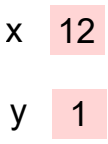
# O que variáveis armazenam?

Variáveis armazenam valores de tipos básicos

In [ ]:

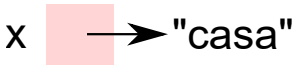
```
x, y = 12, 1
```




Variáveis armazenam **referências** a valores de tipos compostos

In [ ]:

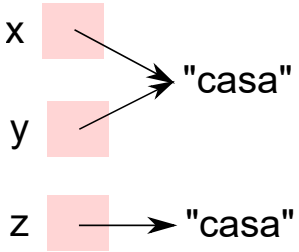
```
x = "casa"
```




Strings são **imutáveis**, não há perigo se variáveis *compartilham* o mesmo objeto

In [1]:

```
x = "casa"
y = "casa"
z = x[:2] + "sa"
```




Dizemos que há *aliasing* entre `x` e `y`

Enquanto `==` verifica igualdade dos conteúdos referenciados

In [2]:

```
x==y
```

Out[2]:

```
True
```

In [3]:

```
x == z
```

Out[3]:

```
True
```

# O operador `is`

o operador `is` verifica igualdade das referências

In [4]:

```
x is y
```

Out[4]:

```
True
```

In [5]:

```
x is z
```

Out[5]:

```
False
```

Na prática, da mesma forma que com números, podemos pensar que as variáveis contém os strings por inteiro

No caso de listas a situação é diferente, pois listas são **mutáveis** (podem ser modificados)

In [6]:

```
a = [1,2,3]
b = a

a[1] = 200

a
```

Out[6]:

```
[1, 200, 3]
```

In [7]:

```
b
```

Out[7]:

```
[1, 200, 3]
```

In [8]:

```
a == b
```

Out[8]:

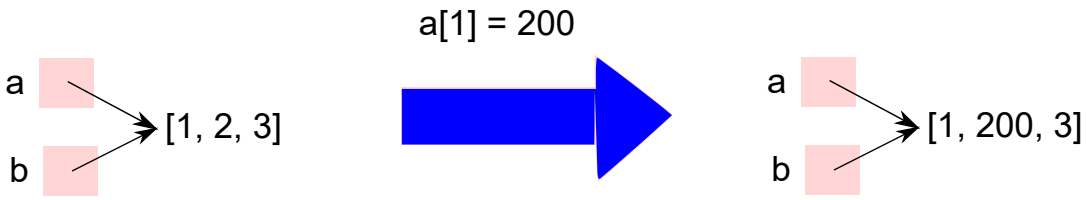
```
True
```

In [9]:

```
a is b
```

Out[9]:

```
True
```




`b` é alterado como um **efeito colateral** da atribuição `a[1] = 200`

# Criando clones

Quando queremos evitar efeitos colaterais por *aliasings*, criamos cópias/**clones** de arrays

In [11]:

```
a = [1,2,3]
b = a[:]     # a[:] é um novo array, cópia/clone de a

a[1] = 200

a
```

Out[11]:

```
[1, 200, 3]
```

In [12]:

```
b
```

Out[12]:

```
[1, 2, 3]
```

In [13]:

```
a == b
```

Out[13]:

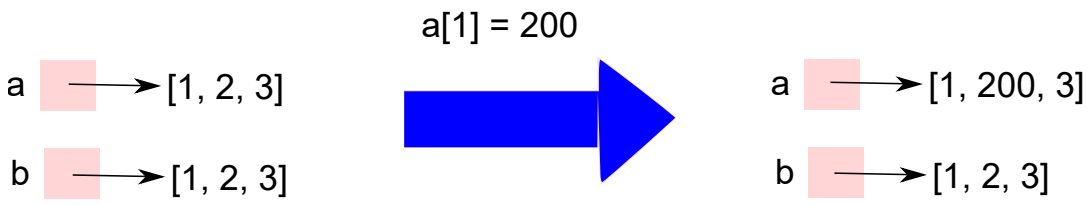
```
False
```

In [14]:

```
a is b
```

Out[14]:

```
False
```




# Passagem de argumentos por valor (cópia)

Numa chamada a função, os argumentos (ou parâmetros reais) são avaliados e copiados em variáveis locais (parâmetros formais)

In [15]:

```
def maior(a, b):
    if a > b:
        return a
    else:
        return b

maior(3+4, 2*5)
```

Out[15]:

```
10
```

1. os argumentos são avaliados, dando `7` e `10`, respectivamente
2. variáveis locais `a` e `b` são criadas
3. `7` é copiado em `a` e `10` é copiado em `b`
4. o corpo de `maior` é executado
    - como `a < b`, `return b` é executado, no caso `return 10`
    - as variáveis locais `a` e `b` são destruídas (liberada a memória)
5. a execução de maior termina

Observe o seguinte exemplo:

In [16]:

```
def inc(w):
    w = w + 1

x = 10
inc(x)

x
```

Out[16]:

```
10
```

A função `inc` não tem utilidade nenhuma

No entanto, quando o argumento é um valor mutável, por exemplo um lista, uma função pode alterar o argumento.

In [17]:

```
def incrementaTodos(a):
    for i in range(len(a)):
        a[i] += 1

lst = [1,2,3]
incrementaTodos(lst)

lst
```

Out[17]:

```
[2, 3, 4]
```

A alteração em `lst` é feita efetivamente por `incrementaTodos` , pois `lst` contém a referência para a lista, e essa referência é passada como argumento