7. 関数を定義する

Defining functions

プログラミング・データサイエンス I

2023/6/1

1　今日の目的

今日の目的

- 関数を定義する目的
  - 動作確認が容易
  - 繰り返し利用
  - 小さな関数を沢山書くのが良い
- 関数の定義
- 関数を使う

論文などの文章を作成するためには、テーマや目的のはっきりした段落を作り、段落から節を作り、というふうに階層的に全体を組み上げていきます。重要なのは、段落や節のテーマや目的を一つに絞って、明確にすることです。こうすることで、読者にとっても、筆者にとっても、議論の流れが明確になります。また、文章を作成する観点からは、手直しすべき点が明らかになります。

プログラムでも、目的と機能が明確な、小さな部品を組み上げるのが基本です。この部品の一つが関数です。関数を使うことで、処理の流れを明確にするとともに、部品である関数の一つ一つの動作を、独立して確認することが可能となります。また、関数とすることで、プログラムの中で、その機能を繰り返し再利用することが可能になります。

ここまで、Python が提供する関数を幾つか利用してきました。対象の長さを測る `len()` や、対象の型を調べる `type()` などです。今回は、自分で関数を作ることが目標です。関数は function の訳です。function には「機能」という意味もあります。一方、「関数」という名前は、function の音訳であって、必ずしも数に関わるものという意味はありません。

サンプルプログラムをダウンロードしてください。

`https://github.com/first-programming-saga/functions`

2 関数

2.1 **引数の無い関数**: Functions without arguments

引数の無い関数: Functions without arguments

- 関数の中で、何か処理する
  - 必要に応じて結果を返す
- 結果を戻す: return 文
- 呼び出しは、単に関数名を使う
  - 戻り値があれば、代入する

最初に関数の定義方法をまとめておきます。「`def` **関数名** `():`」で開始します。末尾にコロンがありますから、プログラムブロックの開始を表しています。次の行からインデントが始まります。インデントは、関数の終わりまで続きます。つまり、このインデントで表されたプログラムブロックが関数の本体です。このプログラムグロックの中に、関数に対応する処理を書きます。それでは `simpleFunctions.ipynb` を開きましょう。

ソースコード 2.1 では、二つの関数 `hello()` と `goodbye()` を定義しています。いずれの関数も、関数の中からメッセージを印刷しています。数学の関数のように、値を返すことはありません。値を返さない関数を、「副プログラム」や「サブルーチン」と呼ぶことがあります。

7 行目と 8 行目で、それらの関数を呼び出しています。この部分にはインデントがありませんから、関数定義の外側であることに注意してください。

数学関数のように、関数を呼び出したほうに値を返すには `return` 文を使います。ソースコード 2.2 で定義している二つの関数 `hello()` と `goodbye()` は、戻り値としてメッセージを返します。値を返す `return` 文は、一つの関数に複数個あっても構いませんが、`return` が実行されると、関数の残りの部分は実行されないことに注意が必要です。ソースコード 2.2 の呼び出し側を見ると、7 行目と 8 行目で戻ってきたメッセージを保存し、9 行目と 10 行目で、それらを印刷しています。

配布したプログラムでは、少し違う形になっています (ソースコード 2.3)。関数定義の右にある`-> str` は、関数の戻り値が `str` 型であることを明示する部分です。これは、無

ソースコード 2.1　引数のない関数

```
def hello():
    print("こんにちは")

def goodbye():
    print("さようなら")

hello()
goodbye()
```

ソースコード 2.2　引数のない関数

```
def hello2():
    return "こんにちは"

def goodbye2():
    return "さようなら"

m1 = hello2()
m2 = goodbye2()

print(m1)
print(m2)
```

くても構いませんが、記述しておくと関数の振る舞いがソースコードを読む人にとって、分かりやすくなります。今後の例題では、このように戻り値を指定することにします。

ソースコード 2.3　戻り値の型を明示した引数のない関数

```
def hello2() -> str:
    return "こんにちは"

def goodbye2() -> str:
    return "さようなら"
```

2.2 引数の有る関数: Functions with arguments

関数を定義する:引数の有る関数: Functions with arguments

- 引数 (arguments)
  - 関数に渡す変数
  - 全て参照渡し
  - int, float, str などは元の値は変わらない
  - mutable 変数は元の値が変わる
- 結果を戻す: return 文

数学の関数、例えば二次関数では、x の値を与えて二次式に従って値を計算します。x のような役割を、関数の「引数 (argument)」と言います。一般には、複数の引数を関数に与えることができるため、「引数並び」と呼びます。関数の定義に現れる引数を「仮引数」、関数を呼ぶ際に関数に渡す引数を「実引数」と区別して使うこともあります。

関数の引数は、参照、つまり値を保存している場所の情報を表します。ただし、int、float、str などのようの変更できない (immutable) 変数は、関数の中で値を変更しても、呼び出し側の変数は変化しません。このような immutable 変数を関数に渡すことを「値渡し」と言います。それ以外は、「参照渡し」になります。リストなどのデータの塊は「参照渡し」です。argumentsTest.ipynb を見ながら説明しましょう。

ソースコード 2.4 の関数 func1() は、引数として受け取った変数に 1 を加えます。引数の部分 x:float は、引数が float 型であることを明示しています。単に x だけでも問題はありません。浮動小数点型は immutable ですから、関数への引数渡しは「値渡し」です。そのため、関数呼び出しに使用した変数 y の値は変化しません。

関数定義の後の"""で括った部分は、関数の説明です。VSCode 内で、関数定義や関数利用の部分、今の場合には func1 にマウスを合わせると、"""に記載した説明を見ることができます。

一方、ソースコード 2.5 の関数 func2() は、引数として受け取るのはリストです。そのリストの各要素を二乗し、元の位置に書き戻します。この場合の引数渡しは、「参照渡し」になっています。従って、関数の戻り値はありません。この場合は、呼び出し側のリスト data そのものを変更していることに注意してください。また、引数の型の表示:list[float] は、float 型を要素とするリストであることを表しています。

ソースコード 2.6 の関数 func4() も、引数として受け取るのはリストです。しかし、

ソースコード 2.4　int 型の引数

```
def func1(x:float) -> float:
    """
    関数内部で引数に 1 を加えて返す
    """
    x += 1
    return x

y = 1
z = func1(y)
print(f'y={y}, z={z}')#呼び出し元の値は変更されない
```

ソースコード 2.5　list 型の引数

```
def func2(d:list[float]) -> None:
    """
    リストの各要素を 2 倍にする
    """
    for i in range(len(d)):
        x = d[i] * 2
        d[i] = x

data = [1, 4, 2, 5]
func2(data)
print(data)
```

最初に引数のリストを別の変数にコピーをして、そのコピーを変更し、変更したリストを返します。そのため、呼び出し側の変数 data を変更することはありません。

課題 2.1　argumentsTest.ipynb の最後にある課題です。$0 \leq x < 1$ の float 型の引数を受け取り、$0 \leq x < 0.5$ ならば -1 を、$0.5 \leq x < 1$ ならば $+1$ を、それ以外ならば 0 を返す関数 func5() を定義し、動作を確かめなさい。

ソースコード 2.6　list 型の引数を変更しない方法

```
def func4(dd:list[float]) -> list[float]:
    """
    リストの各要素を 2 倍して、別のリストを返す
    """
    d = list(dd)
    for i in range(len(d)):
        x = d[i] * 2
        d[i] = x
    return d

data = [1, 4, 2, 5]
data2 = func4(data)
print(data)
print(data2)
```

3　再帰的関数: Recursive functions

再帰的関数: Recursive functions

- 関数を、関数自身で定義する
- 値を確定させる場所に注意

再帰的関数とは、関数の定義の中に、その関数自体を含むものです。例として n の階乗 $n!$ を考えましょう。n の階乗は、1 から n までの積です。

$$n! = \prod_{k=1}^{n} k = n \times (n-1) \times \cdots 2 \times 1 \tag{3.1}$$

$\prod$ は、インデクスを変化させながら掛け算をする記号です。積を表す product を、ギリシャ文字の大文字 P (Π) で表したものです。例えば 5! は

$$5! = \prod_{k=1}^{5} k = 1 \times 2 \times 3 \times 4 \times 5$$

です。

一方、再帰的な観点で見ると、$n!$ の値は、$(n-1)!$ が分かれば、それに n を乗じて得る

ことができます。

$$n! = n \times (n-1)! \tag{3.2}$$
$$0! = 1 \tag{3.3}$$

$n!$ を $(n-1)!$ という同じ関数の引数が異なるものを使って定義しています。つまり、定義が再帰的 (recursive) です。$(n-1)!$ の値を、さらに $(n-2)!$ を使って定義します。このように、一つ一つ値を下げていくのですが、無限に下げてはいけません。そこで $0! = 1$ としておきます。

`factorial.ipynb` を開けてみましょう。ソースコード 3.1 の 2 行目が、$n = 1$ の場合に値を確定している部分です。$n > 1$ の場合には、4 行目で、$n - 1$ の場合を求めるために、再帰的に関数 `factorial()` を呼び出します。

ソースコード 3.1　階乗

```
1  def factorial(n:int) -> int:
2      """
3      階乗
4      """
5      if n < 0:
6          raise ValueError('argument must not be negative')
7      if n == 0:
8          return 1
9      return n * factorial(n - 1)
```

再帰的に手続きを記述することで、単純化することができます。一方、確実に停止して値が確定するように、注意が必要です。階乗の場合には、呼び出しの度に n の値が一つ小さくなります。5 行目で $n = 0$ を確認することで、無限に小さくなることを防いでいます。$n < 0$ に対しては、例外を発生させています。

4　関数の戻り値としてタプルを使う: Tuple as a return value

関数の戻り値としてタプルを使う: Tuple as a return value

- 関数から複数の戻り値を返したい
- タプルを使うと、簡単に返すことができる

前回、タプル (tuple) というデータ構造を導入しました。複数のオブジェクトを括弧で

括ってまとめたものです。このタプルというデータ構造が最も役立つのは、関数から複数の戻り値を返したいときです。

ソースコード 4.1　素数を判定する関数

```
1  def isPrime(n:int) -> tuple[bool,str]:
2      """
3      引数が素数か否かを判定し、結果を理由とともに返す
4      """
5      result = False
6      if n <= 0:
7          message=f'引数は正でなければならない'
8      elif n < 2:
9          message = f'{n}は素数ではない'
10     elif n == 2:
11         message = f'{n}は素数である'
12         result = True
13     elif n % 2 == 0:
14         message = f'{n}は偶数であり、素数ではない'
15     else:
16         m = int(math.sqrt(n))
17         for k in range(3, m + 1, 2):#for で記述
18             if n % k == 0:
19                 message = f'{n}は{k}で割り切れるため、素数ではない'
20                 break
21         else:#ループの最後まで至った場合
22             message = f'{n}は素数である'
23             result = True
24     return result, message
```

isPrime.ipynb を開いてください。ソースコード 4.1 内で定義している関数 isPrime() は、引数 n が素数か否かを判定します。判定が目的ですから、True または False を返せばよいのですが、特に False の場合に、その理由も返したいですね。そこで、isPrime() では、True または False の値をとる判定結果 result と、その理由を示す message の二つを 21 行目でタプルとして返します。単に、2 つのオブジェクトを return 文のところで、カンマ区切りで記述しています。タプルとしてわざわざ宣言することもしていませんし、括弧も省略している点に注意してください。

ソースコード 4.2 は、関数 isPrime() の呼び出し側です。呼び出し側でも、戻り値のタプルを括弧を使わずに受け取っています。r には True/False の結果が、m にはその理由の文字列が返ってきます。

もう一つ例を見ましょう。stat0.ipynb を見ましょう (ソースコード 4.3)。関数

ソースコード 4.2　素数を判定する関数の呼び出し

```
for i in range(3, 100, 2):
    r, m = isPrime(i)
    print(m)
```

stat() は、データの入ったリストを受け取り、データの個数 n と平均 average をタプルとして返します。

ソースコード 4.3　リスト中のデータの平均とデータ数を返す関数

```
def stat(data:list[float]) -> tuple[int,float]:
    """
    引数で渡されたリスト中のデータに対して、データ数、平均を返す関数
    """
    n = len(data)
    s = 0 #和を保存
    for x in data: #data 中のすべてに対して繰り返し
        s += x
    average = s / n #平均
    return n, average
```

5　関数とスコープ: Functions and scopes

関数とスコープ: Functions and scopes

- 変数名には、有効範囲 (scope) がある
- 関数内だけで有効な変数
- 関数の外側でも有効な変数

変数には、スコープ (scope) という、その有効範囲があります。scopeTest.ipynb の例を見ながらスコープの概念を理解しましょう。ソースコード 5.1 では、全てのセルを連結していることに注意してください。

変数スコープの例を見ましょう。

- 最初にグローバル変数 aGlobal を定義しています。グローバル変数とは、関数の外側にあり、プログラム全体で有効な変数です。

- 関数 func1() の中で、変数 x にグローバル変数 aGlobal を代入しています。この変数 x は、関数の中だけで有効です。
- 関数 func2() の内容は、少し混乱します。関数の内部で左辺に aGlobal が出てきます。しかし、これは関数内のローカル変数です。16 行目で関数 func2() を呼び出した後、17 行目で aGlobal を印刷してみると、値は変化していません。
- 関数 func3() 内でローカル変数 y に値を入れています。y は、関数 func3() 内部だけで有効な変数ですから、18 行目で y の値を印刷しようとすると、y という変数は定義されていない、というエラーが発生します。

ソースコード 5.1　変数のスコープを理解する

```
1  #グローバル変数
2  aGlobal = 100
3
4  def func1():
5      x = aGlobal
6      return x
7
8  def func2():
9      aGlobal = 2 #この aGlobal はこの関数内のローカル変数
10
11 def func3():
12     y = 1
13
14 z = func1()
15 print(z)
16 func2()
17 print(aGlobal)
18 print(y)
```

6　引数名の明示と省略: Explicit assignment and default of arguments

引数名の明示と省略

- 引数名を明示して値を指定する
- 省略できる引数

以降の回では、エクセルのデータを作図するなどの例を扱います。作図のための関数などは、図の設定に関する非常に沢山の引数を持っている場合があります。全ての引数を指定することは大変です。また、多くの引数は既定値 (default) のままで十分です。そこで、Python では、一部の値だけを引数名で指定することと、既定値がある場合には、その引数を指定しないことができます。

argumentTest1.ipynb として配布しているソースコード 6.1 を見てください。関数 setRGB は、赤、緑、青の三原色の値を 0 から 255 までの値で指定すると、対応する 16 進表現の文字列を返す関数です。[*1]値を指定しないと、0 になるように既定値を定めています。7 行目から 9 行目が、その関数の利用例です。引数名で指定した値以外は、0 のままで計算しています。既定値を指定すると、その型を指定したことと同じですから、:int のように型を指定することはできません。

ソースコード 6.1　引数の指定と省略

```
1  def setRGB(red = 0, green = 0, blue=0) -> str:
2      """
3      convert rgb decimal expression to hex expression
4
5      Parameters
6      ---
7      red [0,255]
8      green [0,255]
9      blue [0,255]
10
11     Returns
12     ---
13     str such as 0x******
14     """
15     redStr = format(red, '02x')
16     greenStr = format(green, '02x')
17     blueStr = format(blue, '02x')
18     str = "0x" + redStr + greenStr + blueStr
19     return str
```

format() という関数は、最初の引数の値を二番目の引数で指定した書式で表現するものです。02x というのは、二桁の 16 進表現のことです。値が一桁の 16 進数の場合には、

[*1] 16 進数では、0 から 9 まで、更に A から F までの 16 通りの記号で数を表します。先頭に 0x を付けて区別します。桁上がり規則は 0xF + 0x1 = 0x10 です。16 進数は、文字コード、色の表現などで利用されています。

最初に 0 を補完します。

7　課題

quiz.ipynb を開き、二次方程式の解を求める関数 quadratic() を作成し、その動作を確認しなさい。

8　次回

教科書では、この後はクラスの利用、ファイルの利用となっています。講義では、エクセルファイルに集中して進めていきます。