

Introdução a Python

IPL 2021



MIT | MIT BRAZIL



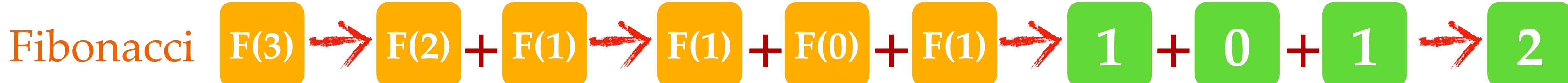
Recursão

- Como vimos, é possível que uma função chame outra: *também é possível que a função chame a si mesma!*
- Pode não parecer muito, mas essa capacidade torna a expressão de certos cálculos muito mais fácil
- Alguns exemplos naturais incluem fatoriais, Fibonacci e sequências aninhadas
- Funções recursivas precisam de duas propriedades centrais: **caso base** e **passo recursivo**

- Caso simples que pode ser resolvido sem recursão
- *Casos em que a resposta é conhecida, geralmente por definição*
- Ex: $0! = 1! = 1$, $\text{Fib}(0) = 0$, $\text{Fib}(1) = 1$

caso recursivo caso base

- Depende da definição da função
- Reduz o problema em direção a um caso base
- Ex: $5!$ depende de $4!$, que é mais perto do caso base $1!$



Recursão: exemplos

```
1 def fib(n):
2     if n <= 1: # caso base
3         # fib(0) = 0, fib(1) = 1
4         return n
5     # passo recursivo
6     return fib(n - 1) + fib(n - 2)
7 |
```

$$\begin{aligned} \text{fib}(3) &= \text{fib}(2) + \text{fib}(1) \\ &= \text{fib}(1) + \text{fib}(0) + \text{fib}(1) \\ &= 1 + \text{fib}(0) + \text{fib}(1) \\ &= 1 + 0 + \text{fib}(1) \\ &= 1 + 0 + 1 \\ &= 2 \end{aligned}$$

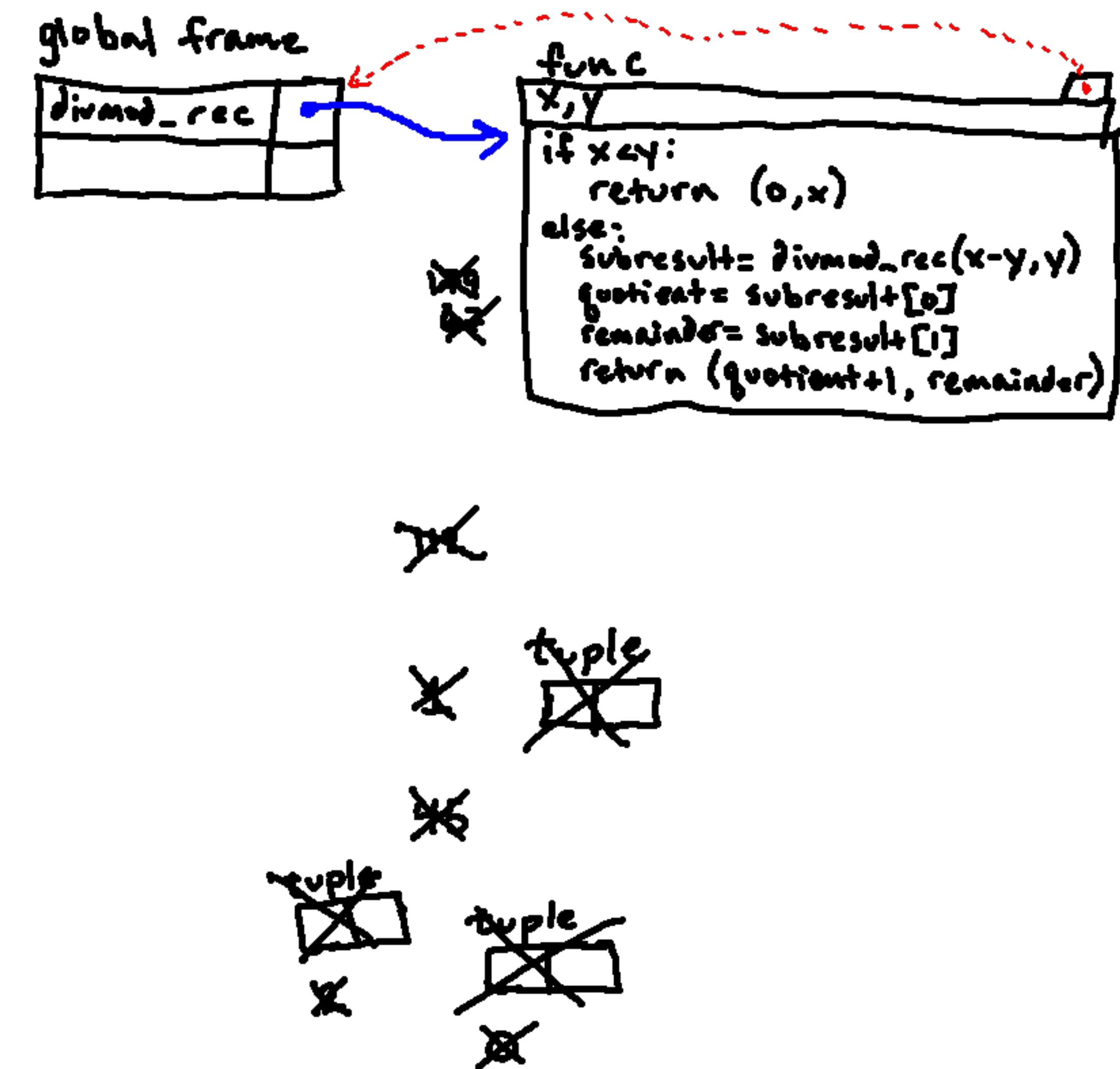
Algoritmo Euclidiano para máximo divisor comum

```
1 def gcd(a, b):
2     if a < b:
3         # passo recursivo 1
4         return gcd(a, b - a)
5     elif b < a:
6         # passo recursivo 2
7         return gcd(a - b, a)
8     # caso base: a == b
9     return a
10 |
```

$$\begin{aligned} \text{gcd}(18, 12) &= \text{gcd}(18 - 12, 12) \\ &= \text{gcd}(6, 12) \\ &= \text{gcd}(6, 12 - 6) \\ &= \text{gcd}(6, 6) \\ &= 6 \end{aligned}$$

Recursão: exemplos

```
1 def divmod_rec(x, y):
2     if x < y: # caso base: quociente 0
3         return 0, x
4     else: # passo recursivo
5         # chamar a função recursiva não
6         # precisa ser no último passo
7         subresult = divmod_rec(x-y, y)
8         quotient = subresult[0]
9         remainder = subresult[1]
10        return quotient + 1, remainder
11
12 print(divmod_rec(179, 67))
13
14
```



Recursão vs. iteração

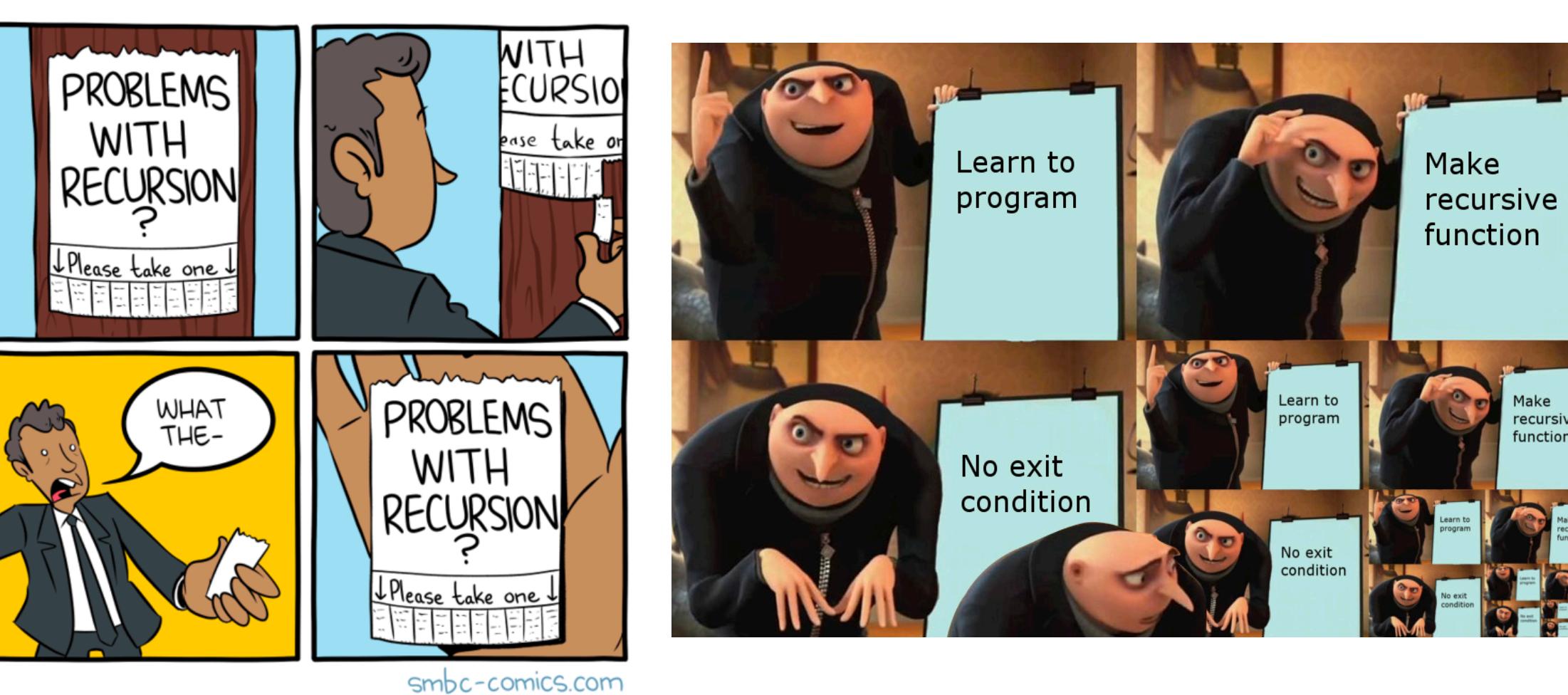
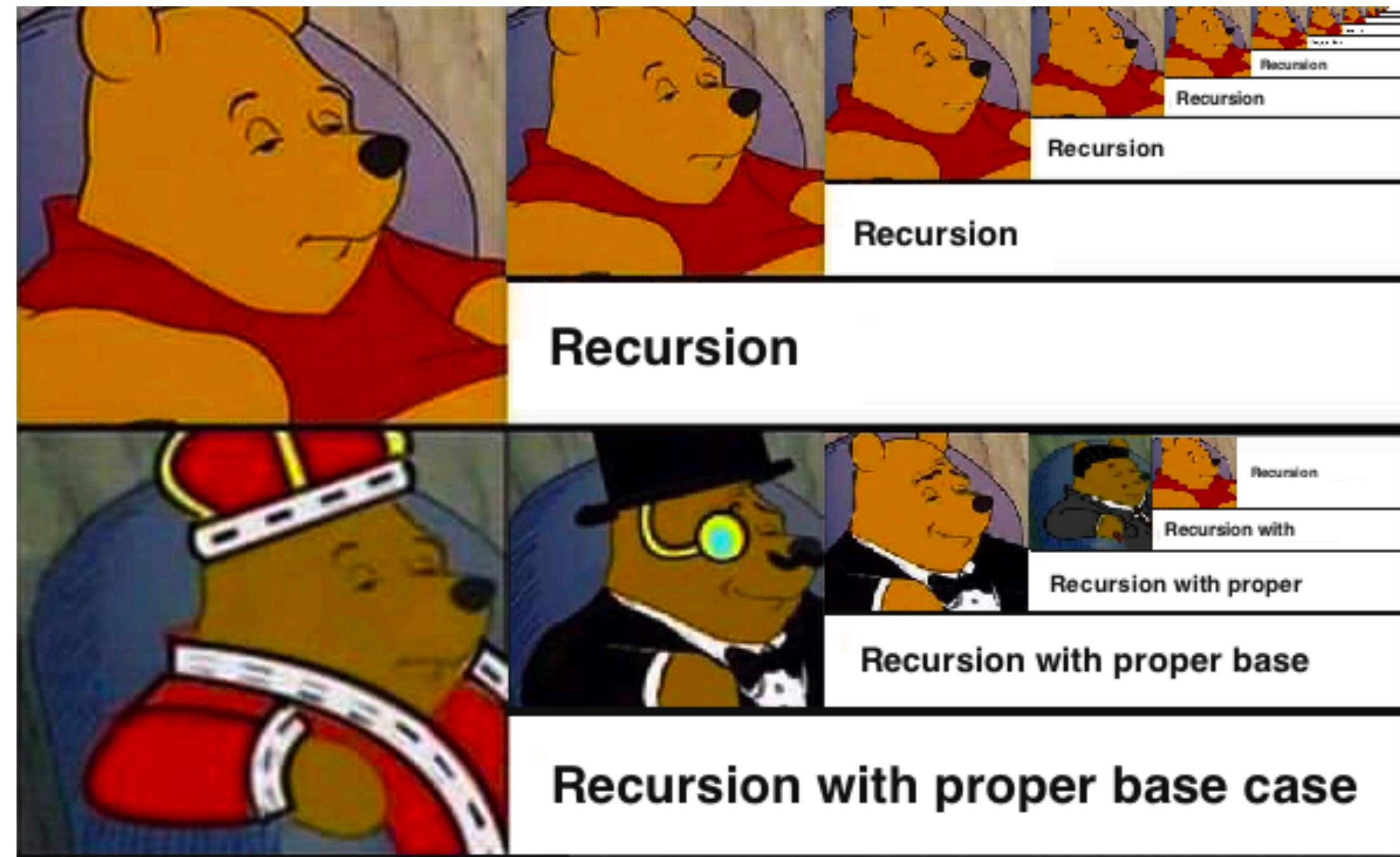
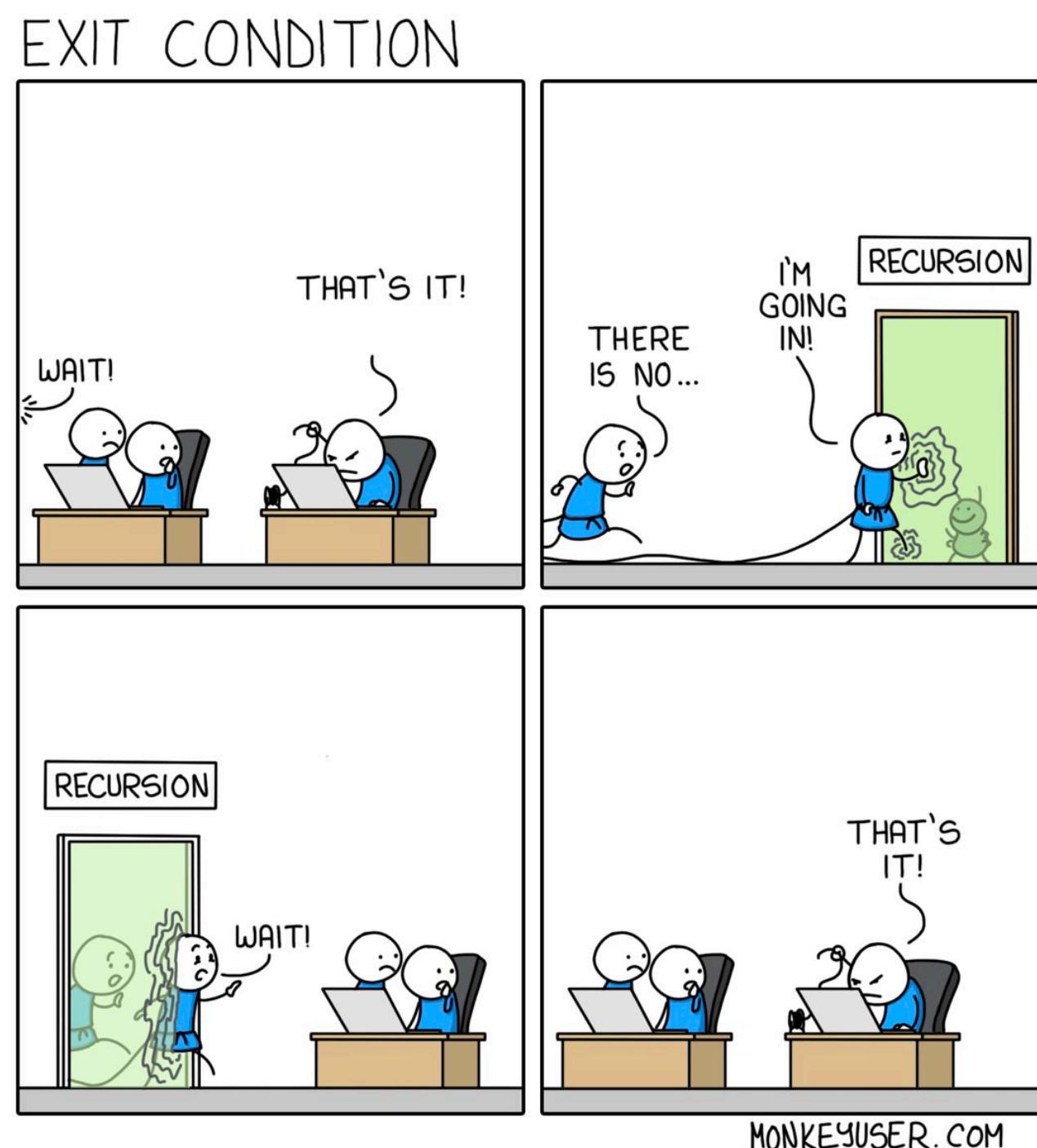
- Em alguns casos, a solução iterativa é melhor: pode ser mais curta e fácil de ler/entender
- Mas há também muitos casos em que recursão é mais intuitiva e mais relacionada ao problema
- Em geral, é possível reescrever qualquer programa iterativo usando recursão e vice-versa
- Não há resposta certa se iteração ou recursão é melhor no *sentido geral*: depende tanto do problema quanto das preferências pessoais do programador
- Em maior detalhe:
 - Recursão é geralmente mais intuitiva e mais parecida com a abordagem natural a um problema: mais legível e clara
 - Iteração quase sempre tem performance melhor e usa significativamente menos memória

```
1 def factorial_iter(n):  
2     prod = 1  
3     while n >= 1:  
4         prod *= n  
5         n -= 1  
6     return prod  
7  
8  
9  
10 def factorial_rec(n):  
11     if n <= 1: # caso base  
12         return 1  
13     return n * factorial_rec(n - 1)  
14  
15 |
```

Legibilidade: **recursão**
Performance: **iteração**

Recursão infinita

- Funções recursivas chamam a si mesmas até que um caso base seja atingido
- Se o caso base não for atingido corretamente (ou se não existir), corremos o risco de *recursão infinita* – muito parecido com a ideia de loop infinito
- **Sempre cheque que seu caso base não depende da função recursiva em si e é corretamente atingido para *todas* as entradas possíveis**



Dicionários

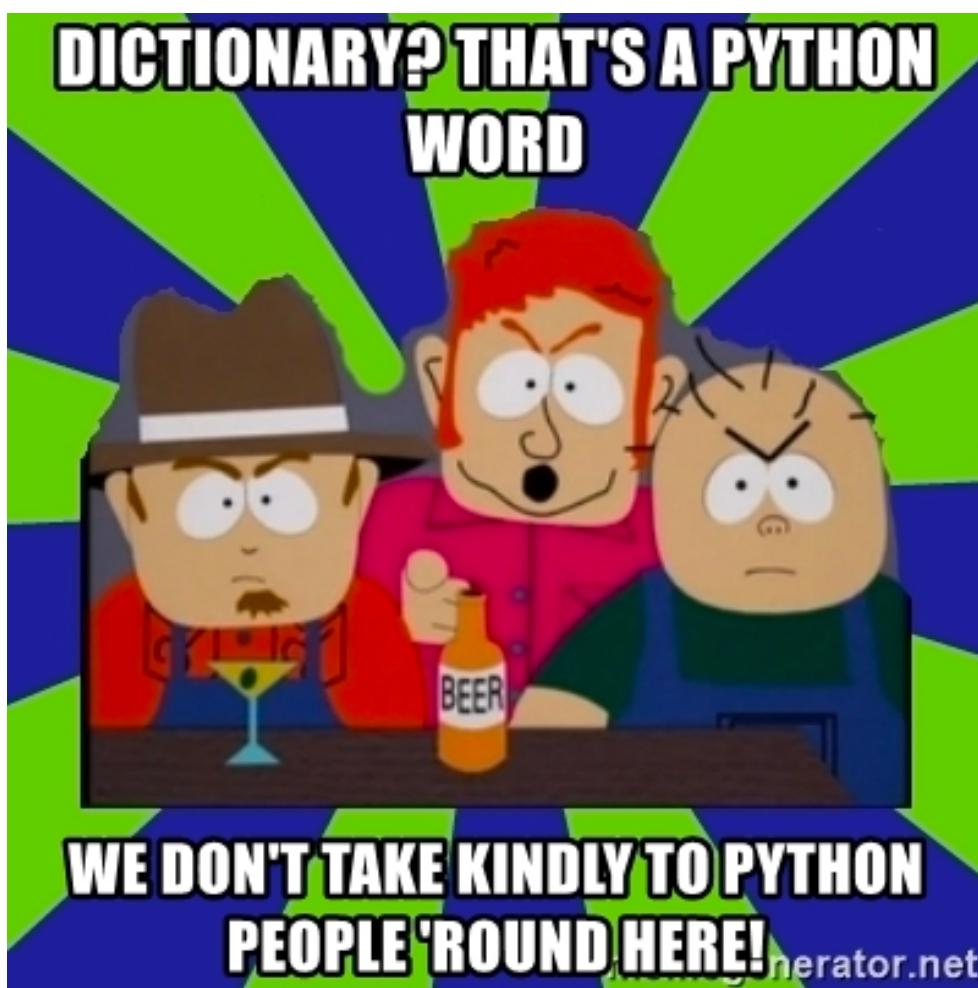
- Dicionários são outro tipo composto integrado e um dos melhores recursos de Python

Chaves e valores

- Dicionários são mapeamentos de **chaves** a **valores**; um par chave-valor é um **item**
- O tamanho de um dicionário é o número de items que contém: `len(x)` retorna quantas chaves/valores x tem

Dicionários e listas

- Dicionários e listas têm algumas semelhanças: ambos são compostos e mutáveis
- Contudo, dicionários são mais gerais: enquanto em uma lista os índices devem ser inteiros, *chaves podem ser de (quase) qualquer tipo em dicionários*



list índice

0	1	2	3	4
17	9	"oi"	True	-10

x = [17, 9, "oi", True, -10]

Índices precisam ser inteiros!

dict chave

"C"	(3, 4)	0	"T"	-10
17	9	"oi"	True	-10

Chaves podem ser (quase) qualquer coisa!

Representação de dict em diagramas de ambiente: muito parecida com um quadro, mapeamento entre chaves e valores!

```
1 # chave: valor
2 d = {"c": 17, (3, 4): 9, 0: "oi",
3       "T": True, -10: -10}
4
5 print(len(d)) # 5
6 |
```

Dicionários: sintaxe

- Dicionários são criados com chaves, {}

Adicionando items

- Elementos podem ser adicionados por colchetes (mesma notação de indexação)
 - **x[chave] = valor** cria um mapa de chave a valor
- Alternativamente, podemos indicar items quando criamos o dicionário:
 - **x = {chave: valor}** cria o mesmo mapa

Procurando items

- Elementos são procurados por chave, não posição, usando a mesma notação de colchetes
 - Ordem dos objetos em dicionário não importa!

Mudando items

- Mesma notação de adicionar, usando colchetes (basicamente “adiciona” de novo, sobrescrevendo)

```
1  d1 = {} # dicionário vazio
2
3  # elementos podem ser adicionados com [k]
4  # em que k é uma chave nova ou atual (sobrescreve)
5  d1["gato"] = "Katze"
6
7  print(d1) # {"gato": "Katze"}
8
9  # outra forma de criar com alguns valores já
10 d2 = {"gato": "Katze", "cavalo": "Pferd",
11      "esquilo": "Eichhörnchen"}
12
13 print(d2) # {"gato": "Katze", "cavalo": "Pferd",
14      "esquilo": "Eichhörnchen"}
15
16 # pesquisados por chave, não ordem
17 print(d2["gato"]) # "Katze"
18 # ausência de chave: erro
19 # print(d2["coelho"]) daria um KeyError
20
21 # modificamos da mesma forma que criamos:
22 d1["gato"] = "cat"
23 print(d1) # {"gato": "cat"}
24 |
```

Dicionários: chaves

- Chaves são *únicas* em dicionários (não repetidas), mas valores podem ser repetidos: duas chaves podem apontar até para o mesmo objeto

Restrições

- Embora mais gerais que listas, nem todos os objetos em Python podem ser chaves de dicionários
- Em particular, *objetos mutáveis não podem ser chaves em dicionários*
 - Permitidos:** int, float, str, bool, tuple
 - Proibidos:** list, dict, set
- Não há restrições para valores de dicionários

Operações

`len(d)`

Retorna o número de itens (pares chave-valor) em d

`x in d`

Retorna **True** se x é uma chave do dicionário d

`d.get(x, a)`

Tenta retornar $d[x]$, mas retorna a se x não for uma chave do dicionário d

`d.keys()`

Retorna a sequência das chaves de d

`d.values()`

Retorna a sequência dos valores de d

`d.items()`

Retorna a sequência de pares (chave, valor) de d

```
1     d1 = {"oi": "ola", 1: 1,
2         2.0: 2.0, (3, 4): (3, 4),
3         3: 5, "yo": [1, 2, 3],
4         10: {1: 1}}
5
6     # d2 = {[1, 2]: [1, 2]}
7     # erro: chave não pode ser list
8
9     # d3 = {{1: 1}: 1}
10    # erro: chave não pode ser dict
11
12    print(len(d1)) # 7
13    print("oi" in d1) # True
14    print("oi" not in d1) # False
15    print("ola" in d1) # False
16
17    print(d1.get("oi", 20)) # "ola"
18    print(d1.get("ola", 20)) # 20
19
20    for c, v in d1.items():
21        print(c, v)
22        # "oi" "ola"
23        # 1 1
24        # 2.0 2.0
25        # ... (ordem aleatória)
26        |
```