

# 再帰 Recursion

情報科 飯島 涼



# (復習) 関数とは

入力した数字や文字に対して，処理をした結果を出力するもの



処理ごとに，関数名が割り当てられる.

# (復習) 関数の使い方

関数名(引数1, 引数2, ... ..., 引数n)

のように記述する.

例1: 小さい数を入力する関数

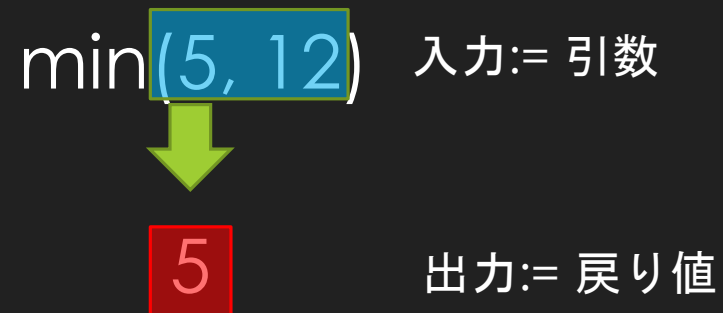
print(min(5, 12))



プログラム内で下のように置き換えられる

print(5)


=> 5が出力される.



# 関数の処理の流れ

① add\_tax関数を呼び出す (5行目)と、関数定義の先頭に処理が移動する.

```
1 def add_tax(price, rate):  
2     added = int( price * (1 + rate) )  
3     return added  
4  
5 print(add_tax(100, 0.1))
```



②引数の部分に書かれた変数に、5行目で指定した引数の値が代入される.

```
1 def add_tax(price=100, rate=0.1):  
2     added = int( price * (1 + rate) )  
3     return added  
4  
5 print(add_tax(100, 0.1))
```

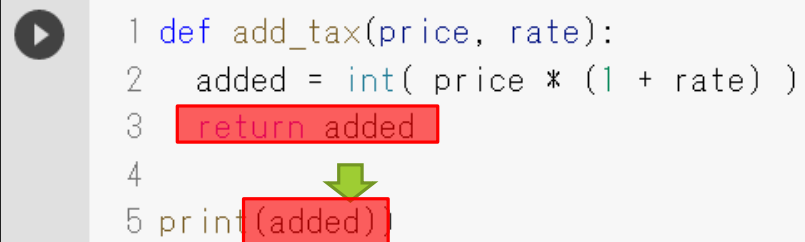
price = 100  
rate = 0.1

# 関数の処理の流れ

③ 代入されたprice, rateを使って処理が行われる (2行目).

```
1 def add_tax(price, rate):  
2     added = int( price * (1 + rate) )  
3     return added  
4  
5 print(add_tax(100, 0.1))
```

④関数を呼び出した場所を, 戻り値の値で置き換える (超重要) .



The diagram illustrates the flow of data from the function back to the caller. It shows the same code as before, but with a red box around the `return added` statement on line 3. A green arrow points from this box down to a red box around the `added` argument in the `print` statement on line 5. A play button icon is visible on the left side of the code editor.

```
1 def add_tax(price, rate):  
2     added = int( price * (1 + rate) )  
3     return added  
4  
5 print(added)
```

110

⑤ 置き換えられた値が実行される.

# 本日の内容

- 再帰（教科書掲載なし、プログラミングコンテスト攻略のためのアルゴリズムとデータ構造 p140- p149）
  - 再帰とは
    - 関数の中で、自分自身を呼び出す関数
  - 再帰の利用例
    - 階乗
    - フィボナッチ数列

# 再帰とは

- 関数の中で、自分自身を呼び出す関数

- 最も簡単な例（nの階乗）

```
1 def factorial(n):  
2     if (n == 1):  
3         return 1  
4     return n * factorial(n-1)  
5  
6 print(factorial(5))  
7
```

関数定義の中で同じ関数名を呼び出し

- いままでの例

```
1 def add_tax(price, rate):  
2     added = int( price * (1 + rate) )  
3     return added  
4  
5 print(add_tax(100, 0.1))
```

関数定義の外でのみ呼び出し

# 再帰とは（よく使われる単語）

- 再帰呼び出し
  - 関数定義の中で、自分の関数を呼び出すこと
- 再帰関数
  - 再帰呼び出しを行う関数のこと
- 再帰的な処理、再帰的に～
  - 再帰関数を連想させるような処理
  - 再帰関数で再現可能な処理



# 再帰関数の考え方のポイント

関数を呼び出した部分は、関数の戻り値に置き換わることを考える


- 考えやすくするために、以下の再帰関数について、処理を確認します  
(これは実行しないでください)

```
1 def factorial(n):  
2  
3     return n * factorial(n-1)  
4  
5 print(factorial(5))
```

# 再帰関数の処理の流れ


- ① 5行目の、factorial(5) が、  
1行目の関数定義def factorial(n)を呼び出す

```
1 def factorial(n):  
2  
3     return n * factorial(n-1)  
4  
5 print(factorial(5))
```



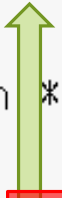
- ② 5行目の関数呼び出しの部分が、  
戻り値の値に置き換わる

```
1 def factorial(n):  
2  
3     return n * factorial(n-1)  
4  
5 print(5 * factorial(4))
```




③5行目の factorial(4) が、  
1行目の関数 factorialを呼び出す

```
1 def factorial(n):  
2  
3     return n * factorial(n-1)  
4  
5 print(5 * factorial(4))
```



④ 5行目の関数呼び出しの部分が、  
戻り値の値に置き換わる

```
1 def factorial(n):  
2  
3     return n * factorial(n-1)  
4  
5 print(5 * 4 * factorial(3))
```



factorial(3)以降も、①=>②の流れを繰り返すと  
次のようになる

# 永遠に終わらない関数 factorial(n)

- factorialの中身は徐々に減り続け、永遠に関数は終わらなくなってしまう。  
ちゃんとしたエラー処理の機構がなければクラッシュする可能性もある

```
1 def factorial(n):  
2  
3     return n * factorial(n-1)  
4  
5 print(5 * 4 * 3 * 2 * 1 * 0 * -1 * -2 * ... ..)
```

階乗で計算したいのはここまで  
=> 関数にif文を加えて、関数の再帰呼び出し

# いつかは終わる関数 factorial(n)

- 先ほどのfactorial関数に、緑枠の部分を追加

```
1 def factorial(n):  
2     if (n==1):  
3         return 1  
4     else:  
5         return n * factorial(n-1)  
6  
7 print(5 * 4 * 3 * 2 * factorial(1))
```

# 処理の流れ

p.10-11の処理を、以下まで繰り返す

```
1 def factorial(n):  
2     if (n==1):  
3         return 1  
4     else:  
5         return n * factorial(n-1)  
6  
7 print(5 * 4 * 3 * 2 * factorial(1))
```



factorial(1) は  $n == 1$  に該当するので、1に置き換わる

```
1 def factorial(n):  
2     if (n==1):  
3         return 1  
4     else:  
5         return n * factorial(n-1)  
6  
7 print(5 * 4 * 3 * 2 * 1)
```

# 終了条件

-  枠のことを、再帰関数の**終了条件**という

再帰呼び出しを終了させるために必要な条件  
(if文を利用して設定されることがほとんど)

再帰関数を正しく動作させるための、  
終了条件の定め方

1. どんな引数を入力しても、必ず終了条件にたどり着けるようにする
2. 終了条件の戻り値には、再帰呼び出しを書かない
3. 想定しない入力があった場合には、エラーの処理を行う (if文でOK)

```
def factorial(n):  
    if (n == 1):  
        return 1  
    return n * factorial(n-1)  
  
print(factorial(5))
```

# 再帰関数のイメージ

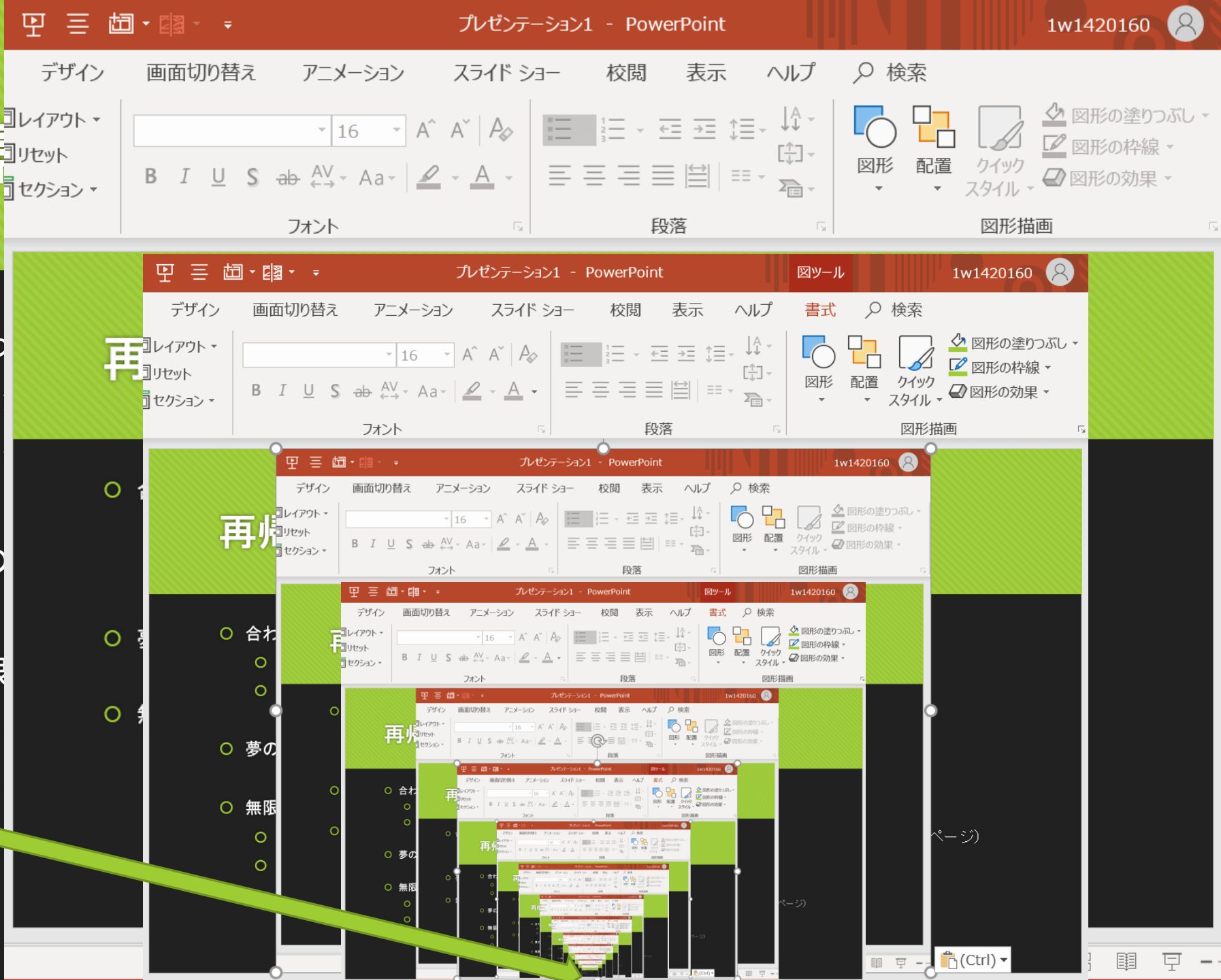
- 合わせ鏡
  - 鏡の中に、鏡自身が移り、その鏡にも鏡が映る（家でできるならやってみてください）
  - 終了条件がない（？）ので終わらない（光が届かなくなる限界が終了条件）
- 夢の中で夢を見ている夢を見る
- 無限スクリーンショット
  - スクリーンショットした画面をスライドに貼り、さらにそれをスクリーンショットする（次のページ）
  - スクリーンショットを撮るのに疲れたらそれまでにとった枚数が終了条件（次のページ）



再帰

- 合わせ
- 
- 
- 夢の
- 無限
- 
- 
- 

終了条件



ページ)

ページ)

ページ)

# フィボナッチ数列

- n番目のフィボナッチ数列を出力する関数をfibonacci(n)とする。

```
1 def fibonacci(n):
2     if (n==1 or n==2):
3         return 1
4
5     return fibonacci(n-1) + fibonacci(n-2)
6
7 fibonacci(6)
8
```

8

## 【定義1】

数列のn番目の数が、n-1番目の数と、n-2番目の数の和となるような数列を、フィボナッチ数列という

# フィボナッチ数列の終了条件

- n番目のフィボナッチ数列を出力する関数をfibonacci(n)とする。

```
1 def fibonacci(n):  
2     if (n==1 or n==2):  
3         return 1  
4  
5     return fibonacci(n-1) + fibonacci(n-2)  
6  
7 fibonacci(6)  
8
```

## 【定義2】

フィボナッチ数列の、1番目、2番目の数は、それぞれ1である

(定義がそのまま終了条件の役割を果たす)  
⇒ 終了条件がわからなくなったら、定義を思い出す  
(数学に近い話の場合は)

# 再帰関数の応用例

- ロボットが進む道の探索（障害物あり）
  - （もっとも最短で目的地にたどり着くためにはどうすればよいか）
- 電車の最短ルート検索
  - グラフ理論と組み合わせて使います
- 数学的な解析
  - 一般項で表しにくい数列の性質を調べる

# 自由課題

1. 再帰関数を利用して、 $n$ の階乗を出力するプログラムを作成してください。  
見ながら写して書いた場合は、すこし経って忘れてから、何もみずにかけるか確認してみてください。
2. 再帰関数を用いずに $n$ の階乗を出力するプログラムを作成してください。
3. 再帰関数を用いて、フィボナッチ数列を30個目まで出力するプログラムを作成してください。
4. `factorial(n)` と、`fibonacci(n)`関数に、本来考えない数字が引数として与えられた場合のエラー処理を書いてください。  
(if文で、何も出力しない、または-1を引数として返す、またはエラーメッセージを表示するようにすればOK)
5. フィボナッチ数列の一般項について調べ、一般項を利用することによってフィボナッチ数列を出力してみてください。

# AtCoderの問題例・解説など

- AtCoder Programming Guide for beginners (APG4b) V- 2.05. 再帰関数
  - [https://atcoder.jp/contests/apg4b/tasks/APG4b\\_v](https://atcoder.jp/contests/apg4b/tasks/APG4b_v)
- [ARC061 C - たくさんの数式 / Many Formulas](#)
- [ABC079 C - Train Ticket](#)
- そのほかは、ほかの技術など(bit全探索、幅優先探索など)を応用するものが多いので、授業で出てきたら提示します。