- le **variabili** sono dei ***nomi*** che corrispondono a ***spazi/posizioni*** in memoria che contengono i dati
- le **istruzioni** indicano al PC quali operazione compiere
- le **funzioni/procedure** sono parti di codice riusabili
  - le **funzioni** restituiscono un risultato
  - le **procedure** NON restituiscono un risultato
    (ma fanno cose)

Tipi di dati: interi (di tipo `int`)

sono codificati in memoria come sequenza di cifre binarie (0/1)
MA a precisione "infinita"!

Python alloca (riserva) automaticamente tutto lo spazio di memoria necessario a contenere la codifica binaria del numero

Il linguaggio **C/C++** invece usa una word (64 bit su un PC a 64 bit) e può rappresentare al massimo il valore $2^{63} - 1$ = 9_223_372_036_854_775_807 (circa 9 miliardi di miliardi)

Com'è codificato un numero naturale in binario

- in base 2, ovvero usando solo le cifre 0 e 1
- come per i numeri in base 10, ciascuna cifra in posizione $P = 0, 1, 2, 3$ ... (contando da destra) 'pesa' $base^P$

Cifre	**0**	**1**	**1**	**0**
Posizione	3	2	1	0
Peso...	2^3	2^2	2^1	2^0
...ovvero	8	4	2	1
Contributo	$8 * 0$	$4 * 1$	$2 * 1$	$1 * 0$

- Risultato: $0 + 4 + 2 + 0 = 6$

- il più grande numero con 4 cifre è 1111 ovvero $8 + 4 + 2 + 1 = 15 = 2^4 - 1$

Com'è codificato un numero relativo (col segno) in binario

La codifica più comune viene chiamata "in complemento a 2"

I pesi sono gli stessi, ma il peso del bit più a sinistra (il più significativo) ha valore negativo

Cifre	**1**	**1**	**1**	**0**
Posizione	3	2	1	0
Peso	$-(2^3)$	2^2	2^1	2^0
ovvero	-8	4	2	1
Contributo	$-8 * 1$	$4 * 1$	$2 * 1$	$1 * 0$

- Risultato: $-8 + 4 + 2 + 0 = -2$
- Il più piccolo numero di 4 bit sarà $1000 = -8 + 0 + 0 + 0 = -8 = -(2^3)$
- Il più grande sarà $0111 = 0 + 4 + 2 + 1 = 7 = 2^3 - 1$
- **Intervallo di rappresentazione per n cifre:**

$$[-2^{n-1}, 2^{n-1} - 1]$$

In [1…
```
# in Python posso tranquillamente scrivere un intero con 48 zeri ... ovvero
# mille miliardi di miliardi di miliardi di miliardi di miliardi
1_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000
```

Out[1…
```
1000000000000000000000000000000000000000000000000
```

In [5…
```
# NOTA: per sapere un dato di che tipo è si usa la funzione 'type'
type(1_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000)
```

Out[5…
```
int
```

Operazioni matematiche

Sono presenti le solite 4 operazioni `+`, `-`, `*`, `/`, e inoltre

- `//` : la divisione intera
  (che produce **int** solo se entrambi gli operandi sono **int**)
  NB: la divisione normale `/` produce sempre **float**
- `%` : il resto della divisione intera (detto anche modulo)
- `**` : la potenza

Altre funzioni matematiche sono disponibili nella libreria `math`

In [3…

```
10**1000 # '**' vuol dire 'elevato a potenza' -> (1 seguito da 1000 zeri)
```

Out[3…

```
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

In [2…

```
# divisione intera con //
print('la divisione intera 81 // 7 fa', 81 // 7)
```

```
la divisione intera 81 // 7 fa 11
```

In [3…

```
# resto della divisione con %
print('il resto di         81 // 7  è', 81 % 7)
```

```
il resto di         81 // 7  è 4
```

In [4…

```
# verifichiamo
print('infatti             4+7*11  fa', 4+7*11)
```

```
infatti              4+7*11  fa 81
```

Numeri con la virgola (di tipo `float`)

Sono codificati con un numero di bit fisso (1 o 2 word)
(Vedi standard IEEE 754)

Esempio: word da 64 bit (per CPU a 64 bit)

segno (1 bit)	esponente (11 bit)	mantissa (52 bit)
0=+ 1=-	esponente	cifre significative

Hanno un numero fisso di bit nella mantissa, quindi la precisione
(cioè il numero di cifre corrette) è limitata

- una word -> 64 bit -> 52 bit di mantissa -> circa 16 cifre decimali
- due word -> 128 bit -> 112 bit di mantissa -> circa 33 cifre decimali

Esempio (con 1 bit di segno, 3 bit di esponente e 4 bit di mantissa)

S	E	E	E	M	M	M	M
1	0	1	0	1	0	1	1

- Il numero si trova calcolando

$$(-1)^{segno} * 2^{esponente+offset} * 1.mantissa$$

- segno = 1 -> numero negativo
- esponente = 010 -> in complemento a 2 vale 2
- per comodità supponiamo che l'offset sia 0 (dipende in realtà dallo standard)
- mantissa = 1011 -> viene aggiunta un 1 (e una virgola) a sinistra
  - 1.1011 viene convertito in decimale sapendo che i pesi, andando verso destra si dividono per 2 -> (0.1 = 1/2, 0.01 = 1/4, 1/8,

1/16, 1/32)
    - quindi il valore della parte mantissa 1.1011 è $1 + 1/2 + 0 + 1/8 + 1/16 = 27/16 = 1.6875$
- il numero quindi è: $-1 * 2^{2+0} * 1.6875 = -6.75$

In [1…
```
# Vediamo quante cifre decimali possono essere rappresentate da una mantissa di 52 bit
# 2 alla 52 è maggiore di 10 alla 15 e minore di 10 alla 16?
print(' 52 bit sono circa 16 cifre decimali?', 10**15 < 2**52 < 10**16)
print(2**52, 10**15, 10**16)
```

```
 52 bit sono circa 16 cifre decimali? True
4503599627370496 1000000000000000 10000000000000000
```

In [7…
```
# 2 alla 112 è maggiore di 10 alla 13 e minore di 10 alla 34?
print('112 bit sono circa 33 cifre decimali?', 10**33 < 2**112 < 10**34)
```

```
112 bit sono circa 33 cifre decimali? True
```

In [1…
```
# vediamo che succede se stampo 10 alla 15 oppure 10 alla 16
print(10.0**15, 10.0**16) # del primo stampa TUTTE LE CIFRE! il secondo è invece "approssimato"

type(17.5)   # ecco il tipo di un numero con la virgola
```

```
1000000000000000.0 1e+16
```

Out[1…
```
 float
```

In [1…
```
# ESPRESSIONI MATEMATICHE
# posso scrivere espressioni matematiche e calcolarle
# (valgono le precedenze: PRIMA **, POI * e / E POI + e -)
print('3 + 14 * 5 / 4 = ', 3 + 14 * 5 / 4 )
# prima calcola 14*5=70 poi 70/4=17.5 poi 17.5+3=20.5
```

```
3 + 14 * 5 / 4 =  20.5
```

In [1…
```
print('3 + 5 * 10**2 / 4 = ', 3 + 5 * 10**2 / 4)
# prima calcola 10**2=100 poi 5*100=500 poi 500/4=125.0 poi 125.0+3=128.0
# NOTA la divisione produce un float e da quel momento in poi i risultati sono float
```

```
3 + 5 * 10**2 / 4 =  128.0
```

In [1…
```
# ma se voglio che vengano calcolate in ordine diverso
```

```
# basta usare le parentesi tonde
# (SOLO TONDE! le quadre e graffe hanno significato diverso)
print('14 * 5 / (4 + 3) = ', 14 * 5 / (4 + 3))
# prima calcola 14*5=70 poi 4+3=7 poi 70/7=10
# dato che uso la divisione normale ottengo sempre un float
```

```
14 * 5 / (4 + 3) =  10.0
```