

Linguaggio python, variabili, operatori, espressioni

1

Linguaggio Istruzioni ed espressioni

- Nella prima lezione abbiamo visto che lo scopo del programmatore è definire un **algoritmo** in un **linguaggio interpretabile** dalla macchina.
- In python scriveremo gli **algoritmi** come sequenze di **istruzioni**.
- Ogni **istruzione** modificherà la memoria del calcolatore avvicinandoci al risultato finale (e.g. calcolo sqrt).
- Le istruzioni sono costituite da costrutti ed **espressioni**.
- Oggi studiamo come scrivere correttamente espressioni

2

Tipi di dato

- In python i dati hanno un tipo. Scriveremo in memoria delle informazioni come stringhe di bit secondo delle convenzioni.
- Ogni area di memoria sarà caratterizzata dal valore che vi è scritto e dal tipo di dato che rappresenta.
- I tipi sono matematicamente degli insiemi che definiscono quali valori sono rappresentabili nell'elaboratore tramite il tipo scelto.
- Tipi: `int`, `float`, `str`, `chr`, `bool`

3

Interi - `int`

- Si usano per rappresentare numeri naturali con e senza segno.
- In python hanno precisione arbitraria.

```
>>> a = 1 # questo è un valore intero (int) assegnato ad una variabile a
```

```
>>> a = -1 # questo è un valore intero negativo (int) assegnato ad una variabile a
```

4

Virgola mobile – valori rappresentabili

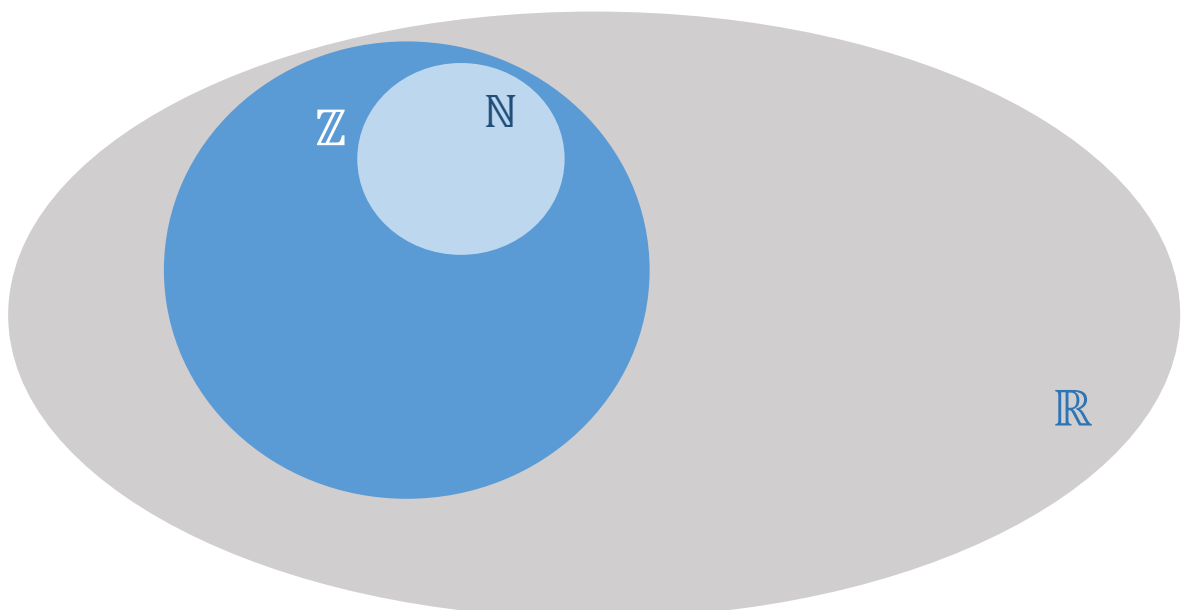
- Si usano per rappresentare i razionali li possiamo usare

`float` – si usano per rappresentare numeri su 64 bit

```
>>> a = 1.2 #questo è un float
```

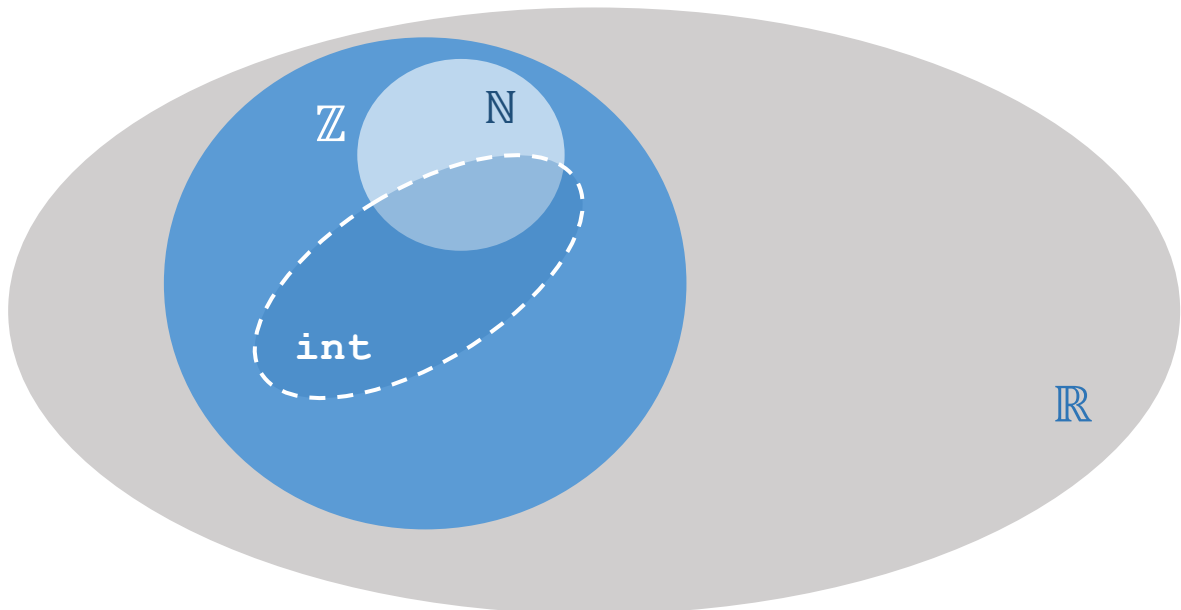
5

Relazione tra insiemi e tipi



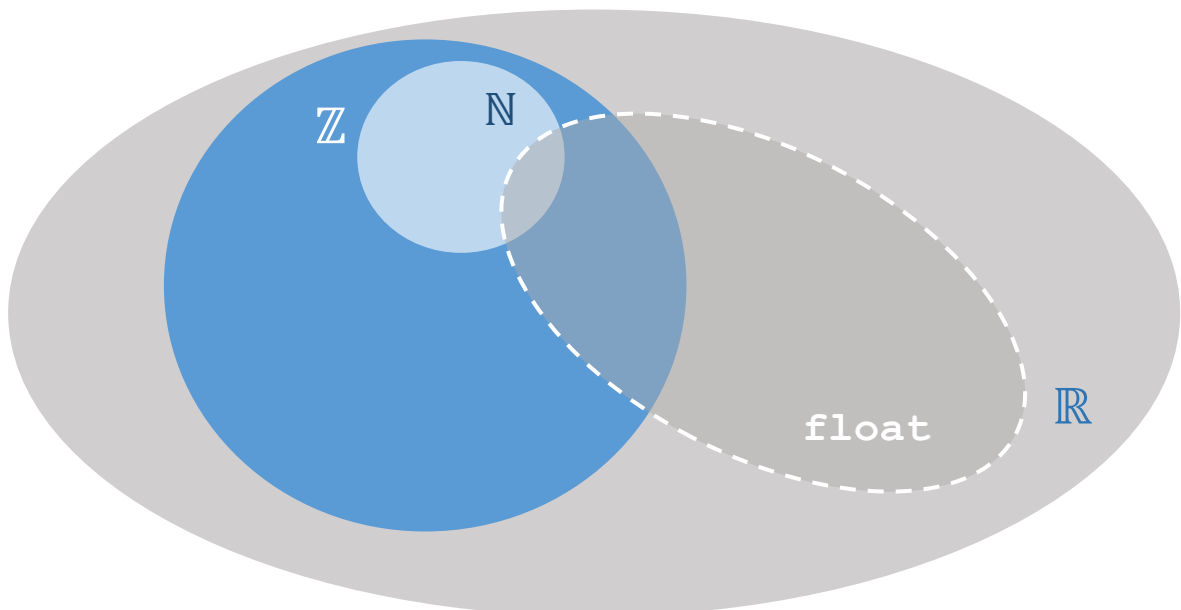
6

Relazione tra insiemi e tipi



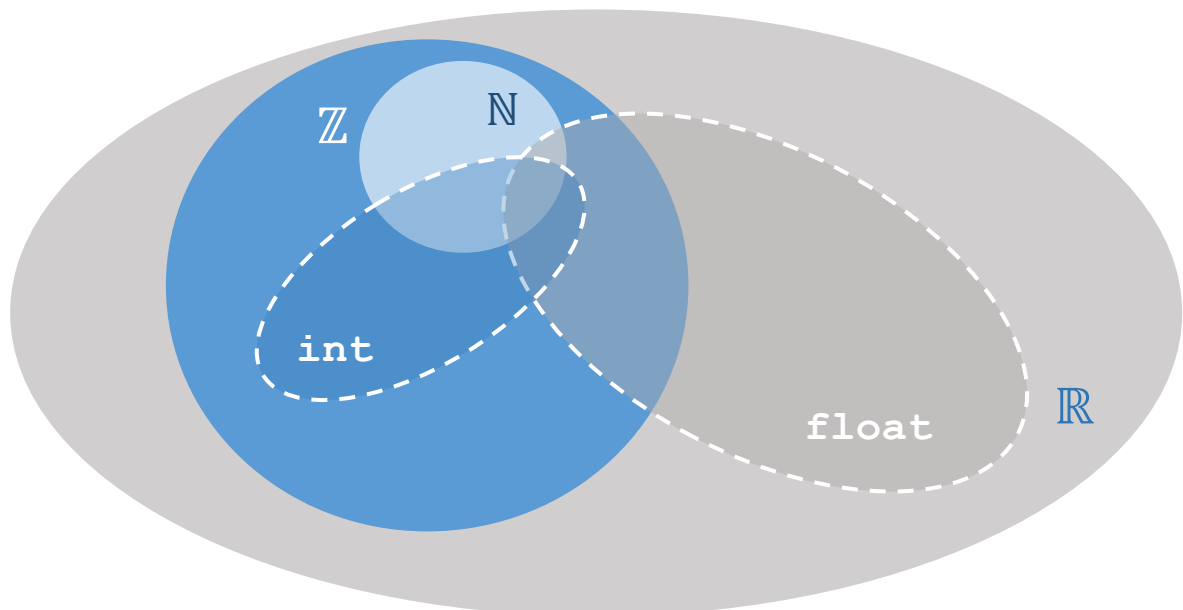
7

Relazione tra insiemi e tipi



8

Relazione tra insiemi e tipi



9

Caratteri - `chr`

- Si usano per rappresentare i caratteri usando la codifica ASCII
- Si usa un byte per rappresentarli si hanno quindi 255 valori

`chr` – si usa per rappresentare un carattere tramite un intero a 8bit

10

char – Valori rappresentabili

```
v = 97
c = chr (v)
print(c, v) #stampa a, 97
```

Tabella ASCII: American Standard Code for Information Interchange

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q

11

Stringhe - str

- Sono sequenza di caratteri
- Si definiscono usando apici singoli o doppi

```
c = 'ciao!' #una stringa
```

```
c = "ciao!" #una stringa
```

12

Sintassi

- Useremo spesso una notazione per descrivere la sintassi ammessa per un certo tipo di istruzione.
- Questo linguaggio si usa per definire la grammatica di altri linguaggi ed è detto EBNF: extended Backus-Naur form

`<non_terminale>:=<non_terminale> terminale;`

- Uso <> per indicare un simbolo che può espandersi

`<non_terminale>:= terminale1 | terminale2;`

- Uso | per indicare l'alternativa

`<non_terminale>:= [terminale opzionale] terminale | terminale;`

- Uso [] per indicare l'opzionalità di un simbolo (sia terminale che non terminale)

13

Costanti

- Una costante denota un valore che non si può modificare durante l'esecuzione di un programma.

```
10    #costante intera
10.1  #costante float
'c'   #costante str
```

- Le useremo assieme agli **operatori** e alle **variabili** per creare delle **espressioni**

14

Variabili

- Una variabile dà il nome ad una **locazione di memoria** in cui è memorizzato un valore di un certo **tipo**.
- In python i tipi delle variabili sono *inferiti* al momento dell'assegnazione
- Nel dubbio possiamo usare `type(variable)` per scoprire il tipo di una variabile
- Le variabili esistono da quando gli viene assegnato un valore per la prima volta

```
a = 10
type(a)
<class 'int'>
```

```
a = 10.0
type(a)
<class 'float'>
```

15

Variabili

- Le variabili esistono da quando gli viene assegnato un valore per la prima volta
- `<var_decl>:= <var_name> = <expr>`

```
a = 10
type(a)
<class 'int'>
```

```
a = 10.0
type(a)
<class 'float'>
```

16

Variabili

- Regole per `<var_name>`
- Nomi di variabile o identificatori non possono iniziare con un numero.
- Possono contenere tutti i caratteri alfanumerici compresi gli `_`

Esempi corretti:

```
time1 = 10
time_to_live = 1 #C style
HomeAddress = 'Via S. Marta 3' #JAVA style o CamelCase
_value = 2 #si può iniziare con _
```

Esempi errati:

```
time # non si può definire una variabile senza legarla ad un
valore
age-person # il - è ambiguo e essere scambiato per un operatore
12time = 10 #non si può iniziare una variabile con un numero
```

17

Variabili

- Buone regole per `< var_name >`
- Le variabili devono avere un nome evocativo. La stringa che indica la variabile deve descrivere il suo contenuto

Esempi che usano una buona convenzione:

```
time = 0
radius = 1.0
vehicle_mass = 1000.0
```

Esempi che **non usano** una buona convenzione:

```
x, y, z = 10, 20, 30 #a meno che non siano numeri generici
gianni, pinotto = 10, 20 #il vostro programma funzionerà...
pippo = 64; #ma rileggendolo non avrà senso...
```

18

Variabili

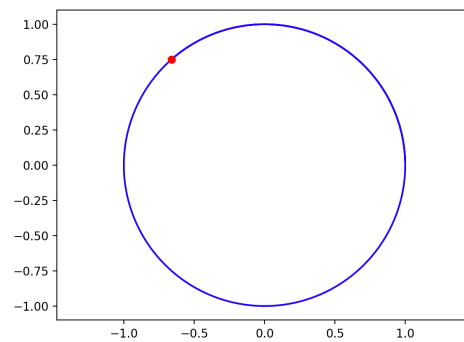
- Vogliamo scrivere un programma in grado di generare traiettorie di un moto circolare uniforme

```
import math
import matplotlib.pyplot as plt
```

$$x = R \cdot \cos(\theta)$$

$$y = R \cdot \sin(\theta)$$

```
radius = 1
theta = 0
X = []
Y = []
while True:
    theta += math.pi/100
    x = radius * math.cos(theta)
    y = radius * math.sin(theta)
    X.append(x)
    Y.append(y)
    if theta > 2*math.pi:
        plt.plot(X,Y,'b-',)
        plt.plot(x,y,'ro')
        plt.axis('equal')
        plt.show()
        plt.pause(0.01)
        plt.clf()
```



x=-0.422619 y=0.430952

19

Variabili

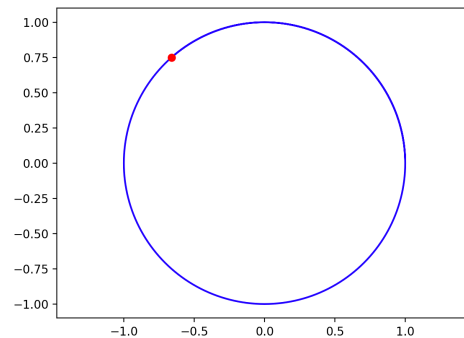
- Un programma identico dal punto di vista dell'elaboratore è però molto poco leggibile...

```
import math
import matplotlib.pyplot as plt
```

$$x = R \cdot \cos(\theta)$$

$$y = R \cdot \sin(\theta)$$

```
ciao = 1
mamma = 0
programmo = []
bene = []
while True:
    mamma += math.pi/100
    ma = ciao * math.cos(mamma)
    ehi = ciao * math.sin(mamma)
    programmo.append(ma)
    bene.append(ehi)
    if mamma > 2*math.pi:
        plt.plot(programmo,bene,'b-',)
        plt.plot(ma,ehi,'ro')
        plt.axis('equal')
        plt.show()
        plt.pause(0.01)
        plt.clf()
```



x=-0.422619 y=0.430952

20

Conversioni

- E' possibile, conveniente e spesso necessario convertire i dati da un tipo ad un altro
- Le conversioni possono avvenire in modo **implicito** o **esplicito**
- Il caso più semplice e frequente è la conversion da int a float

`a = 1` #questa è una variabile int

`b = a + 1.2` #la somma di un int e un float (1.2) restituisce un float

- b vale 2.2
- a viene prima convertito a float (`a = 1.0`)
- Successivamente viene applicata la somma tra float

21

Conversioni

- Il caso opposto si verifica se applichiamo un operatore **definito solo per int** come la divisione intera `//`

`a = 1.2` #questa è una variabile float

`b = a // 2` #la divisione intera tra un float e un int restituisce un int

- b vale 0
- a viene prima convertito in int
- Successivamente viene applicato l'operatore divisione intera

22

Espressioni

- Combinano **variabili** e **costanti** tramite **operatori**

`a` = 10 + 1

`a` = `a` + 1

`a` = (`b` + `c`) / 12.0

- Ogni istruzione termina con un a capo

24

Espressioni

- Combinano **variabili** e **costanti** tramite **operatori**

`a` = 10 + 1

`a` = `a` + 1

`a` = (`b` + `c`) / 12.0

- Ogni istruzione termina con un a capo

25

Espressioni

- Combinano **variabili** e **costanti** tramite **operatori**

`a = 10 + 1`

`a = a + 1`

`a = (b + c) / 12.0`

- Ogni istruzione termina con un a capo

26

Espressioni

- Semantica di ciascuna espressione è suddivisibile in:
 - Valore assunto dall'espressione
 - Side-effects* sulle variabili usate

Valore

- Ciascuna espressione assume un valore:

`1 + 1` #il valore assunto è 2

`a = 10`

`a + a` #il valore assunto è 20

27

Espressioni

- Semantica di ciascuna espressione è suddivisibile in:
 - Valore assunto dall'espressione
 - Side-effects* sulle variabili usate

Side Effects

- Alcune espressioni modificano il valore delle variabili che contengono
- ```
a = 10 #la variabile a viene modificata e vale 10
a = a + 1 #la variabile viene incrementata di 1
```

28

## Espressioni

- Operatori binari aritmetici: - + // \* %
- Si usano per creare espressioni di tipo aritmetico, la loro semantica è ovvia, salvo casi particolari.
- L'espressione `a+b` ha il valore della somma dei valori di a e b. Se ad esempio a = 1 e b = 2, scrivere `a+b` è come scrivere `3`.

```
a + 1 #la somma del valore di a e 1
a % 2 #il resto della divisione intera tra il
valore di a e 2
23 // 2 #la divisione intera tra 23 e 2, vale 11
23 / 2.0 #...vale 11.5
```

| Operatore | Funzione         |
|-----------|------------------|
| +         | Somma            |
| -         | Sottrazione      |
| *         | Moltiplicazione  |
| /         | Divisione        |
| //        | Divisione Intera |
| %         | Resto            |

29

## Espressioni

- Operatore assegnazione: =
- Si usa per assegnare un **valore** ad una **variabile**. E' un operatore binario. La sua semantica consiste nel variare il valore dell'operando **sinistro** sostituendovi **il valore** dell'operando **destro**.

```
a = 1 #d'ora in poi a vale 1
a = b + c #d'ora in poi a vale la somma di b e c
a = a + 1 #incrementa d 1 il valore di a
```

- **Non commutativo:  $a=b$  è diverso da  $b=a$**
- L'operando sinistro non può ovviamente essere una costante o un'espressione ma solo una variabile

30

## Espressioni

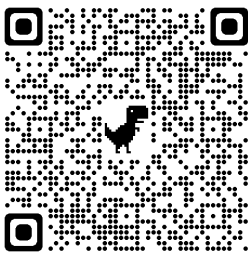
- **Non commutativo:  $a=b$  è diverso da  $b=a$**
- L'operando sinistro non può ovviamente essere una costante o un'espressione ma solo una variabile

- Esempio errore di assegnazione ad una costante

```
1 = a # <- ERRORE
```

- Se questa espressione fosse sintatticamente corretta (ma non lo è), la sua semantica sarebbe una cosa del tipo:

*Assegna al valore costante 1 il valore della variabile a!!!!*



QUIZ  
<https://forms.gle/LoJr9hEsZKnFjbTs6>

31

## Espressioni

- E' possibile combinare gli operatori aritmetici con l'operatore assegnazione:

`<var><op2>=<expr>`

è una forma contratta equivalente a

`<var> = <var><op2><expr>`

`x += 10 #x=x+10`

`x /= 2 #x=x/2`

`x *= a + 1 #x=x*(a+1)` prima si valuta la parte destra e poi si applica l'assegnazione eventualmente combinata.

32

## Espressioni

- Operatori binari relazionali: `<` `<=` `>` `>=` `==` `!=`
- Si usano per determinare la relazione tra due valori. Un'espressione relazionale ha valore `False` se falsa e `True` altrimenti.

`10 < 1 # 10 è minore di 1: falso -> False`

`a == 2 #a vale esattamente 2, se a è 2 è True`

- Le costanti `False` e `True` sono costanti di tipo `bool`

| Operatore          | Funzione        |
|--------------------|-----------------|
| <code>&lt;</code>  | Minore          |
| <code>&lt;=</code> | Minore o Uguale |
| <code>==</code>    | Uguale          |
| <code>&gt;=</code> | Maggiore Uguale |
| <code>&gt;</code>  | Maggiore        |
| <code>!=</code>    | Diverso         |

36



## Espressioni

- Operatori logici binari: **and or**
- Si usano per combinare valori di verità/falsità tipicamente ottenuti da espressioni relazionali
- Un'espressione logica ha valore `False` o `True`
- Esiste anche l'operatore unario di negazione: **not**

```
x = 2
x > 1 and x < 10 #x è compreso tra 1 e 10:
 vero->1
1 < x < 10 # sintassi legale

x == 0 or x > 20 #x è 0 oppure x maggiore di 20.
False entrambe-> False.

not(x < 0) # x<0: False, not False -> True
```

| OR    |       |        |
|-------|-------|--------|
| a     | b     | a or b |
| False | False | False  |
| False | True  | True   |
| True  | False | True   |
| True  | True  | True   |

| AND   |       |         |
|-------|-------|---------|
| a     | b     | a and b |
| False | False | False   |
| False | True  | False   |
| True  | False | False   |
| True  | True  | True    |

37

## Espressioni

- Sintassi delle espressioni si può definire con

```
<expr>:= <const>|<var>|(<expr>)|...
 := <expr><op2><expr>|<op1><expr>|...

<op2> := + | - | * | ** | / | // | % | ...
 := < | <= | == | != | >= | > | ...
 := and | or

<op1>:= not | - | +
```

39

## Sintassi

- Useremo spesso una notazione per descrivere la sintassi ammessa per un certo tipo di istruzione.
- Questo linguaggio si usa per definire la grammatica di altri linguaggi ed è detto EBNF: extended Backus-Naur form

`<non_terminale>:=<non_terminale> terminale;`

- Uso <> per indicare un simbolo che può espandersi

`<non_terminale>:= terminale1 | terminale2;`

- Uso | per indicare l'alternativa

`<non_terminale>:= [terminale opzionale] terminale | terminale;`

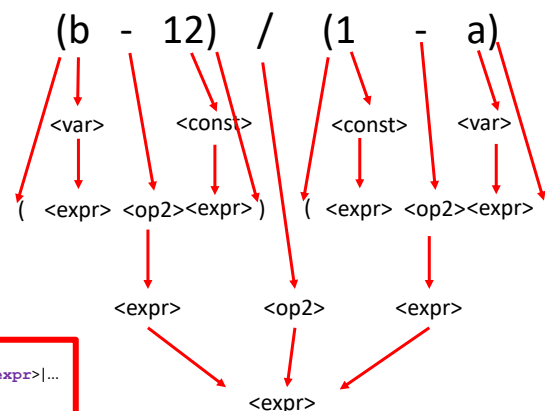
- Uso [] per indicare l'opzionalità di un simbolo (sia terminale che non terminale)

40

## Espressioni

- Interprete python quando si presenta un'espressione effettua il *parsing* sintattico

- Albero sintattico dell'espressione
- Ogni elemento terminale (es. <op>) viene *riconosciuto*
- La combinazione di più elementi terminali dà origine ad elementi non terminali (<expr>)
- Al termine se l'albero si conclude con <expr> l'espressione è sintatticamente corretta.



```
<expr>:= <const>|<var>|(<expr>)|...
 := <expr><op2><expr>|<op1><expr>|...

<op2>:= + | - | * | ** | / | // | % | ...
 := <| <= | == | != | >= | > | ...
 := and | or

<op1>::= not | - | +
```

41

# Espressioni

## Precedenze degli operatori

- Gli operatori si applicano in un ordine di precedenza prestabilito ad es. per gli aritmetici: \* / + -
- In generale è consigliabile **usare le parentesi** sempre per essere certi e massimamente espressivi sull'ordine in cui vogliamo che le espressioni siano valutate.

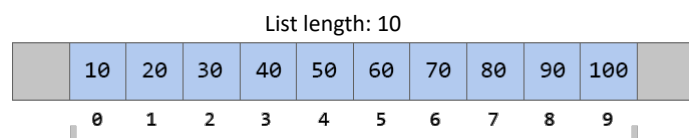
42

# Liste

- Una lista è una sequenza di locazioni di memoria contenenti variabili

```
vector = [1,3,3] #una lista di 3 elementi
counts = [] #una lista vuota
```

- Usiamo le liste per dare un nome ad un insieme di valori di cui manteniamo l'ordine



43

## Liste

- Dopo che ho definito una lista

```
vector = [1,2,3] # una lista di 3 elementi
```

- Posso accedere alle sue locazioni usando un indice.

```
vector[0] = 1 # definisco il vettore i per una base ortonormale
```

```
vector[1] = 0
```

```
vector[2] = 0
```

- Si conta da 0 e si arriva fino a DIMENSIONE-1.

45

## Liste

- Dopo che ho definito una lista

```
vector = [-30,-20,-10] # una lista di 3 elementi
```

- L'accesso ad una lista si chiama **indicizzazione**, in pratica a partire da un indice si ottiene una variabile

- L'indice è un valore **intero positivo**.

- In python posso usare un **intero negativo** per accedere agli elementi partendo dall'ultimo:

```
vector[-1]
```

```
-10
```

46

## Liste

- Posso sapere quanto è lunga una lista usando

```
vector = [1,2,3]
len(vector)
3
```

- Le liste sono strutture dinamiche a cui posso aggiungere elementi in coda usando

```
vector.append(4)
```

47

## Stringhe

- Una stringa è una sequenza di caratteri. In python `str` è il tipo che si usa per le stringhe. Le stringhe somigliano alle liste nel senso che si possono elaborare come sequenze.

```
nome = 'lorenzo'

nome[0]
'l'
nome[1]
'o'
```

- Posso misurare le stringhe sempre usando `len(nome)`
- Non posso modificare i caratteri di una stringa (vedremo perché a breve)

48

## Stringhe

- E' definito l'operatore somma (+) per le stringhe
- L'operatore somma restituisce una nuova stringa che rappresenta la concatenazione degli operandi

```
nome = 'lorenzo'

print ('Il mio nome è' + nome)

'Il mio nome è lorenzo'
```

49

## print

```
print(variabile1, variabile2, ...)
```

- La funzione print viene usata per stampare a schermo i valori delle variabili
- In python i tipi base sono convertiti automaticamente a stringhe se passati a print
- Non posso però concatenare stringhe e interi

```
eta = 40
print('Ho' + eta + ' anni')
```

```
TypeError: can only concatenate str (not "int") to str
```

50

## Formattazione

- Per poter combinare variabili di tipo diverso possiamo usare la formattazione
- Nelle ultime versioni di python questo è semplicissimo


```
print (f'Ho {eta} anni')
```

- La `f` che precede la stringa indica all'interprete che dovrà *formattare* la stringa
- Le variabili che voglio trasformate in stringhe saranno riportate tra `{ }`

51

## input

```
risposta = input('Domanda')
```

- `risposta` è una stringa che contiene l'input da tastiera terminato da Invio (Enter) 
- E' possibile inserire una stringa che verrà stampata a schermo per richiedere l'input all'utente

52

## Conversioni

- La conversione di una variabile da un tipo ad un altro può anche essere forzata in maniera esplicita.
- Non sono molti i casi in cui questo è necessario

```
voto = input('Inserire Voto') #la variabile voto è di tipo str
```

- Supponiamo che si voglia calcolare la media dei voti e di conseguenza occorra fare operazioni di somma e divisione
- voto va convertito ad esempio in float

```
voto = float(voto) # questo assegnerà alla variabile voto invece
che la stringa '27' il float 27.0
```

53

## Esercizi per casa

- Scrivere un programma che acquisisce 2 valori e ne calcola e stampa la media.
- Scrivere un programma che acquisisce un valore e ne stampa il quadrato
- Scrivere un programma che acquisisce le variabili nome, cognome, età e poi stampa una frase con le informazioni acquisite

Es.:

```
nome? Lorenzo
cognome? Seidenari
Eta? 100
Ciao mi chiamo Lorenzo Seidenari e ho 100 anni
```

54

54