

Números no Python!

Just Python

Danilo J. S. Bellini
@danilobellini

2018-07-14

Mostrar os recursos do Python a respeito de números e matemática:

- 5 tipos de números, sendo 3 *built-ins*:
 - `int`
 - `float`
 - `complex`
 - `decimal.Decimal`
 - `fractions.Fraction`
- Conversão de números de/para strings
- Conversão de objetos em números
- Módulos da biblioteca padrão:
 - `numbers`
 - `math`
 - `cmath`
 - `decimal`
 - `fractions`
 - `random`
 - `statistics`
 - `array`
 - `struct`

Literais

Um “literal” é um objeto que a linguagem permite escrever diretamente em um dado tipo (o próprio *token* denota o tipo). Não é um literal algo que exige conversão/processamento ou armazenamento associado a algum identificador. Dos números em Python, apenas os 3 tipos built-in possuem literais.

```
# int
2
128_237 # 0 _ é apenas visual
12_82_37 # Este é o mesmo número acima

# float (presença do ".")
2.3
234_122.999_112
float("inf") # Isto não é um literal de float,
              # mas o "inf" é um literal do tipo string

# complex (sufixo "j" ou "J")
2.23j
2 + 3J
complex(2, 3) # 0 número acima, sem ser como literal
```

Muito mais que 32, 64 ou 128 bits!

```
>>> def fatorial(n):  
...     return n * fatorial(n - 1) if n > 1 else 1  
...  
>>> fatorial(12)  
479001600  
>>> fatorial(32)  
263130836933693530167218012160000000  
>>> fatorial(132)  
11182486511960043074499630760761690299756247557184263383841 ] ...  
↪ 21675683611696728201184540457302606885100879909271961049626 ] ...  
↪ 85462595837360336094267205134948250389032461924909766607715 ] ...  
↪ 92408648929771520000000000000000000000000000000000000000000000
```

Representações do int em outras bases

A base padrão para os inteiros literais é 10. Há 3 *built-ins* específicos para representação de inteiros em strings em potências de 2 como base: `bin`, `oct` e `hex`. A string fornecida é da mesma forma que o Python aceita um literal em tais bases.

```
>>> # Binário
>>> 0b10110
22
>>> bin(23)
'0b10111'
>>> # Hexadecimal
>>> 0x16
22
>>> hex(23)
'0x17'
>>> # Octal
>>> 0o26
22
>>> oct(23)
'0o27'
```

```
>>> int("2112", base=3) # Base arbitrária
68
>>> 2*27 + 1*9 + 1*3 + 2*1 # Check!
68
>>> 00
0
>>> 01 # Prefixo 0: octal no Python 2
Traceback (most recent call last):
  File "<stdin>", line 1
    01
    ^
SyntaxError: invalid token
```

Operadores matemáticos

Os operadores, denotados por símbolos como $+$ e $-$, são *unários* quando possuem 1 argumento/número, e *binários* quando possuem 2. Cada operador executa um *dunder* (método com *double underscore*) do tipo do número, cujo nome está no comentário:

```
>>> +2 # __pos__ # 0 "@", __matmul__,  
2  
>>> -2 # __neg__ # não é usado pelas  
-2  
>>> 2 + 2 # __add__ (soma) # classes do Python  
4  
>>> 2 * 4 # __mul__ (multiplicação), precede soma  
8  
>>> 2 ** 4 # __pow__ (potenciação), precede multiplicação  
16  
>>> 17 % 3 # __mod__ (resto da divisão)  
2  
>>> 17 // 3 # __floordiv__  
5  
>>> 17 / 3 # __truediv__  
5.666666666666667
```

O Python 2 possuía um `__div__`, o qual era associado ao operador `"/` na ausência do `__future__`. Em todo legado que ainda utilizar o Python 2, é recomendado utilizar esse `__future__`, a qual deve ser a primeira linha de todo módulo.

```
>>> 3 / 2 # __div__ no Python 2
1
>>> from __future__ import division
>>> 3 / 2 # __truediv__
1.5
>>> divmod(7, 5) # 7 // 5, 7 % 5
(1, 2)
```

O `divmod` é um *built-in* existente em ambas as versões do Python.

Built-in range

Esse é um exemplo famoso de uso de inteiros, o `range` devolve um objeto que gera tardiamente valores inteiros dentro de uma faixa, e comporta-se como uma sequência. No Python 2, esse *built-in* devolvia uma lista, e o `xrange` é o que possuía um comportamento mais próximo do `range` do Python 3.

```
>>> range(5) # 1 parâmetro: stop (nunca incluído)
range(0, 5)
>>> range(3, 7) # 2 parâmetros: start, stop
range(3, 7)
>>> list(range(3, 7))
[3, 4, 5, 6]
>>> range(3, 7)[2:] # Pode aplicar slices
range(5, 7)
>>> list(range(3, 7, 2)) # Terceiro parâmetro: step
[3, 5]
>>> list(range(-5, -12, -2)) # Negativos!
[-5, -7, -9, -11]
```

Dijkstra prefere esse modelo pois `len(...) == stop - start`.

Os operadores são funções no módulo `operator`, as quais podem ser utilizadas ao invés dos *tokens* (símbolos) dos operadores. Isto pode ser útil caso o operador seja um parâmetro.

```
>>> import operator
>>> operator.neg(25) # Operador unário
-25
>>> operator.add(3, 5) # Operador binário
8
>>> from functools import reduce
>>> reduce(operator.mul, [5, 3, 2]) # Produtório
30
```

Os operadores relativos aos números existem com o mesmo nome do respectivo *dunder*.

A igualdade de números independe do tipo!

```
>>> 7 == 2 # __eq__
False
>>> 18 != 18. # __ne__ de int e float
False
>>> 7 > 3 # __gt__
True
>>> 7 >= 7. # __ge__ de int e float
True
>>> 8 < 7 # __lt__
False
>>> 5 <= 18 # __le__
True
>>> 1 + 0j
(1+0j)
>>> 1 + 0j == 1
True
```

Operadores lógicos ou *bitwise*

Esses operadores são específicos para inteiros, aplicados nos bits pensando no número em binário.

```
>>> ~38 # __invert__, ~valor == -(valor + 1)
-39
>>> 13 & 7 # __and__, 0b1101 & 0b0111 == 0b0101
5
>>> 13 | 7 # __or__, 0b1101 | 0b0111 == 0b1111
15
>>> 13 ^ 7 # __xor__, 0b1101 ^ 0b0111 == 0b1010
10
>>> 7 << 2 # __lshift__, move 2 bits p/ a esquerda
28
>>> 7 >> 1 # __rshift__, move 2 bits p/ a direita
3
```

float: Ponto flutuante radix 2 IEEE 754 de precisão dupla

São números escritos da forma $1.xxxx * 2 ** y$, além do sinal. A representação em base decimal costuma ser aproximada, e não existe literal de float em outra base.

```
>>> 0. # Basta 1 dígito decimal e o ponto
0.0
>>> -.0 # Há um zero positivo e um negativo!
-0.0
>>> 0. == -0. # Só curiosidade, nunca use == com float!
True
>>> 1e-2 # Notação científica
0.01
>>> 1E+5 # O "+" é opcional, "e"/"E" são iguais
100000.0
>>> float.hex(3.) # Método hex, 3 == 0b1.1 * 2
'0x1.8000000000000p+1'
```

Cuidados com o float

O número de bits da mantissa (quantidade de x) é fixo, mas números como $1/3$ e $1/5$ são dízimas periódicas em binário.

```
>>> (1/3).hex()
'0x1.555555555555p-2'
>>> (1/5).hex()
'0x1.999999999999ap-3'
>>> 1/3 - 1/2 + 1/6
-2.7755575615628914e-17
>>> 1/3 - 1/2 == 1/6
False
```

math.isclose

Use `math.isclose` ao invés de igualdade para comparar números quando estiverem em ponto flutuante. A comparação é:

$$|a - b| \leq \max(\text{tol}_{rel} \cdot \max(|a|, |b|), \text{tol}_{abs})$$

Em Python, c/ os nomes dos argumentos nominados de tolerância:

```
abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)
```

```
>>> 1/2 - 1/3 == 1/6
False
>>> import math
>>> math.isclose(1/2 - 1/3, 1/6)
True
>>> math.isclose(27, 26.1)
False
>>> math.isclose(27, 26.1, abs_tol=1)
True
>>> math.isclose(2.7e30, 2.6999e30, abs_tol=1)
False
>>> math.isclose(2.7e30, 2.6999e30, rel_tol=.0001)
True
```

Informações no sys.float_info

```
>>> import sys
>>> sys.float_info.mant_dig # Dígitos da mantissa
53
>>> sys.float_info.max_exp # Maior expoente do 2
1024
>>> sys.float_info.min_exp # Menor expoente do 2
-1021
>>> sys.float_info.radix
2
>>> sys.float_info.max # Maior float representável
1.7976931348623157e+308
>>> sys.float_info.min # Menor float representável
2.2250738585072014e-308
>>> sys.float_info.max + 1 # 0 1 é descartável
1.7976931348623157e+308
>>> sys.float_info.max + 1e292 # Passou do máximo!
inf
```

Uma forma de verificar se o ponto flutuante representa um número inteiro.

```
>>> (.5 + .5).is_integer()
True
>>> (1/3) * 5 * (3/5) # Deveria ser 1
0.9999999999999999
>>> ((1/3) * 5 * (3/5)).is_integer() # Gotcha!
False
```

O ideal é comparar com `math.isclose`.

Operadores com float

Os mesmos operadores matemáticos e booleanos podem ser usados com ponto flutuante.

```
>>> 2.27 > 2.25
True
>>> -2.27 > -2.25
False
>>> 2 ** -4 # __pow__ de inteiros pode devolver float!
0.0625
>>> 2 ** .25 # Raiz quarta usando __pow__
1.189207115002721
>>> import math
>>> math.sqrt(2) # Raiz quadrada
1.4142135623730951
>>> 2 ** .5 # Alternativa (prefira o math.sqrt)
1.4142135623730951
```

Divisão com float e round

```
>>> 2.7 / 1.2 # Divisão
2.2500000000000004
>>> 2.7 // 1.2
2.0
>>> round(2.7 / 1.2) # Versões antigas eram como o //
2
>>> round(2.7 / 1.2, 0) # Chama float.__round__(x, ndigits)
2.0
>>> round(2.7 / 1.2, 1)
2.3
>>> 2.7 % 1.2 # Resto da divisão
0.30000000000000027
>>> 2.7 % 1 # Parte fracionária
0.7000000000000002
```

nan, inf e verificação no math

Ponto flutuante possui 3 números especiais: nan, inf e -inf.

```
>>> float("nan") == float("nan") # not a number
False
>>> float("inf") == float("inf") + 1
True
>>> import math
>>> math.isnan(float("nan"))
True
>>> math.isnan(float("inf"))
False
>>> math.isinf(float("nan"))
False
>>> math.isinf(float("inf"))
True
>>> math.isfinite(float("inf"))
False
>>> math.isfinite(float("nan"))
False
>>> math.isfinite(1e300)
True
```

complex: Complexos formados por float

O complexo é um número na forma $a + b * 1j$, em que o sufixo “j” do literal denota a unidade complexa, e suas partes a e b são float.

```
>>> complex(2, 3) # Parecem int, mas ...
(2+3j)
>>> complex(2, 3).real
2.0
>>> complex(2, 3).imag
3.0
>>> (2 + 3j) + (1 - 3j) # Tem os operadores esperados
(3+0j)
>>> (2 + 3j) * (1 - 3j) # Incluindo multiplicação!
(11-3j)
>>> import cmath
>>> cmath.e ** (1j * cmath.pi) # exp(pi * 1j) == -1
(-1+1.2246467991473532e-16j)
>>> 1j > 2j # Não possuem relação de ordem
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'complex' ...
↪ and 'complex'
```

decimal.Decimal: Ponto flutuante radix 10

SIM! É ponto flutuante! E a precisão não é ilimitada! Mas a base é 10, os valores são exatos nessa base. Útil para trabalhar com valores monetários.

```
>>> import decimal
>>> decimal.getcontext().prec # Precisão (dígitos)
28
>>> context = decimal.getcontext()
>>> context.prec = 5
>>> decimal.setcontext(context) # Define precisão
>>> d = decimal.Decimal("18.7654")
>>> d # Mantém a atribuição
Decimal('18.7654')
>>> d * 2 # Utiliza o contexto antes de fazer contas
Decimal('37.531')
>>> round(d * 2, 2) # Chama Decimal.__round__
Decimal('37.53')
```

fractions.Fraction: Números racionais arbitrários

Frações de números inteiros, já simplificada.

```
>>> from fractions import Fraction
>>> Fraction(5040, 120)
Fraction(42, 1)
>>> Fraction(1, 2) - Fraction(1, 3)
Fraction(1, 6)
>>> round(Fraction(1, 6), 3) # Força representação decimal
Fraction(167, 1000)
>>> 1 / 6 # 0.167, se representado com 3 dígitos após "."
0.16666666666666666
>>> Fraction(7, 2).numerator
7
>>> Fraction(7, 2).denominator
2
```

Remove o sinal do número, devolve a magnitude no caso de um complexo. Chama o método `__abs__` do tipo.

```
>>> abs(-17)
17
>>> abs(-2.7)
2.7
>>> abs(4 + 3j)
5.0
>>> import decimal, fractions
>>> abs(decimal.Decimal("-8.231"))
Decimal('8.231')
>>> abs(fractions.Fraction(8, -3))
Fraction(8, 3)
```

Peraê, booleanos não são números! Mas para todos os tipos de números, a conversão para booleano é:

- False quando zero
- True caso contrário

```
>>> bool(-.0) # float
False
>>> bool(.0000001)
True
>>> bool(float("nan"))
True
>>> bool(float("inf"))
True
>>> bool(0j) # complex
False
>>> bool(0j + .0002)
True
>>> bool(.002j)
True
```

```
>>> bool(0) # int
False
>>> bool(42)
True
>>> from decimal import Decimal
>>> bool(Decimal(0))
False
>>> bool(Decimal("0.0002"))
True
>>> from fractions import Fraction
>>> bool(Fraction(0, 5))
False
>>> bool(Fraction(1, 2**90))
True
```


Todo número é imutável e *hashable*. Por conta da igualdade com o `__eq__`, o hash de um número não depende do tipo.

```
>>> hash(0) # Para inteiros, é o próprio número
0
>>> hash(123)
123
>>> hash(55.)
55
>>> hash(float("inf")) # Valor de sys.hash_info.inf
314159
>>> hash(float("nan")) # Valor de sys.hash_info.nan
0
>>> hash(.55)
1268213655067531776
>>> float("inf") == 314159
False
>>> {float("inf"): 1, 314159: 2, # Just for fun
... float("nan"): 3, 0: 4, 0.: 5, 0j: 6}
{inf: 1, 314159: 2, nan: 3, 0: 6}
```

Esses módulos possuem funções trigonométricas, logaritmos e outras funções matemáticas. O `cmath` trabalha com complexos, enquanto o `math` lida principalmente com `float`. Há no `math` alguns recursos para inteiros (atualmente se discute migrá-los para um novo módulo `imath`).

```
>>> import math, cmath
>>> cmath.sin(math.pi / 6) # sin(30 degrees)
(0.4999999999999999+0j)
>>> math.sin(math.pi / 6)
0.4999999999999999
>>> cmath.exp(1j * cmath.pi) # exp(pi * 1j)
(-1+1.2246467991473532e-16j)
>>> math.exp(1j * math.pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
```

No próximo slide constam todas as funções do módulo `cmath`.

Funções matemáticas dos módulos math e cmath

- `acos`: Arco-cosseno
- `acosh`: Arco-cosseno hiperbólico
- `asin`: Arco-seno
- `asinh`: Arco-seno hiperbólico
- `atan`: Arco-tangente
- `atanh`: Arco-tangente hiperbólica
- `cos`: Cosseno
- `cosh`: Cosseno hiperbólico
- `sin`: Seno
- `sinh`: Seno hiperbólico
- `tan`: Tangente
- `tanh`: Tangente hiperbólica

- `exp`: Exponencial
- `log`: Logaritmo natural
- `log10`: Logaritmo base 10
- `sqrt`: Raiz quadrada

Específico do `cmath`:

- `phase`: Fase/argumento do complexo
- `polar`: Raio e fase do complexo
- `rect`: Complexo a partir de raio e fase

Específico do `math`:

- `degrees`: Radianos p/ graus
- `radians`: Graus p/ radianos

Exemplo de execução de funções do cmath

```
>>> from math import sqrt, pi
>>> from cmath import phase, polar, rect
>>> abs(1 + 1j) # Módulo
1.4142135623730951
>>> phase(1 + 1j) # Fase
0.7853981633974483
>>> pi / 4 # A fase é realmente pi / 4
0.7853981633974483
>>> polar(1 + 1j) # Cartesiano -> Polar
(1.4142135623730951, 0.7853981633974483)
>>> rect(sqrt(2), pi / 4) # Polar -> Cartesiano
(1.0000000000000002+1j)
```

Funções do math p/ float

Arredondamento:

- trunc: Trunca p/ mais próximo de 0
- floor: Trunca p/ menor valor
- ceil: Arredonda p/ maior valor

Operações nativas:

- fabs: Valor absoluto (float)
- fmod: Resto da divisão da máquina
- fsum: Somatório acurado p/ float

Decomposição:

- copysign: Mistura magnitude/sinal
- frexp: Mantissa/expoente base 2
- modf: Mantissa e parte inteira

Estatística:

- gamma: Gamma de Euler
- erf: Função erro (integral da normal)

Outros:

- erfc: $1 - \text{erf}(n)$
- expm1: $\exp(x) - 1$
- log1p: $\log(1 + x)$
- log2: Logaritmo base 2
- ldexp: $a * 2^{**} b$
- hypot: $\sqrt{a*a + b*b}$
- pow: $a^{**} b$

Exemplo de execução de funções do math

```
>>> import math
>>> math.gamma(6) # Para inteiros, é o fatorial(n - 1)
120.0
>>> math.fmod(-5, 3) # Nem sempre é igual ao %
-2.0
>>> (-5) % 3
1
>>> math.modf(27.38) # Separa parte fracionária/inteira
(0.3799999999999999, 27.0)
>>> math.copysign(-27.38, 12) # 1º com sinal do 2º
27.38
>>> math.frexp(22) # Decompõe mantissa/expoente base 2
(0.6875, 5)
>>> 2 ** 5
32
>>> .6875 * 32
22.0
```

Há duas: gcd e factorial.

```
>>> from math import gcd, factorial
>>> factorial(7) # 7*6*5*4*3*2*1
5040
>>> gcd(18, 12) # Maior divisor comum
6
```

Módulo random para embaralhamento

Esse módulo possui rotinas para geração de números pseudo-aleatórios fundamentado no Mersenne Twister. Segue um exemplo de rotinas para embaralhamento:

```
>>> import random
>>> random.seed(42) # Congela valores p/ re-execuções
>>> # Seleciona algum dos possíveis valores
>>> [random.choice("ABCD") for unused in range(10)]
['A', 'A', 'C', 'B', 'B', 'B', 'A', 'A', 'D', 'A']
>>> data = list(range(10))
>>> random.shuffle(data) # Embaralha in place
>>> data
[6, 7, 2, 9, 5, 4, 8, 3, 1, 0]
>>> random.sample(data, 4) # "Amostra" de data sem repetir
[1, 8, 9, 3]
```


Módulo random para geração de números

```
>>> random.seed(11)
>>> # Distribuição uniforme:
>>> random.random() # Gera float 0 <= x < 1
0.4523795535098186
>>> a, b = 3, 7
>>> random.randint(a, b) # Gera int a <= x <= b
7
>>> random.randrange(a, b) # Gera int a <= x < b
6
>>> random.uniform(a, b) # Gera float entre a e b
4.807329282398261
>>> # Há outras distribuições! lognormal, gamma, normal, ...
>>> random.triangular()
0.7307841348511
```

Há ainda uma classe `Random` para permitir guardar múltiplos estados ao invés de um único estado global.

- mean: Média
- harmonic_mean: Média harmônica
- median: Mediana (média¹)
- median_low: Mediana (menor)
- median_high: Mediana (maior)
- median_grouped: Mediana do agrupamento
- mode: Moda

Segue um exemplo ...

¹Média é dos 2 valores centrais, caso haja um número par de elementos.

Medidas de tendência central com o módulo statistics

```
>>> import statistics
>>> statistics.mean([3, 5, 10]) # int
6
>>> statistics.mean([3, 5, 11]) # float!
6.333333333333333
>>> statistics.median([3, 5, 6, 11])
5.5
>>> statistics.median_low([3, 5, 6, 11])
5
>>> statistics.median_high([3, 5, 6, 11])
6
>>> statistics.median_grouped([3, 5, 5, 6, 11])
5.25
>>> statistics.mean([3, 5, 5, 6, 11])
6
>>> statistics.mode([3, 5, 5, 6, 11])
5
```

Com `variance` e `stdev`, esse módulo calcula a variância e o desvio padrão amostrais. Com o prefixo “p”, os valores são da população.

```
>>> import statistics
>>> statistics.variance([3, 5, 5, 6, 11])
9
>>> statistics.stdev([3, 5, 5, 6, 11])
3.0
>>> statistics.pvariance([3, 5, 5, 6, 11])
7.2
>>> statistics.pstdev([3, 5, 5, 6, 11])
2.6832815729997477
```

Representação de float em strings

Já vimos que podemos converter de strings para números usando os construtores das próprias classes. Mas e o caminho contrário, dos números chegar nas strings? E se quisermos limitar a precisão nos números em ponto flutuante? Há 3 formas:

```
>>> "%f" % 15.7
'15.700000'
>>> "%g" % 15.7
'15.7'
>>> "%.32f" % 15.7 # str.__mod__
'15.69999999999999928945726423989981'
>>> "{:.32f}".format(15.7) # str.format
'15.69999999999999928945726423989981'
>>> f"{15.7:.32f}" # f-string
'15.69999999999999928945726423989981'
>>> number, digits = 15.7, 32
>>> f"{number:.{digits}f}"
'15.69999999999999928945726423989981'
```

Podemos usar as mesmas formas adotadas para converter `float`, mas há um método `normalize` que permite "cortar zeros à direita".

```
>>> from decimal import Decimal
>>> "%f" % Decimal("15.70")
'15.700000'
>>> "%g" % Decimal("15.70")
'15.7'
>>> Decimal("15.70")
Decimal('15.70')
>>> Decimal("15.70").normalize()
Decimal('15.7')
>>> number, digits = Decimal("15.7"), 32
>>> f"{number:.{digits}f}"
'15.700000000000000000000000000000000000'
```

Representação de int em strings

```
>>> "%d" % -232
'-232'
>>> "%6d" % -232
'  -232'
>>> "%06d" % -232
'-00232'
>>> "{:07d}".format(-232)
'-000232'
>>> value = -232
>>> f"{value + 2:08d}"
'-0000230'
```

Container de números de um único tipo. Os métodos lembram os de uma lista.

```
>>> from array import array
>>> ar = array("f", [.1, .2, .3, .4])
>>> ar.typecode # Ponto flutuante de precisão simples
'f'
>>> ar.extend([.5]) # Métodos como do tipo list
>>> ar.append(1e100) # Maior que o máximo representável
>>> ar # Exibe o ruído de conversão p/ precisão dupla
array('f', [0.10000000149011612, 0.20000000298023224,
↪ 0.300000001192092896, 0.4000000059604645, 0.5, inf])
>>> ar.itemsize # Bytes por elemento
4
>>> len(ar) # Número de elementos
6
```

...

Tipos que podem ser usados com o array

- "b": inteiro de 1 bytes com sinal
- "B": inteiro de 1 bytes sem sinal
- "h": inteiro de 2 bytes com sinal
- "H": inteiro de 2 bytes sem sinal
- "i": inteiro de 2 bytes com sinal
- "I": inteiro de 2 bytes sem sinal
- "l": inteiro de 4 bytes com sinal
- "L": inteiro de 4 bytes sem sinal
- "q": inteiro de 8 bytes com sinal
- "Q": inteiro de 8 bytes sem sinal
- "f": ponto flutuante IEEE754 radix 2 de precisão simples (4 bytes)
- "d": ponto flutuante IEEE754 radix 2 de precisão dupla (8 bytes)
- "u": unicode de 2 ou 4 bytes

Módulo struct

Esse módulo é usado para comunicação binária, convertendo dados de diversos tipos (inclusive números) de/para bytes.

```
>>> import struct
>>> struct.unpack("bb", b"\x05\x06") # Inteiros de 1 byte
(5, 6)
>>> struct.unpack("Bb", b"\xF5\xF6") # Sem/com sinal
(245, -10)
>>> struct.pack("hh", 0xC7, 0xF3) # Inteiros de 2 bytes
b'\xc7\x00\xf3\x00'
>>> struct.pack(">hh", 0xC7, 0xF3) # Big endian
b'\x00\xc7\x00\xf3'
>>> struct.pack("<hh", 0xC7, 0xF3) # Little endian
b'\xc7\x00\xf3\x00'
>>> pair_struct = struct.Struct(">hh") # Lembra re.compile
>>> pair_struct.pack(0xD8, 0xE2)
b'\x00\xd8\x00\xe2'
>>> pair_struct.unpack(b"\x07\x00\x00\xff")
(1792, 255)
```

O ">hh" nativo pode ser tanto little como big endian.

Módulo numbers: ABCs de números

Há 5 ABCs de números nesse módulo: Integral, Rational, Real, Complex e Number, cada um herdando do anterior nessa ordem.

```
>>> import numbers
>>> numbers.Complex.mro()
[<class 'numbers.Complex'>, <class 'numbers.Number'>, <class ...
↩ 'object'>]
>>> isinstance(2, numbers.Complex)
True
```

Criando sua ABC c/ `__instancecheck__`

```
>>> class OddMeta(type):
...     def __instancecheck__(self, value):
...         return isinstance(value, int) and value % 2 == 1
...
>>> class Odd(metaclass=OddMeta):
...     pass
...
>>> isinstance(2, Odd)
False
>>> isinstance(3, Odd)
True
>>> isinstance(4, Odd)
False
>>> isinstance(5, Odd)
True
```

Fazendo um método ser aceito como inteiro!

Os *dunders* `__int__`, `__float__` e `__complex__` podem ser usados p/ um objeto poder ser convertido em um dos 3 tipos homônimos.

```
>>> class Something(object):
...     value = 0
...     def __int__(self):
...         self.value += 1
...         return self.value
...
>>> s = Something()
>>> 5 + int(s)
6
>>> 5 + int(s)
7
>>> 5 + int(s)
8
>>> 5 + s
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and
...
↩ 'Something'
```

E com herança?

```
>>> class Sucessor(int):  
...     def __new__(cls, value):  
...         return super().__new__(cls, value + 1)  
...  
>>> Sucessor(7)  
8  
>>> type(Sucessor(7))  
<class '__main__.Sucessor'>
```

FIM!