzione")
        func_destinazione()
        print("elementi dopo la funzione")
    return wrapper
```

Nel caso in cui volessimo eseguire una funzione decorate SENZA decoratore, dopo aver importato

`from undecorated import undecorated (pip install)`

useremo la sintassi `undecorated(funzione)(parametri)`

Quindi se chiameremo la funzione decorata direttamente otterremo la decorazione come previsto, se useremo undecorated… otterremo la funzione NON decorata!

15 Lambda function

Un'espressione che genera un oggetto funzione!

Lambda ritorna una funzione *anonima*

`lambda arg1, arg2, argN : expression (con gli argomenti)`

[48]:
```
risultato = lambda x,y : x * y
```

[51]:
```
print("il valore richiesto è ", risultato(2,3))
```

```
il valore richiesto è  6
```

16 Object-Orientation

- definizione di classi
- creazione istanze di classi
- come strutturare le classi in gerarchie di generalizzazione

16.1 Classi ed Istanze

funzioni –> metodi di classe

16.2 Lo Statement Class

Un oggetto composto che serve a creare degli oggetti attraverso la sua istanziazione.

"' class Classname(base-classes): (base-classes sono le superclassi, le classi padre) statements

Nello statement ci saranno metodi ed attributi della classe…

[83]:
```
class MyClass: # la convenzione python prevede la lettera maiuscola per le␣
↪classi
    pass
```

Non abbiamo usato le parentesi tonde, indicando che la classe di orgine è la object, propria di python…

istanza di una classe:

```
myObj = MyClass()
```

16.3 Attributi di Classe

Gli attributi possono essere di classe o di istanza. * attributi di classe: condivisi da tutte le istanze della classe * attributi di istanza: propri di quella e solo quella istanza, non della classe

[1]:
```
class MyClass:
    myAttr = 10
```

[3]:
```
m1 = MyClass()
m2 = MyClass()
```

[6]:
```
m1.myAttr
```

[6]: 10

[7]:
```
m2.myAttr = 40
m2.myAttr
```

[7]: 40

[9]:
```
m2.attributo = 555
```

[12]:
```
print(m2.myAttr)
print(m2.attributo) # non è presente nella classe, diventa un attributo di istanza!
```

```
40
555
```

16.4 Metodi di istanza

[20]:
```
class MyClass:
    def myMethod(self):
        print(id(self)) # metodo di classe e self che rappresenta l'istanza invocata dal metodo
```

[21]:
```
m1= MyClass()
m2 = MyClass()
```

[23]:
```
m1.myMethod()
```

```
140538000774736
```

[24]:
```
m2.myMethod()
```

```
140538000774544
```

[25]:
```
MyClass.myMethod(m1)
```

```
140538000774736
```

[26]:
```
MyClass.myMethod(m2)
```

```
140538000774544
```

16.5 Attributi di ISTANZA

[27]:
```
class MyClass:
    def setMessage(self, message):
        self.message = message
    def printMessage(self):
        print(self.message)
```

In questo codice c'è un problema: se chiamiamo in istanza direttamente il metodo printMessage, otterremo un errore in quanto non è stato settato il messaggio; per ovviare a questo problema dovremo definire dei metodi di inizializzazione.

16.6 il costruttore init

Viene chiamato sempre ed automaticamente ogni volta che una istanza di classe viene attivata.

[30]:
```
class MyClass:
    def __init__(self, message):
        self.message = message
    def printMessage(self):
        print(self.message)
```

[31]:
```
m1 = MyClass()
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-31-f7aebff7631d> in <module>
----> 1 m1 = MyClass()


TypeError: __init__() missing 1 required positional argument: 'message'
```

[32]:
```
m1 = MyClass("ciao")
```

[35]:
```
m1.printMessage()
```

```
ciao
```

16.7 Metodi di classe

Eseguiti non sulle istanze ma proprio sull'oggetto classe!

[43]:
```
class MyClass:
    counter = 0 #attributo della classe
    def __init__(self):
        MyClass.counter += 1
    @classmethod
    def istanze(cls): # cls indica l'oggetto classe
        print(cls.counter)
```

[44]:
```
m1 = MyClass()
m2 = MyClass()
m3 = MyClass()
```

[45]:
```
MyClass.istanze()
```

```
3
```

[ ]:
```

```

16.8 STATIC Methods

[47]:
```
class MyClass:
    @staticmethod
    def somma(a,b):
        return(a + b)
```

[48]:
```
s = MyClass.somma(10,5)
print(s)
```

```
15
```

Non si riferisce alle classi e nemmeno alle istanze!

17 Inheritance (ereditarietà)

[2]:
```
class BClass: #superclasse
    pass

class AClass(BClass):
    pass
```

La funzione isinstance “‘ m1 = AClass() isinstance(m1, AClass) True isinstance(m1, BClass) True

[3]:
```
m1 = AClass()
```

[4]:
```
isinstance(m1, BClass)
```

[4]:
```
True
```

17.1 Override

Possiamo ridefinire un attributo all'ìnterno di una sottoclasse

[7]:
```
class BClass:
    def setMessage(self, message):
        self.message = message
    def printMessage(self):
        print(self.message)

class AClass(BClass):
    def printMessage(self):
        print("AClass " + self.message)
```

[8]:
```
m1 = AClass()
m1.setMessage("andrea")
m1.printMessage()
```

```
AClass andrea
```

Questo metodo non è un metodo corretto, vediamo la procedura corretta con il costruttore. Infatti quando invochiamo la sottoclasse, se questa ha un suo costruttore viene usato ma, nella sintassi utilizzata, non viene considerato il costruttore della superclasse, viene sovrascritto il costruttore!

18 la funzione SUPER

[3]:
```
class BClass:
    def __init__(self, message):
        self.message = message
    def printMessage(self):
        print(self.message)
    def scatola(self):
        scatola = "BICHER"
        return scatola

class AClass(BClass):
    def __init__(self, message, valore):
        super().__init__(message)
        self.valore = valore
```

[4]:
```
m1 = AClass("andrea", 100)
```

[5]:
```
m1.printMessage()
```

```
andrea
```

[6]:
```
m1.valore
```

[6]:
```
100
```

non si usa solo per invocare init della superclasse, ma per accedere a tutto il contenuto della stessa:

[7]:
```
class CClass(BClass):
    def __init__(self, name):
        self.nome = name
        super().scatola()
```

[11]:
```
m2 = CClass("valerio")
```

[13]:
```
m2.scatola()
```

[13]:
```
'BICHER'
```

18.1 Properties

Information HIDING possibilità di rendere privati degli attributi che rappresentano dei dati, nascosti all'esterno della classe. i Metodi setter e getter settano e leggono gli attributi privati.

[20]:
```
class MyClass:
    def __init__(self, my_attr):
        self.priv_attr = my_attr
    def get_attr(self):
        return self.priv_attr
    def set_attr(self, attr):
        self.priv_attr = attr

    attr = property(get_attr, set_attr) # costruiamo una proprietà che nasconde␣
 ↪attr
```

[21]:
```
obj = MyClass("andrea")
obj.attr
```

[21]:
```
'andrea'
```

Gli attributi che iniziano con doppio underscore non sono accessibli al di fuori della classe:

[26]:
```
class MyClass:
    def __init__(self, my_attr):
        self.__priv_attr = my_attr
    def get_attr(self):
        return self.__priv_attr
    def set_attr(self, attr):
        self.__priv_attr = attr
```

```
    attr = property(get_attr, set_attr) # costruiamo una proprietà che nasconde␣
↪attr
```

[28]:
```
obj1 = MyClass("nascosto")
obj1.__private_attr
```

```
      ␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-28-5efacf79dd33> in <module>
         1 obj1 = MyClass("nascosto")
   ----> 2 obj1.__private_attr


       AttributeError: 'MyClass' object has no attribute '__private_attr'
```

[29]:
```
obj1._MyClass__priv_attr # accesso diretto al nostro attributo name Mangling
```

[29]:
```
'nascosto'
```

18.2 Property Decorators

Le proprietà forniscono un modo di personalizzare l'accesso agli attributi dell'istanza. Per crearli, si utilizza il decoratore @property messo prima del metodo. Il loro scopo è quello di definire attributi read-only (non possono essere modificati).

"‘ @property (decoratore del metodo getter) @name.setter (decoratore del metodo setter)

[54]:
```
class MyClass():
    def __init__(self, my_attr):
        self.__priv_attr = my_attr

    def metodoPrivato(self):
        print("Ciao") #questo metodo non può essere chiamato fuori dalla classe!

    @property
    def attr(self):
        return self.__priv_attr

    @attr.setter
    def attr(self, my_attr):
        self.__priv_attr = my_attr
```

[55]:
```
obj = MyClass("decorato")
obj.attr
```

[55]:
```
'decorato'
```

[56]:
```
obj.__metodoPrivato()
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-56-606cd4ed12cc> in <module>
----> 1 obj.__metodoPrivato()


AttributeError: 'MyClass' object has no attribute '__metodoPrivato'
```

[ ]:
```

```

19 Exceptions

Le eccezioni sono degli oggetti che appartengono ad una gerarchia base di python, ma possiamo anche crearne di nuove secondo le nostre necessità!

[16]:
```
def myFunc(a,b):
    return a // b
```

[17]:
```
myFunc(10,0)
```

```
␣
↪---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call␣
↪last)

<ipython-input-17-c0381f4b20fc> in <module>
----> 1 myFunc(10,0)


<ipython-input-16-e07cba645158> in myFunc(a, b)
  1 def myFunc(a,b):
----> 2     return a // b
```

```
ZeroDivisionError: integer division or modulo by zero
```

Viene elevata un'eccezione di divisione per ZERO!

Nessuno ha detto ha python come gestire questa eccezione e quindi viene invocato il messaggio standard (si tratta di un oggetto). Il runtime prima verifica se noi abbiamo definito un modo per gestire questa eccezione, se non lo trova risale di uno stack alla volta fino ad arrivare alla interruzione del programma mostrando l'errore connesso a questa eccezione.

Tutte le eccezioni sono istanze di una particolare classe sempre tutte sottoclassi di BaseException!

ZeroDivisionError ==> ArithmeticError ==> Exception ==> BaseException ==> object

19.1 lo Statement try/except (si tratta di uno statement composto)

"' try:

`suite`

except:

`suite`

[25]:
```
def myFunc(a,b):
    try:
        a // b
    except (ZeroDivisionError, ValueError):
        print("non posso dividere per zero")
    except IndexError:
        print("IndexError")
```

[24]:
```
myFunc(120,0)
```

```
non posso dividere per zero
```

[32]:
```
def myFunc(a,b):
    try:
        return a // b
    except ZeroDivisionError as e:
        print("Errore della funzione\n",e.args)
```

[33]:
```
myFunc(10,0)
```

```
Errore della funzione
 ('integer division or modulo by zero',)
```

La clausolo FINALLY viene usata per eseguire sempre, a prescindere dall'errore, una serie di istruzioni.

[35]:
```
def myFunc(a,b):
    try:
        a // b
    except ZeroDivisionError:
        print("Errore di divisione")
    finally:
        print("abbiamo provato ad eseguire la tua funzione")
```

[36]:
```
myFunc(10,6)
```

```
abbiamo provato ad eseguire la tua funzione
```

[37]:
```
myFunc(29,0)
```

```
Errore di divisione
abbiamo provato ad eseguire la tua funzione
```

Dopo tutte le clausole except possiamo eseguire una else (se tutto andrà bene verrà eseguita la clausola else). Potremmo usare anche una finally, ma in questo caso else deve essere posta PRIMA della finally.

[59]:
```
def myFunc(a,b):
    try:
        a // b
        risultato = True
    except ZeroDivisionError:
        print("Errore di divisione")
        risultato = False
    else:
        print("tutto a posto, abbiamo eseguito la funzione")

    finally:
        if risultato == True:
            print("siamo giunti alla fine eseguendo la tua funzione")
        else:
            print("non abbiamo potuto finire")
```

[60]:
```
myFunc(28,5)
```

```
tutto a posto, abbiamo eseguito la funzione
siamo giunti alla fine eseguendo la tua funzione
```

[61]:
```
myFunc(10,0)
```

```
Errore di divisione
non abbiamo potuto finire
```

19.2 Gli statement raise ed assert

raise si usa per sollevare esplicitamente una eccezione

La classe di eccezione dopo raise può essere omessa.

[62]:
```
for i in range(10):
    print(i)
    raise IndentationError("Errore nel loop")
```

```
0
```

```
        Traceback (most recent call last):


          File "/home/andrea/.local/lib/python3.7/site-packages/IPython/core/
↪interactiveshell.py", line 3319, in run_code
        exec(code_obj, self.user_global_ns, self.user_ns)


          File "<ipython-input-62-76e0d4921a86>", line 3, in <module>
        raise IndentationError("Errore nel loop")


          File "<string>", line unknown
    IndentationError: Errore nel loop
```

raise senza classe risolleva una except che precedentemente era stata intercettata.

[63]:
```
def myFunc(a,b):
    try:
        a // b
    except ZeroDivisionError:
        print("ERRORE")
        raise
```

[64]:
```
myFunc(129,0)
```

```
ERRORE
```

```
      ␣
↪---------------------------------------------------------------------------

        ZeroDivisionError                         Traceback (most recent call␣
↪last)
```

```
    <ipython-input-64-5540b34fef5d> in <module>
----> 1 myFunc(129,0)


    <ipython-input-63-f194659d205d> in myFunc(a, b)
      1 def myFunc(a,b):
      2     try:
----> 3         a // b
      4     except ZeroDivisionError:
      5         print("ERRORE")


    ZeroDivisionError: integer division or modulo by zero
```

assert expression, argument

Valutiamo una espressione e se falsa verrà elevata una eccezione con in aggiunta la stringa argument.

[67]:
```
x = 5
assert x == 0, "valore errato"
```

```
    ␣
↪---------------------------------------------------------------------------

    AssertionError                            Traceback (most recent call␣
↪last)

    <ipython-input-67-53d27c38eaec> in <module>
      1 x = 5
----> 2 assert x == 0, "valore errato"


    AssertionError: valore errato
```

[70]:
```
x = 10
y = 20
try:
    if x != y:
        raise # invoco un errore forzando la except!
    else:
        print("sono uguali")

except:
      print("si è verificato un errore")
```

```
si è verificato un errore
```

20 Ereditarietà multipla

[4]:
```
class BClass:
    def bFunc(self):
        print("sono in bFunc")

class CClass:
    def CFunc(self):
        print("sono in cFunc")
```

[5]:
```
class AClass(BClass, CClass):
    pass

a = AClass()
```

[7]:
```
a.bFunc()
```

```
sono in bFunc
```

[8]:
```
a.CFunc()
```

```
sono in cFunc
```

```
class Persona:
    def __init__(self, fname, lname):
        self.nome = fname
        self.cognome = lname

class Indirizzo:
    def __init__(self, via, paese):
        self.via = via
        self.paese = paese

class Utente(Persona, Indirizzo):
    def __init__(self, nome, cognome, via, paese):
        Persona.__init__(self, nome, cognome)
        Indirizzo.__init__(self, via, paese)

    def scheda(self):
        return f"""
        nome: {self.nome}
        cognome: {self.cognome}
        via: {self.via}
        paese: {self.paese}"""


io = Utente("andrea", "prestini", "BICHER", "Esine")
```

```
print(io.scheda())
```

Cosa accade se entrambe le classi hanno lo stesso attributo funzione?

MRO Method Resolution Order

L'attributo viene cercato prima nella sottoclasse stessa, poi nella prima classe presente nella gerarchia, poi la seconda, etc. come presenti nella dichiarazione della sottoclasse.

In estrema ratio l'ultima classe in cui ricerca sarà Object!

[ ]: