PYTHON ACADEMY

BIBLIOTECA ITERTOOLS DO PYTHON

Nesse ebook você vai dominar a biblioteca itertools do Python e adicionar mais um módulo poderosíssimo à sua caixa de ferramentas!

PYTHONACADEMY.COM.BR

Hoje vamos aprender sobre uma biblioteca “matadora” do Python: a famosa `itertools`!

Com ela, você vai impressionar qualquer recrutador mundo afora! ❤️

A *itertools* provê uma série de funções de manipulação de iteráveis para se construir iteradores complexos e úteis!

Ainda não sabe o que é um iterador?

Então já corre pro nosso post completo sobre Iterators e Generators!

Vamos nessa!

Introdução

O módulo `itertools` é definido pela própria documentação como:

> *The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination.*

Traduzindo:

> *O módulo padroniza um conjunto básico de ferramentas rápidas e eficientes em termos de memória que são úteis isoladamente ou em combinação.*

Ou seja, é uma boa pedida quando o assunto é eficiência (e em todos os casos em geral)!

O módulo forma uma "álgebra iterativa", tornando possível construir ferramentas especializadas de forma sucinta e eficiente em Python puro.

O módulo traz três categorias de iteradores:

- Iteradores infinitos (*Infinite Iterators*)
- Iteradores combinatórios (*Combinatoric Iterators*)
- Iteradores de encerramento (*Terminating Iterators*)

Vamos tratar de cada tipo agora!

Iteradores Infinitos

Iteradores são objetos que permitem percorrer todos os elementos de uma coleção, independentemente de sua implementação específica.

Eles são bem conhecidos por sua capacidade de serem usados em loops `for _ in iterator`.

Listas, tuplas e *sets* são exemplos de iteráveis que podem ser percorridos por iteradores.

Os iteradores podem ser finitos ou infinitos, ou seja, a iteração sobre seus elementos pode ter um fim (caracterizado pela exceção `StopIteration`) ou pode ser que um iterador nunca se esgote.

Esses tipos de iteradores, que nunca acabam, são conhecidos como **iteradores infinitos**.

Existem três funções do `itertools` que operam sobre iteradores infinitos, que são: `count()`, `cycle()` e `repeat()`.

Função `count`

Assinatura da função: `count(start=0, step=1)`

Esta função não opera sobre um iterável de entrada e sim cria um conjunto de dados infinitos.

Ele começa imprimindo a partir do início especificado pelo parâmetro `start` e segue infinitamente.

Se o parâmetro `step` for definido, os números serão pulados de `step` em `step`, caso contrário, o valor *default* será aplicado (que é 1 por padrão, ou seja, de 1 em 1).

Veja o exemplo abaixo de como utilizá-lo em loops `for`, imprimindo apenas os valores dispostos nos índices pares da lista de entrada, depois os valores disponíveis nos índices ímpares:

```python
from itertools import count

entrada = [
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
  12, 13, 14, 15, 16, 17, 18, 19, 20
]

# Imprime entrada[0], entrada[2], ...
for i in count(0, 2):
  if i >= len(entrada):
    break
  else:
    print(entrada[i], end =" ")

# Imprime entrada[1], entrada[3], ...
for i in count(1, 2):
  if i >= len(entrada):
    break
  else:
    print(entrada[i], end =" ")
```

```
1 3 5 7 9 11 13 15 17 19
2 4 6 8 10 12 14 16 18 20
```

Função `cycle`

Assinatura da função: `cycle(iterable)`

Esta função imprime todos os valores em ordem a partir do iterável `iterable` passado como parâmetro.

Ele reinicia a impressão do início novamente quando todos os elementos são impressos de maneira cíclica.

Entenda melhor no exempo a seguir:

```python
from itertools import cycle

contador = 0
entrada = [1, 2, 3, 4, 5, 6, 7]

for i in cycle(entrada):
  if contador > len(entrada) * 3:
    break
  else:
    print(i, end = " ")
    contador += 1
```

```
1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1
```

Função `repeat`

Assinatura da função: `repeat(object[, times])`

Este iterador imprime repetidamente o objeto `object` passado como parâmetro. Se o parâmetro opcional `times` for definido, ele imprimirá repetidamente um número `times` de vezes.

Exemplo:

```python
from itertools import repeat

repeater = repeat(10, 3)
print(next(repeater))
print(next(repeater))
print(next(repeater))
print(next(repeater))
```

```
10
10
10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Nesse exemplo, é utilizada outra forma de percorrer o iterador, utilizando a função `next`.

A cada chamada, ela busca o próximo valor do iterador, lançando a exceção `StopIteration` quando o iterador se esgota.

Como foi criado um iterador com o valor 10 repetido 3 vezes, ao tentar chamar a quarta iteração com `next`, é lançada a exceção, demonstrando que o iterador já se esgotou.

> 💡 *Estou desenvolvendo o **DevBook**, uma plataforma que usa IA para gerar ebooks técnicos profissionais. Te convido a conhecer!*

Iteradores Combinatórios

O módulo `itertools` nos traz 4 funções que simplificam o uso de análises combinatórios, como permutações e combinações, por exemplo.

São elas: `product`, `permutations`, `combinations` e `combinations_with_replacement`.

Agora vamos aos detalhes!

Função `product`

Assinatura da função: `product(*iterables, repeat=1)`

Essa função computa o **produto cartesiano** dos iteráveis de entrada.

Para quem não se lembra: produto cartesiano é a multiplicação entre pares ordenados de conjuntos distintos.

Note que o parâmetro `*iterable` possibilita a passagem de um número variável de entradas.

Para calcular o produto de um iterável consigo mesmo, utiliza-se o argumento opcional `repeat` para especificar o número de repetições.

Por exemplo: considere os conjuntos `(1, 2)` e `(a, b)`. O resultado do produto cartesiano desse dois conjunto será `(1, a)`, `(1, b)`, `(2, a)` e `(2, b)`

A saída desta função são tuplas ordenadas.

Alguns exemplos:

```
from itertools import product

print(list(product([1, 2], repeat = 2)))
print(list(product([1, 2], [1, 2])))      # Mesma saída do exemplo aci-
        ma
print(list(product(['Python'], ['Academy', 'Rocks'])))
print(list(product([1, 2], [3, 4], [5, 6])))
```

```
[(1, 1), (1, 2), (2, 1), (2, 2)]
[(1, 1), (1, 2), (2, 1), (2, 2)]
[('Python', 'Academy'), ('Python', 'Rocks')]
[(1, 3, 5), (1, 3, 6), (1, 4, 5), (1, 4, 6), (2, 3, 5), (2, 3, 6), (2,
4, 5), (2, 4, 6)]
```

Função `permutations`

Assinatura da função: `permutations(iterable[, r])`

Utilizado para gerar todas as possíveis permutações de um iterável `iterable`.

A função trata todos os elementos como únicos baseado em suas posições, e não em seus valores.

Por conta dessa característica, a saída de `list(permutations([1, 1]))` será `[(1, 1), (1, 1)]` e não apenas `[(1, 1)]`.

O parâmetro `r` é o tamanho da permutação e se não especificado ele é definido como o comprimento do iterável de entrada.

Veja alguns exemplo:

```
from itertools import permutations

print(list(permutations([1, 2])))
print(list(permutations([1, 2, 3])))
print(list(permutations([1, 2, 3], r=2)))
```

```
[(1, 2), (2, 1)]
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

Função `combinations`

Assinatura da função: `combinations(iterable[, r])`

Essa função traz todas as combinações do iterável `iterable` de tamanho `r`, sem substituição.

Se você já estudou estatística para concurso se lembra dese tipo de questão: calcule a quantidade de possíveis combinações de X elementos, escolhidos de tanto em tanto 😆

Essa função é muito poderosa e pode auxiliar em diversos tipos de problemas diferentes.

Vamos analisar uma questão comum de entrevistas de emprego:

> *Você tem três notas de R$ 20, cinco de R$ 10, duas de R$ 5 e cinco de R$ 1. De quantas maneiras você pode pagar uma conta de R$ 100?*

A seguir uma possível solução, utilizando-se `combinations`:

```
from itertools import combinations

# Notas disponíveis
notas = [20, 20, 20, 10, 10, 10, 10, 10, 5, 5, 1, 1, 1, 1, 1]

soma_100 = []

# Loop sobre o tamanho de combinações possíveis
for n in range(1, len(notas) + 1):
  # Loop sobre as possíveis combinações verificando se somam 100
  for combinacao in combinations(notas, n):
    if sum(combinacao) == 100:
      soma_100.append(combinacao)

# Remove os itens repetidos
resultado = list(set(soma_100))

print('As possíveis combinações de notas que somam R$ 100 são:')
print(resultado)
```

```
[
  (20, 20, 20, 10, 10, 10, 5, 5),
  (20, 20, 20, 10, 10, 10, 5, 1, 1, 1, 1, 1),
  (20, 20, 10, 10, 10, 10, 10, 5, 5),
  (20, 20, 10, 10, 10, 10, 10, 5, 1, 1, 1, 1, 1),
  (20, 20, 20, 10, 10, 10, 10)
]
```

Função `combinations_with_replacement`

Assinatura da função: `combinations_with_replacement(iterable, r)`

Esta função retorna uma combinação de elementos de comprimento `r` a partir dos elementos do iterável `iterable`, com substituição.

Os elementos individuais podem se repetir, diferente do que acontece na função `combinations`.

Vejamos os mesmos exemplo, para ver a diferença:

```
from itertools import combinations

print(list(combinations_with_replacement([1, 2], r=1)))
print(list(combinations_with_replacement([1, 2], r=2)))
print(list(combinations_with_replacement([1, 2, 3], r=2)))
print(list(combinations_with_replacement('ABC', r=2)))
```

```
[(1,), (2,)]
[(1, 1), (1, 2), (2, 2)]
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C',
'C')]
```

Agora vamos entender as diferenças entre `combinations` e `combinations_with_replacement`.

Suponha as chamadas `combinations([1, 2, 3], r=2)` e `combinations_with_replacement([1, 2, 3], r=2)`, que resultam em `[(1, 2), (1, 3), (2, 3)]` e `[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]`, respectivamente.

Veja que a saída da segunda função inclui `(1, 1)` e `(2, 2)`. Essa é a diferença para `combinations`, isto é: elementos individuais podem se repetir.

E assim, terminamos os **Iteradores Combinatórios**!

Vamos ver agora os Iteradores de Encerramento (*Terminating Iterators*).

Iteradores de Encerramento

Os iteradores de encerramento são usados para processar iteráveis de entrada finitos e produzir saídas com base na função utilizada.

Os métodos presentes nessa subdivisão são: `accumulate`, `chain`, `chain.from_iterable`, `compress`, `dropwhile`, `filterfalse`, `grouby`, `islice`, `starmap`, `takewhile`, `tee` e `zip_longest`.

Agora vamos um à um.

Função `accumulate`

Assinatura da função: `accumulate(iterable[, func])`

Esta função recebe dois argumentos, o iterável `iterable` e a função `func` que será aplicada a cada iteração.

Se nenhuma função for passada, a função `operator.add` é utilizada por padrão.

Se o iterável de entrada estiver vazio, o iterável de saída também estará vazio.

A cada iteração um acumulador é utilizado para auxiliar no cálculo das iterações seguintes.

Vamos entender melhor com um exemplo!

```
import itertools
import operator

lista = [2, 2, 4]

# Função a ser utilizada: operator.add (padrão)
print(list(itertools.accumulate(lista)))
# Função a ser utilizada: operator.mul (multiplicação)
print(list(itertools.accumulate(lista, operator.mul)))
# Função a ser utilizada: operator.sub (subtração)
print(list(itertools.accumulate(lista, operator.sub)))
# Função a ser utilizada: min
print(list(itertools.accumulate(lista, min)))
```

```
[2, 4, 8]
[2, 4, 16]
[2, 0, -4]
[2, 2, 2]
```

Para entender, vamos pegar a primeira chamada `itertools.accumulate(lista)`.

Sabemos que a função `func` será a `operator.add` (padrão) que recebe dois valores como parâmetro e os soma.

O que `accumulate` vai fazer, iteração à iteração, é: - Na primeira iteração, o acumulador se inicia em 0 e soma o primeiro elemento da lista de entrada, que no caso é `2`. - Em seguida, o acumulador está com o valor 2, e soma o segundo elemento da lista, portanto `2 + 2 = 4`. - Na última iteração, o acumulador está em 4 e recebe o valor 4 para ser somado, totalizando `8`. - Dessa forma, a saída será a lista `[2, 4, 8]`

Também podemos definir funções e passar como o parâmetro `func`, da seguinte forma:

```
import itertools

lista = [2, 2, 4]

exponenciacao = lambda a, b: a ** b

print (list(itertools.accumulate(lista, exponenciacao)))
```

```
[2, 4, 256]
```

Função `chain`

Assinatura da função: `chain(*iterables)`

Como seu próprio nome sugere, sua função é encadear iteráveis.

Ela funciona esgotando os elementos do primeito iterável, seguindo para o próximo, esgotando-o e assim por diante.

Sua entrada permite um número variável de iteráveis.

Veja o exemplo abaixo:

```python
import itertools

lista = list([2, 2, 4] )
conjunto = set({1, 2, 3})
print(list(itertools.chain(lista, conjunto, range(5))))
```

```
[2, 2, 4, 1, 2, 3, 0, 1, 2, 3, 4]
```

Que coisa louca não? Juntar uma lista, um conjunto e um `range`

Mas é isso mesmo!

Essa função é muito útil quando se tem que encadear estruturas diferentes (desde que todas respeitem o protocolo *Iterator*)!

Função `chain.from_iterable`

Assinatura da função: `chain.from_iterable(iterable)`

Essa função é similar à `chain`, contudo recebe um contâiner de iteráveis (e não uma lista variável de iteráveis, como acontece na `chain`).

Veja o exemplo:

```python
import itertools

lista = list([2, 2, 4] )
conjunto = set({1, 2, 3})
serie = range(5)

# Container de dados
container = (lista, conjunto, serie)
print(list(itertools.chain.from_iterable(container)))
```

```
[2, 2, 4, 1, 2, 3, 0, 1, 2, 3, 4]
```

Função `compress`

Assinatura da função: `compress(data, selectors)`

Essa função é muito interessante!

Ela seleciona os elementos da entrada `data` através de booleanos presentes na entrada `selectors`, como se fosse uma máscara.

Por exemplo, se `data = ['A', 'B', 'C']` e `selectors = [True, False, True]`, a saída de `compress` seria: `['A', 'C']`.

Ou seja, ela utiliza as posições e valores de `selectors` para selecionar quais valores de `data` estarão presentes na saída!

Outro exemplo:

```
import itertools

entrada = [range(3), '0', (1, 2)]

print(list(itertools.compress(entrada, selectors=[1, 0, 0])))
```

```
[range(0, 3)]
```

Veja que o parâmetro `selectors` é fexível e também aceita 0's e 1's como booleanos!

Note também que ele não itera os dados de entrada automaticamente, por isso `range(3)` não foi transformado em elementos (0, 1 e 2).

Para isso, podemos chamar a função que acamos de ver: a `chain`!

Veja o exemplo abaixo:

```
from itertools import compress, chain

entrada = [range(3), '0', (3, 4)]
saida   = compress(chain.from_iterable(entrada), selectors=[0, 1, 1, 0,
        0, 1])

print(list(saida))
```

```
[1, 2, 4]
```

Dessa forma, `chain.from_iterable` cria o iterador encadeado, com todos os valores presentes e `compress` faz a seleção dos elementos!

Função `dropwhile`

Assinatura da função: `dropwhile(predicate, iterable)`

Essa função descarta (*drop*) elementos enquanto (*while*) a função `predicate` retorna `True`.

Após o primeiro retorno `False`, ela replica os elementos de entrada na saída.

Por exemplo: suponha que você tenha que você tenha que percorrer uma lista de entrada descartando elementos até que o primeiro valor `100` seja encontrado. Depois disso, os valores iguais à `100` não devem mais ser descartados.

Uma forma possível de resolver esse problema utilizando `dropwhile` seria:

```
import itertools

entrada = [
  0, 10, 2, 20, 34, 45, 40, 41, 63, 100,
  5, 1, 80, 100, 101, 142, 2, 4, 5
]

print(list(itertools.dropwhile(lambda x: x <= 100, entrada)))
```

```
[101, 142, 2, 4, 5]
```

Note que os primeiros 10 elementos são menores ou iguais à 100.

Dessa forma, nosso predicado `lambda x: x <= 100` retorna `True` em todos eles, *dropando-os*!

Função `filterfalse`

Assinatura da função: `filterfalse(predicate, iterable)`

Essa função filtra os valores do iterável de entrada `iterable` quando o resultado da função `predicate` for falso, elemento à elemento.

Veja como filtrar os elementos de uma lista, retirando os número primos.

```
import itertools

def verifica_se_primo(numero):
  if numero > 1:
    for i in range(2, int(numero/2)+1):
      resultado = False if (numero % i) == 0 else True
      return resultado
  else:
      return False

entrada = [
  10, 7, 20, 34, 43, 40, 41, 63, 100, 80, 100, 101, 142
]

print(list(itertools.filterfalse(verifica_se_primo, entrada)))
```

```
[10, 20, 34, 40, 100, 80, 100, 142]
```

Note que os números primos não estão presentes na saída!

Função `groupby()`

Assinatura da função: `groupby(iterable, key=None)`

Função utilizada para agrupar elementos de um iterável de entrada `iterable` pela função de seleção de chaves `key`.

A função `key` deve retornar onde a chave se encontra na estrutura de dados para proceder com o agrupamento de elementos.

Por exemplo, suponha que você tenha um conjunto de pedidos de um restaurante (composto por nome do prato e horário do pedido) e lhe fosse pedido que agrupasse os pedidos pelo nome do prato:

```python
import itertools

entrada = [
  ("spaghetti", "10:23:52"),
  ("spaghetti", "10:27:52"),
  ("pennete_rigate", "11:14:44"),
  ("pennete_rigate", "13:17:24"),
  ("ravioli", "11:45:33"),
  ("pizza", "19:45:44")
]

funcao_chave = lambda x : x[0]

for chave, grupo in itertools.groupby(entrada, funcao_chave):
  print({chave : list(grupo)})
```

```
{'spaghetti': [('spaghetti', '10:23:52'), ('spaghetti', '10:27:52')]}
{'pennete_rigate': [('pennete_rigate', '11:14:44'), ('pennete_rigate',
        '13:17:24')]}
{'ravioli': [('ravioli', '11:45:33')]}
{'pizza': [('pizza', '19:45:44')]}
```

A saída é uma estrutura de dados que contém os pedidos agrupados por nome!

Função `islice()`

Assinatura da função: `islice(iterable, stop)` ou `islice(iterable, start, stop[, step])`

Se você saber fazer *slice* (fatiamento) em strings (ex: `string[:-5:2]`), já sabe utilizar `islice`.

Essa função possui duas assinaturas: - Na primeira, recebe um iterável `iterable` de entrada e vai iterá-lo até o elemento definido pelo parâmetro `stop`. - Na segunda, recebe um iterável `iterable` de entrada, com um índice `start` de início do *slicing*, índice `stop` de parada e um parâmetro `step` de passo para pular elementos.

Vamos a alguns exemplos:

```
import itertools

entrada = [0, 1, 4, 5, 8, 10, 22, 30, 44, 59, 67, 74]

# Slicing do início até metade do iterável de entrada
print(list(itertools.islice(entrada, int(len(entrada)/2))))
# Slicing começando do índice 3 até o final
print(list(itertools.islice(entrada, 3, None)))
# Slicing do início até o fim, pulando de 2 em 2
print(list(itertools.islice(entrada, None, None, 2)))
```

```
[0, 1, 4, 5, 8, 10]
[5, 8, 10, 22, 30, 44, 59, 67, 74]
[0, 4, 8, 22, 44, 67]
```

Função `starmap()`

Assinatura da função: `starmap(function, sequence)`

Essa função é uma mão na roda! Dizem que já salvou muitas vidas 😆

A `starmap` recebe dois parâmetros: - Uma função `function` de entrada e - Uma sequência `sequence` de iteráveis.

Com isso, ela vai iterar sobre a sequência de iteráveis de entrada, passando seus elementos para `function` processar.

Suponha que você tenha um conjunto de dados bem heterogêneo e queira verificar se há números pares nos iteráveis de entrada.

Uma forma de fazer seria:

```
import itertools

entrada = [
  (1, 2),
  (1, 3, 5, 7, 9),
  range(10),
  {1, 2, 3, 4},
  (1, 1, 1, 1, 1, 1, 1, 1, 1)
]

# Função que verifica se algum dos elementos de entrada é par
def contem_par(*elementos):
  for elemento in elementos:
    if elemento % 2 == 0:
      return True
  return False

print(list(itertools.starmap(contem_par, entrada)))
```

```
[True, False, True, True, False]
```

A diferença para sua irmã-gêmea `map` é a forma na qual os dados de entrada estão dispostos.

Use `itertools.starmap` quando os dados já estiverem agrupados em um container de dados.

Função `takewhile()`

Assinatura da função: `takewhile(predicate, iterable)`

Essa função é o oposto da `dropwhile`.

Enquanto a `dropwhile` ignora elementos enquanto alguma condição é satisfeita, a `takewhile` vai **escolher** (*take*) elementos **enquanto** (*while*) a condição `predicate` for verdadeira.

Veja o exemplo:

```
import itertools

entrada = [1, 4, 5, 10, 20, 40, 100]

print(list(itertools.takewhile(lambda x: x <= 5, entrada)))
```

```
[1, 4, 5]
```

Função `tee()`

Assinatura da função: `tee(iterable, n=2)`

Cria um número `n` de iteradores a partir do iterável de entrada `iterable`.

Quando a chamada à função `tee` é realizada, o iterável de entrada não deve mais ser utilizado!

E atenção: de acordo com a documentação, `tee` não é *threadsafe*, isto é: não é seguro utilizar os iteradores resultantes de forma paralela!

Vamos agora ao exemplo:

```python
import itertools

# Iterável de entrada
entrada = [1, 2, 3, 4]

# Transforma nossa lista em um iterador com iter()
iterador = iter(entrada)

for i in itertools.tee(iterador, 3):
    print(f'Iterador: {list(i)} (Tipo = {type(i)}')
```

```
Iterador: [1, 2, 3, 4] (Tipo = <class 'itertools._tee'>)
Iterador: [1, 2, 3, 4] (Tipo = <class 'itertools._tee'>)
Iterador: [1, 2, 3, 4] (Tipo = <class 'itertools._tee'>)
```

Quer saber mais sobre formatação de strings com f-strings?

Então já deixa o próximo post engatilhado clicando aqui 😉

Função `zip_longest()`

Assinatura da função: `zip_longest(*iterables, fillvalue=None)`

Combina os elementos dos iteráveis de entrada `*iterables`.

Caso os iteráveis de entrada não sejam do mesmo tamanho, o valor de preenchimento `fillvalue` será usado.

Veja como utilizar no exemplo abaixo:

```python
import itertools

letras = ['a', 'b', 'c']
numeros = [1, 2, 3, 4, 5]

print(list(itertools.zip_longest(letras, numeros)))
print(list(itertools.zip_longest(letras, numeros, fillvalue="-")))
```

```
[('a', 1), ('b', 2), ('c', 3), (None, 4), (None, 5)]
[('a', 1), ('b', 2), ('c', 3), ('-', 4), ('-', 5)]
```

BÔNUS! Questão de entrevista

Agora, vamos analisar uma questão de entrevista de emprego que poderia ser resolvida utilizando-se o módulo `itertools`!

Atenção ao enunciado da questão:

> *Escreva uma função que imprima o menor inteiro que não está presente em uma determinada lista e não pode ser representada pela soma dos subelementos da lista.* ***Exemplo****: Para* `a = [1, 2, 5, 7]` *o menor número inteiro não representado pela lista ou uma fatia da lista é 4, e se* `a = [1, 2, 2, 5, 7]` *então o menor inteiro não representável é 18.*

Em outras palavras: *Dados os somatórios possíveis combinando-se números de uma lista de entrada, qual o menor número que não pode ser representado?*

Para isso, é necessário calcular as possíveis combinações de tamanho 1, depois de tamanho 2 e assim sucessivamente, até a última combinação, com todos os número da lista de entrada.

Em seguida, calcular os somatórios dessas combinações, removendo os números repetidos (pense na entrada `a = [1, 2, 2]`: as somas `a[0] + a[1]` e `a[1] + a[2]` terão o mesmo resultado).

Por último, iterar de 1 em 1 iniciando em 0 até o maior somatório, verificando se o número consta na lista de possíveis somatórios, e caso não esteja presente, esse valor é retornado como **resultado**!

Para essa lógica, o código abaixo poderia ser escrito:

```python
import itertools

def menor_inteiro_nao_representavel(lista):
  # Cria lista auxiliar para guardar possíveis somatórios
  somatorios = []

  # Itera sobre o tamanho das possíveis combinações
  for tamanho_combinacao in range(0, len(lista) + 1):
    # Calcula todas as possíveis combinações de tamanho "tamanho_combinacao"
    for combinacao in itertools.combinations(lista, tamanho_combinacao):
      # Soma os valores presentes na combinação
      somatorios.append(sum(combinacao))

  # Cria uma nova lista, removendo os elementos repetidos e ordenando-os com sort
  nova_lista = list(set(somatorios))
  nova_lista.sort()

  # Itera de 0 até o maior elemento da lista de somatórios +2 de padding para o
  # caso de todos os somatórios serem possíveis de serem representados
  for elemento in range(0, nova_lista[-1] + 2):
    if elemento not in nova_lista:
      return elemento

# Entradas
lista_1 = [1, 2, 5, 7]
lista_2 = [1, 2, 2, 5, 7]
resultado_1 = menor_inteiro_nao_representavel(lista_1)
resultado_2 = menor_inteiro_nao_representavel(lista_2)

print(f'O menor inteiro representável da lista {lista_1} é {resultado_1}')
print(f'O menor inteiro representável da lista {lista_2} é {resultado_2}')
```

```
O menor inteiro representável da lista [1, 2, 5, 7] é 4
O menor inteiro representável da lista [1, 2, 2, 5, 7] é 18
```

Congratulations! You're hired!

Conclusão

Com isso, finalizamos mais um post!

Esse post serviu pra você aumentar ainda mais seu leque de ferramentas com o poderosíssimo módulo `itertools`!

Pronto pra arrasar na próxima entrevista de emprego?!

E você... Já utilizou o módulo `itertools` em alguma entrevista?

Conta pra mim aqui embaixo! Gostaria de saber!

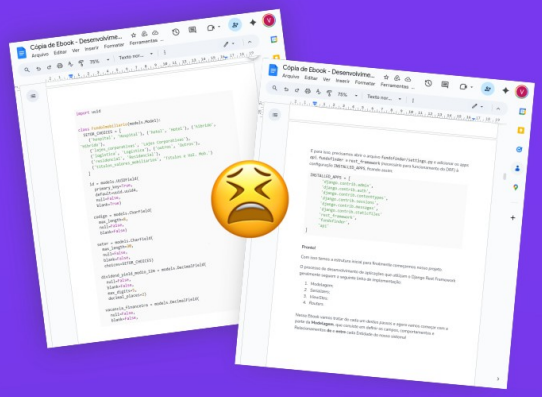
Valeu e até o próximo post!

Não se esqueça de conferir!

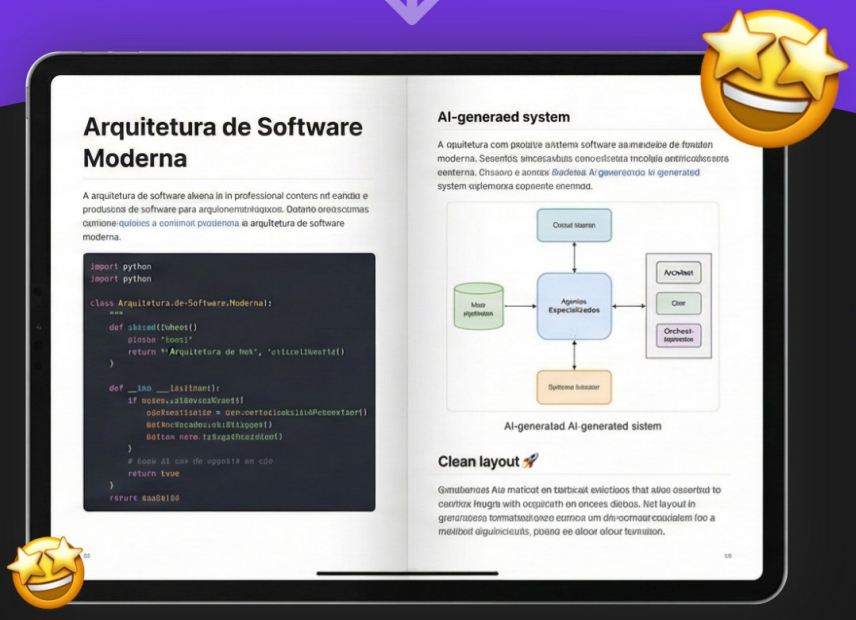
DevBook

Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código**!



Chega de formatar código no Google Docs



Syntax Highlight

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS