□□□□□□□□□□

□1□

られるオブジェクト

ストにも、**辞書にも格納**

り値にも

```
x

    # □□□□□□□□□
    # ==> 25
    # ==> 'square'
```

□□□□□

要。別の型も代入可能

き換え可能、辞書はキーから値を参照

字下げ、スペースは4個

```
= 42, "Alice", True

   # □を持つのは値


ks.pop()

      # 無いキーでも落ちない

 range(5) if x % 2 == 0]
```


い付録へ

□□□□□

ると、演算子の数だけ関数が伸長

に入れ、名前で処理を参照

えても `calc` は1行のまま

```
, b):
add":
 a + b
 "sub":
 a - b

 -1
```

```
r
operator.add, "sub": operator.sub, "mul": operator.mul}
```

□□□□

言語では、分岐のたびに再ビルド

は辞書に関数を足すだけ

して扱えるため、選択肢を表に保持

```
ax          # □□で語彙がひとつ増える
 9)         # 9
```

しが、動的言語の面白さ

□□□□□□□□

□□□□□□

別の関数を定義可能

は、外側の引数 `n` を保持したまま動作

ージャと呼ぶ

```
(n):
x):
 x * n      # □側の n を覚えたまま
er

blier(2)
 20
```

□□□□

□関数を受け取り、別の関数を返す

`= logged(add)` と書くのと意味は同一

「定義の直後に `f = deco(f)`」と読む

```
c):
r(*args):
f"{func.__name__}{args}")
 func(*args)
oper


 b
dd)
```

□□□□□

クロージャが保持する辞書

書のキーになるため、引数ごと結果を保存

はそのまま、呼び出し方だけを包む

```
:

r(*args):
s not in cache:
che[args] = func(*args)
 cache[args]
pper


, z):
```

☒☒☒☒☒

本文は1文字も変更なし

再計算をやめただけ

呼び出しは約5万分の1

)	呼び出し回数	時間
	12,604,861 回	約 1 秒
	225 回	一瞬

。一度出した答えを覚えておけば足りる

□□□□□□

は関数を辞書に足して、そのまま返す

`("shout")` の時点で `COMMANDS` に載る

が一度に済み、一覧がずれない

```
me):
unc):
OS[name] = func
 func
p

t")
):
```

命令、残りが引数

れば、知らない命令として破棄

処理系を起動する骨格はこれで十分

```
s = line.split()
t in COMMANDS:
ValueError(f"□□□□□□: {name}")
MANDS[name](*args)

lo")    # 'HELLO!!'
```

□□□□□□□□

変数にも辞書にも格納

チテーブル。`if` の連鎖が辞書ひとつに

は中身を変えずに機能を追加。`@deco` は `f = deco(f)`

、定義と同時に表が完成

□□□□□□

ソッドを1つも定義していない

`r__` が、名前を見た瞬間に関数を返す

定義を探しても見つからない

```
ttr__(self, name):
thod(*args):
int(f"{name} □ {args} で呼ばれました")
 method



')
```

□□□□

loat、`//` は切り捨て

ふれは起きない。その代わり遅い

更不可。`s[0] = "X"` はエラー

```
3.5
3
]あふれしない


[0:3], s[::-1]
 {score:6.2f}"
                    # print デバッグ用
```

ほとんどの文字列処理は十分

`< x < 100` と連鎖可能

tr / dict、`0`、`None` は False

くなったら `range(len(...))` より `enumerate` か `zip`

```
 enumerate(names):
(names, scores):
```

□□□□

□元を書き換えて `None` を返す

数で受け取ると、呼び出し元まで変化

値の `[]` は、定義時に1回だけ生成

```
        # □違い。sorted(xs) を使う
)

em):
item)    # 呼び出し元のリストが書き換わる


xs=[]):
item)    # bad(1) -> [1],  bad(2) -> [1, 2]
```

□□□□□□□□□□

□□□□□□□□

た道具を、課題の順に手を動かして確認

は、前の課題の結果を部品として利用

ではない

	ねらい
を実装し、呼び出し回数を数える	再帰、爆発する計算量
ッチテーブルで電卓	辞書に関数を入れる
を書き、課題1 に組み込む	デコレータ、クロージャ
`nd` と `run()` でコマンド表	字句解析、ディスパッチ
帰の木を表示 / `__doc__` から `help` を自動生成	

■■■■■

開く」から課題を開き、「ドライブにコピーを保存」

`ODO` を埋め、直下の確認セルで `OK` を確認

消さない

」のテストケースで採点するので消しても害はない

課題1に組み込むと、関数の中身を変えずに呼び出し回数を削減