第2回 Pythonの基礎 後編

第1回ではPythonの基礎として、エディタや実行環境、データ型、および演算子について学びました。第2回では、後編として次の項目について学びます。

- 制御構文
- 文字列の操作
- 予約語
- 関数とメソッド
- 例外
- 注意すべき用語

代表的な制御構文

ここまで見てきたソースコードは、複数の命令を記述されたとおりに上から下へ、順に実行していくだけのものでした。

Pythonを含むプログラム言語では、制御構文というものを用いて、条件により実行する処理を変化させたり、まとまった処理のかたまりを複数回繰り返し実行することができます。

ここでは代表的な制御構文として、「if文」と「for文」について紹介します。

条件により処理を変更する(if文)

if文の基本

「**if文**」を使用すると、条件によってプログラムの処理を変更することができます。if文の基本的な構文は、次の通りです。

```
if 条件式:
    条件式を満たしたときに実行する命令1
    条件式を満たしたときに実行する命令2
                :
```

- 条件式は、「比較演算子(x == y、x > yなど)」や「ブール演算子(and、orなど)」などを用いて記述します。
- 条件式の後ろの:(コロン)を忘れないようにしましょう。

次のソースコードは、if文を用いて実行する命令を変化させる例です。

In [ ]:

```
x = 10
y = 10

# xとyが等しいとき
if x == y:
    print("xとyは等しいです")

# xがyより大きいとき
if x > y:
    print("xはyより大きいです")
```

- ifの後ろに記述した「x == y」「x > y」が、比較演算子による条件式です。これがTrueとなった場合だけ、後続のprint関数が実行されます。
- 一つ目のif文ではprint関数が実行されますが、二つ目のif文ではprint関数が実行されません。

インデントについて

ところで、次のソースコードを見てみてください。条件式「x == y」はFalseであり条件を満たしませんが、出力結果はどうなるでしょうか。

In [ ]:

```
x = 10
y = 100

# 条件式を満たさない例
if x == y:
    print("1つ目のprint関数")
    print("2つ目のprint関数")
    print("3つ目のprint関数")
print("4つ目のprint関数")
print("5つ目のprint関数")
print("6つ目のprint関数")
```

条件式を満たさないので、if文以降のprint関数は実行されませんが、4つ目のprint関数以降は実行されました。つまり、次のことが分かります。

- 1つ目～3つ目のprint関数は、if文が有効である。
- 4つ目～6つ目のprint関数は、if文が有効でない。

この有効範囲は、どのように決定したのでしょうか。答えは「行頭に1つ以上の連続した半角空白があるかどうか」です。このようにPythonでは、制御文の有効範囲を行頭の半角空白で決定します。

行頭に1つ以上の連続した半角空白を入れることを、「**インデント**」または「字下げ」と呼びます。本講座では、「インデント」の方に表現を統一します。

半角空白の個数は任意ですが、4個とすることが多いです。本講座の資料でも4個としていますが、Google Colaboratoryのデフォルト値は2個となっていますので、お好みの設定に変更してください。

また逆に、制御文の有効範囲以外の場所で、行頭に半角空白を入れるとプログラムはエラーとなります。次のプログラムを実行してみましょう。

In [ ]:

```
    print("1つ目のprint関数")

if x == y:
    print("2つ目のprint関数")
```

【小ネタ】Python以外のプログラミング言語では、例えば次のような方法で、制御文の有効範囲を決定します。これらの方法では、行頭の半角空白は単に見やすさのために入れているだけであり、必須ではありません。

- {}などの括弧で囲まれた範囲を有効範囲とする。(C、Javaなど)

```
if（条件式）{
    条件式を満たしたときに実行する命令1
    条件式を満たしたときに実行する命令2
}
```

- 「then」「end」というキーワードで囲まれた範囲を有効範囲とする。(Rubyなど)

```
if 条件式 then
    条件式を満たしたときに実行する命令1
    条件式を満たしたときに実行する命令2
end
```

else

ifに加えてelseを使用すると、条件式を満たしたときだけではなく、条件式を満たさなかったときだけ実行する命令を記述することができます。if-elseの基本的な構文は、次の通りです。

```
if 条件式:
    条件式を満たしたときに実行する命令1
    条件式を満たしたときに実行する命令2
                :
else:
    条件式を満たさなかったときに実行する命令1
    条件式を満たさなかったときに実行する命令2
                :
```

- 条件式を満たさなかったときに実行する命令も、インデントを使って有効範囲を定めます。
- elseの後ろの:(コロン)を忘れないようにしましょう。

次のソースコードは、if-elseを用いて実行する命令を変化させる例です。

In [ ]:

```
x = 10
y = 20

if x == y:
    print("xとyは等しいです")
else:
    print("xとyは等しくないです")
```

if-elseに加えて、さらにelifを使用すると、条件1を満たしたときだけ処理1を実行、条件2を満たしたときだけ

処理2を実行・・・といった記述ができます。if-elif-elseの基本的な構文は、次の通りです。

```
if 条件式1:
    条件式1を満たしたときに実行する命令1
    条件式1を満たしたときに実行する命令2
                :
elif 条件式2:
    条件式2を満たしたときに実行する命令1
    条件式2を満たしたときに実行する命令2
                :
elif 条件式3:
    条件式3を満たしたときに実行する命令1
    条件式3を満たしたときに実行する命令2
                :
else:
    ifおよびelifの条件式を満たさなかったときに実行する命令1
    ifおよびelifの条件式を満たさなかったときに実行する命令2
                :
```

- 上記ではelifを2つ記述していますが、何個でも記述することができます。
- elifの条件式を満たしたときに実行する命令も、インデントを使って有効範囲を定めます。

次のソースコードは、if-elif-elseを用いて実行する命令を変化させる例です。

In [ ]:

```
x = 20
y = 10

if x == y:
    print("xとyは等しいです")
elif x > y:
    print("xはyより大きいです")
else:
    print("xはyより小さいです")
```

処理を繰り返す(for、range関数、リスト内包表記)

for文の基本的な使い方

「**for文**」を使用すると、まとまった処理のかたまりを、複数回繰り返し実行することができます。

for文の基本的な構文は、次の通りです。

```
for 変数 in 繰り返し要素:
    繰り返す処理1
    繰り返す処理2
        :
```

- 「繰り返し要素」は、リストやタプル、後述する数列など、繰り返し処理したい複数の要素を持つデータです。
- 「繰り返し要素」から順番に値が取り出され、「変数」に格納されます。この「変数」は、繰り返し処理の中で使うことができます。
- 「繰り返し要素」の値をすべて取り出すまで、処理を繰り返し実行します。

- 「繰り返し要素」の後ろの、:(コロン)を忘れないようにしましょう。

次のソースコードは、繰り返し要素として整数のリストを使っています。

文字列に対する「+」演算子、「str関数」なども使っていますので、復習しておきましょう。

In [ ]:

```
num_list = [1, 2, 3]

# 繰り返し要素(num_list)の内容を一つずつ取り出して処理を繰り返す
for num in num_list:
    print("num=" + str(num) + ", 命令1")
    print("num=" + str(num) + ", 命令2")
```

リストの値が順番に取り出され、変数numに格納されます。

リストの値をすべて取り出すまで、2つのprint関数を繰り返し実行します。

if文と同様に、for文も有効範囲をインデントで定めます。次のソースコードを実行してみましょう。

In [ ]:

```
num_list = [1, 2, 3]

# 繰り返し要素(num_list)の内容を一つずつ取り出して処理を繰り返す
for num in num_list:
    print("num=" + str(num) + ", 命令1")
    print("num=" + str(num) + ", 命令2")
print("命令 X")
print("命令 Y")
```

最後の2行、"命令 X"と"命令 Y"を出力するprint関数は、インデントされていませんのでfor文の有効範囲外です。

for文の繰り返しがすべて終わった後で、一度ずつだけ実行されています。

range関数

for文とともによく使われる関数として、数列を生成する「**range関数**」があります。

- range(N)と記述すると、0～(N-1)の数列を作成します。
- range(N, M)と記述すると、N～(M-1)の数列を作成します。このとき、N < Mとなるようにします。
- range(N, M, D)と記述すると、次のとおりとなります。
  - N < M かつ D > 0 のとき、N～(M-1)の範囲でDずつ増えていく数列を作成します。
  - N > M かつ D < 0 のとき、N～(M+1)の範囲でDずつ減っていく数列を作成します。

次のソースコードは、繰り返し要素としてrange関数で作成した数列を記述しています。繰り返し処理では、変数numの値を出力します。

In [ ]:

```
for num in range(6):
    print(num)

print("--------------------------------")

for num in range(5, 10):
    print(num)

print("--------------------------------")

for num in range(5, 10, 2):
    print(num)

print("--------------------------------")

for num in range(9, 4, -2):
    print(num)
```

リスト内包表記

for文の応用的な使い方として、「**リスト内包表記**」があります。リスト内包表記は、特定のルールに従った新しいリストを作成するのに役立ちます。

リスト内包表記は、[式 for 変数 in 繰り返し要素] と記述します。

次のソースコードは、繰り返し要素としてrange関数で作成した数列を記述しています。式には、変数の2乗を記述しています。

In [ ]:

```
# 0～9それぞれを2乗した値のリストを作成します
for_list = [ num ** 2 for num in range(10) ]

# リストの内容を画面に出力します
print(for_list)
```

- range関数で作成した数列の値が順番に取り出され、変数numに格納されます。
- ここでは0～9の数列が作成され、それぞれを2乗した値を要素として持つリストが作成されます。

その他の繰り返し制御文

処理を繰り返すもう一つの制御構文として、while文というものがあります。

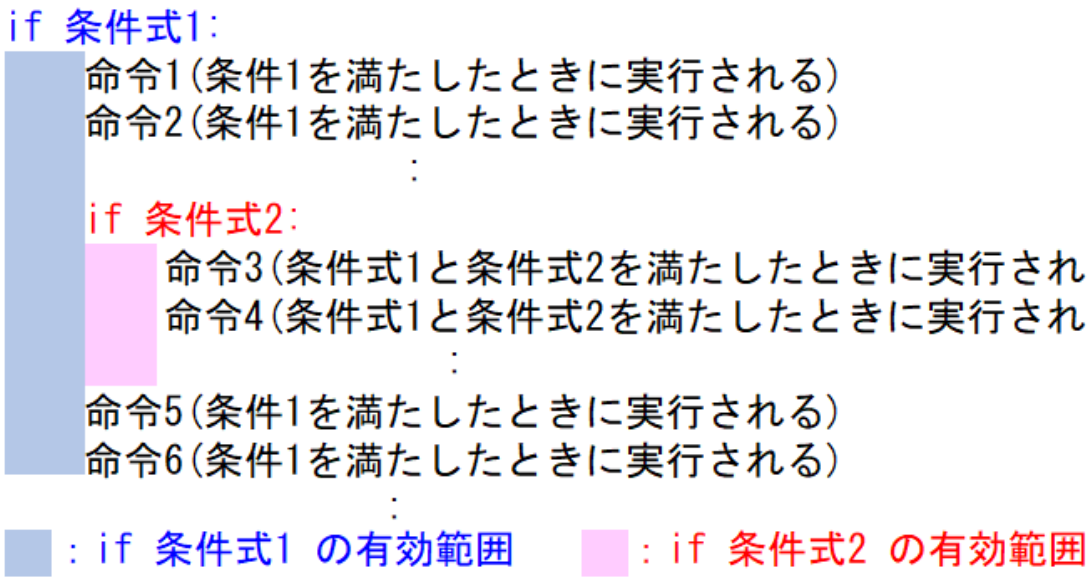
しかし、while文でできることはfor文で大概のことはできますので、あまり使われません。

本講座では取り上げませんが、興味があれば各自で調べてトライしてみてください。

制御文の入れ子記述

制御文は、if文の中にif文を記述したり、for文の中にif文を記述するなど、入れ子で記述することができます。

このとき、インデントの半角空白の個数に注意が必要です。例として、if文の中にif文を記述する場合は、次のように記述します。



- 2つ目のif文であるif 条件式2:は、1つ目のif文であるif 条件式1を満たしたときだけ実行されるので、if 条件式1:よりも半角空白4つぶんインデントされています。
- 命令3および命令4は、2つ目のif文であるif 条件式2:を満たしたときだけ実行されるので、if 条件式2:よりも半角空白4つぶんインデントされています。
- 結果として、命令3および命令4は行頭から半角空白8つぶんインデントされています。これは、命令3および命令4がif 条件式1とif 条件式2の両方を満たしたときだけ実行されることを意味します。

このように、入れ子の深さの分だけインデントする必要があります。if文の中にfor文を記述するなど、ほかの組み合わせでも同じようにインデントで制御文の有効範囲を定めます。

break文とcontinue文

ここで、break文とcontinue文について学びます。

break文とcontinue文はfor文(＋while文)の機能ですが、for文とif文の入れ子構造で頻繁に使用されるため、あえてここで紹介します。

break

for文で繰り返す命令の中に「**break文**」を記述すると、次の通り動作します。

- 現在の繰り返し処理を中断する。
- 次の繰り返し処理に進まず、for文を終了する。

次のソースコードを見てみましょう。

In [ ]:

```
num_list = [1, 2, 3]

# 繰り返し要素(num_list)の内容を一つずつ取り出して処理を繰り返す
for num in num_list:
    print("num=" + str(num) + ", 命令1")
    if num == 2:
        break

    print("num=" + str(num) + ", 命令2")
```

for文の中にif文を記述する、入れ子構造になっていますね。print関数やif文は半角空白4つぶんのインデント、breakは半角空白8つぶんのインデントとなっています。

変数numには整数1、2、3と順番に格納されていきますが、numが2になったとき、条件式を満たすためbreak文が実行されます。次の点に注目してください。

- 「"num=2, 命令2"」が出力されない。(命令を中断する)
- num = 3 の繰り返し処理は行われない。(for文による繰り返し処理を強制的に終了)

continue文

for文で繰り返す命令群の中に「**continue文**」を記述すると、次の通り動作します。

- 現在の繰り返し処理を中断する。
- 次の繰り返し処理に進み、for文を継続する。

次のソースコードは、前のソースコードのbreak文がcontinue文に変わっただけです。実際に動かしてみましょう。

In [ ]:

```
num_list = [1, 2, 3]

# 繰り返し要素(num_list)の内容を一つずつ取り出して処理を繰り返す
for num in num_list:
    print("命令 " + str(num) + "-1")

    if num == 2:
        continue

    print("命令 " + str(num) + "-2")
```

今回も変数numには整数1、2、3と順番に格納されていきますが、numが2になったとき、条件式を満たすためcontinue文が実行されます。次の点に注目してください。

- 「"num=2, 命令2"」が出力されない。(命令を中断する)
- num = 3 の繰り返し処理が実行される。(次の繰り返し処理に進む)

2点めの「num = 3 の繰り返し処理が実行される」という点が、break文とcontinue文の違いになります。

文字列の操作

"Hello, world."などの文字列について、ここまでprint関数による画面出力などの例を見てきました。

ここでは、文字列の加工や検索など、データ構造化でよく使われる様々な操作方法を紹介します。

文字列の分割(split)

「**split**」という命令を使用すると、文字列を任意の区切り文字で分割することができます。

- 文字列.split(区切り文字列) と記述すると、文字列を区切り文字列で分割したリストを取得できます。
- 区切り文字列を指定しない場合、連続した半角空白・タブ・改行で分割します。

次のソースコードは、カンマおよび半角空白で文字列を分割しています。

In [ ]:

```
# カンマ区切りの文字列を分割
comma_str = "aaa,bbb,ccc"
print(comma_str.split(","))

print("----------------------------------")

# 空白区切りの文字列を分割
space_str = "xxx yyy   zzz"
print(space_str.split())
```

文字列の先頭および末尾の文字を削除(strip、lstrip、rstrip)

「**strip**」「**lstrip**」「**rstrip**」という命令を利用すると、文字列の先頭および末尾の不要なデータを削除できます。

- 文字列.strip(削除文字列) と記述すると、**文字列の先頭および末尾の両方から**、連続した削除文字列をすべて削除します。削除文字列を指定しないと、連続した半角空白・タブ・改行を削除します。
- 文字列.lstrip(削除文字列) と記述すると、**文字列の先頭からだけ**、連続した削除文字列をすべて削除します。削除文字列を指定しないと、連続した半角空白・タブ・改行を削除します。
- 文字列.rstrip(削除文字列) と記述すると、**文字列の末尾からだけ**、連続した削除文字列をすべて削除します。削除文字列を指定しないと、連続した半角空白・タブ・改行を削除します。

次のソースコードは、上記のstrip、lstrip、およびrstripそれぞれについて、先頭および末尾の不要なデータを削除しています。

In [ ]:

```
abc_str = "abcxyzabcabc"

# 先頭および末尾の文字列を削除
print(abc_str.strip("abc"))

# 先頭の文字列を削除
print(abc_str.lstrip("abc"))

# 末尾の文字列を削除
print(abc_str.rstrip("abc"))

print("--------------------------------")

space_str = "   xyz   "
# 先頭および末尾の文字列を削除
print(space_str.strip() + "END")

# 先頭の文字列を削除
print(space_str.lstrip() + "END")

# 末尾の文字列を削除
print(space_str.rstrip() + "END")
```

【ワンポイント】「split」と「strip」は、どちらも文字列に対する操作で、かつ名称が似ているため、非常に間違いやすいです。

データ構造化のプログラムでは、「split」と「strip」はメタデータの抽出において必須となる操作です。csvの数値列、文字列から必要となるメタデータや値を得るためにsplit(分割)が多用され、不要な空白や改行をなくすためにstrip(削除)が多用されます。

文字列の判定(in、not in、startswith、endswith)

比較演算子「**in**」を使用すると、if文などと組み合わせて、文字列全体の中に特定の文字列が含まれるかどうか判定できます。

次のソースコードは、文字列中に特定の部分文字列が含まれるかどうかを判定しています。

In [ ]:

```
in_str = "abcdefg"

# inの結果がTrueとなるケース
if "def" in in_str:
    print("defが含まれています。(True)")
else:
    print("defが含まれていません。(False)")

print("--------------------------------")

# inの結果がFalseとなるケース
if "xyz" in in_str:
    print("xyzが含まれています。(True)")
else:
    print("xyzが含まれていません。(False)")
```

inとは逆に、「**not in**」を使用すると、文字列全体の中に特定の文字列が含まれないかどうか判定できます。

次のソースコードは、文字列中に特定の部分文字列が含まれないかどうかを判定しています。

In [ ]:

```
in_str = "abcdefg"

# not inの結果がFalseとなるケース
if "def" not in in_str:
    print("defが含まれていません。(True)")
else:
    print("defが含まれています。(False)")

print("------------------------------")

# not inの結果がTrueとなるケース
if "xyz" not in in_str:
    print("xyzが含まれていません。(True)")
else:
    print("xyzが含まれています。(False)")
```

「**startswith**」を使用すると、文字列全体が特定の文字列で開始するかどうか判定できます。startswithは、「文字列.startswith(部分文字列)」という構文で記述します。start's'withの's'を忘れがちなので、気を付けてください。

次のソースコードは、文字列が特定の部分文字列で開始するかどうかを判定しています。

In [ ]:

```
startend_str = "abcdef"

# startwithの結果がTrueとなるケース
if startend_str.startswith("abc"):
    print("abcで開始します。(True)")
else:
    print("abcで開始しません。(False)")

print("--------------------------------")

# startwithの結果がFalseとなるケース
if startend_str.startswith("xyz"):
    print("xyzで開始します。(True)")
else:
    print("xyzで開始しません。(False)")
```

startswithとは逆に、「**endswith**」を使用すると、文字列全体が特定の文字列で終了するかどうか判定できます。endswithは、「文字列.endswith(部分文字列)」という構文で記述します。end's'withの's'を忘れがちなので、気を付けてください。

使い方は、startswithとほとんど同じです。こちらはソースコード例を載せませんので、各自でチャレンジしてみてください。

【ワンポイント】これらの判定は、特定のメタデータの探索に多用されます。特に in 演算子は、英文的にもわかりやすい演算子です。

文字列の検索(find、count)

「**find**」を使用すると、文字列全体で特定の文字列が何文字目に存在するかを取得できます。このとき、最初の文字は0文字目であることに注意が必要です。

次のソースコードでは、特定の部分文字列が文字列全体の何文字目に存在するかを取得しています。

In [ ]:

```
abc_str = "abcdefg"

# "def"が何文字目に存在するかを取得する
# 0文字目から開始する点に注意
print(abc_str.find("def"))
```

「**count**」を使用すると、文字列全体で特定の文字列が何回出現するかを取得できます。

次のソースコードでは、特定の部分文字列が文字列全体で何回出現するかを取得しています。

In [ ]:

```
abc_str = "abcdefgabcdefgabc"

# "abc"が何回出現するかを取得する
print(abc_str.count("abc"))
```

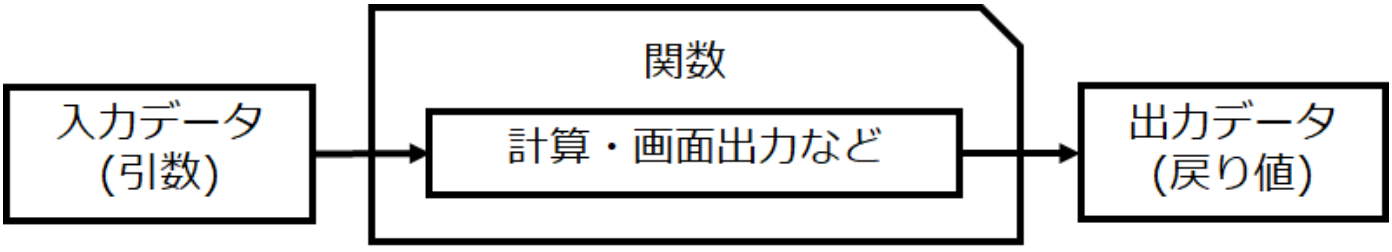
関数とメソッド

関数についての詳細

ここまでに、print関数、len関数、type関数などの「**関数**」を紹介しました。このように、Pythonではよく使用する便利な機能を「関数」という形で提供しています。

関数はデータを入力として受け取り、受け取った入力データに対して計算や画面出力などの機能を実行して、結果のデータを出力します。

入力するデータを「**引数**」、出力する結果のデータを「**戻り値**」と言います。



数学の関数と基本的な考え方は同じです。$y = f(x)$ の x が引数に、y が戻り値に、それぞれ相当するイメージとなります。

「文字列を出力する」のところで、「print("Hello, world.")」という命令を実行しました。

- 文字列"Hello, world."が、入力データである「引数」です。
- print関数には、出力データである「戻り値」はありません(厳密には、値がないことを示す「None」という戻り値を出力しています)。

- 画面に出力した文字列は、あくまでprint関数の機能として画面表示を行っているだけであり、関数の「戻り値」ではない点に注意してください。

「文字列の長さを求める」のところで、「len("Hello, world.")」という命令を実行しました。

- 文字列"Hello, world."が、入力データである「引数」です。
- 文字列"Hello, world."の長さである13が、出力データである戻り値です。

ここで、第1回の「文字列の長さを求める」でご紹介した、次のソースコードを思い出してください。

In [ ]:

```
str_len  = len("Hello, world.")
print(str_len)
```

len関数の戻り値(文字列の長さ)を変数に格納して、その変数の値をprint関数で出力するプログラムでした。

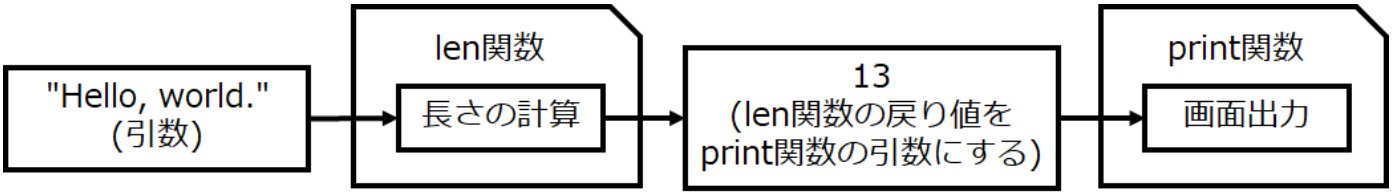
それでは次に、次のソースコードを見てみましょう。

In [ ]:

```
print(len("Hello, world."))
```

このソースコードも、「文字列の長さを求める」のソースコードと同じ結果となります。

違いとして、len関数の戻り値を変数に格納せず、そのままprint関数の引数としています。



このようにprint関数の引数には、文字列や数値などの値、および変数に加えて、関数を直接記述することで、その戻り値を渡すこともできます。

メソッド

今回「文字列の操作」の中で、「文字列.split(区切り文字)」や「文字列.strip(削除文字列)」といった命令をご紹介しました。

これらも関数の一種で、区切り文字が「引数」、操作した結果得られた文字列やリストが「戻り値」となります。

ただし、このように「対象のデータ」(この例では「文字列」)に.(ピリオド)を付けて呼び出す関数のことを、特別に「**メソッド**」と呼びます。

    対象のデータ.メソッド名(引数)

ただし、.(ピリオド)を付けて呼び出したものはすべてメソッド、ということではないので、注意が必要です。

今回の講義の最後で「モジュール」というものについて簡単にご説明しますが、次のようなパターンも存在します。このパターンは、メソッドではなく関数です。

    モジュール名.関数名(引数)

メソッドについて詳細を理解するには、今回はご説明しませんが、「クラス」という機能を学ぶ必要があります。

かなり分かりにくい話とは思いますが、関数とメソッドのいずれも、入力として引数を渡し、出力として戻り値を得る、という基本的な仕組みは変わりません。今のところは、関数の一種としてメソッドと呼ばれるものがある、ということだけ覚えておいてください。

関数を自作する

print関数やlen関数のような、Pythonであらかじめ用意された関数は「**組み込み関数**」と呼ばれます。

それ以外にもPythonでは、自分で関数を作成することもできます。関数を自作するには、「**def**」というキーワードを使用します。

```
def 自作関数名(引数):
    自作関数で行う処理
```

自作した関数を使うときは、基本的にはprint関数やlen関数と同じように、「関数名(引数)」と記述するだけです。

(「基本的には」としたのは、メソッドのところでご説明した「モジュール名.関数(引数)」というパターンもあるためです。)

実際に自作関数を作って、呼び出してみます。次のソースコードで、結果を確認してみましょう。

In [ ]:

```
def display_add10(num):
    print(num + 10)

# 自作関数add10を呼び出す
display_add10(5)
```

引数は、複数個とすることができます。その場合、関数定義、関数呼び出し、どちらも引数を,(カンマ)で区切って複数個指定します。次のソースコードで、確認してみましょう。

In [ ]:

```
def display_add3nums(num1, num2, num3):
    print(num1 + num2 + num3)

# 自作関数display_add3numsを呼び出す
display_add3nums(5, 10, 15)
```

戻り値を返す関数を自作する

これまで自作した関数は、数値を画面に出力するだけで、戻り値はありませんでした。

今度は、display_add3nums関数を改良して、合計した結果を画面に出力するのではなく、戻り値として返すようにしてみます。

戻り値を返すには、関数内で「**return文**」を記述します。

```
def 自作関数名(引数):
    自作関数で行う処理
        :
    return 戻り値
```

次のソースコードで、確認してみましょう。

In [ ]:

```
def return_add3nums(num1, num2, num3):
    return num1 + num2 + num3

# 自作関数return_add3numsを呼び出す
ret_val = return_add3nums(5, 10, 15)

print(ret_val + 10)
```

関数で取得した戻り値に、さらに10を加算して画面に出力しました。

このように、関数の結果を使って後続の処理を行う場合は、結果を戻り値として返し、後続の処理に引き継ぎます。

関数を使うコツ

ソースコードを記述するときは、プログラムの全てをダラダラと上から下へ書いていくのではなく、機能ごとに関数を作成するのがコツです。

特に、何度も実行するようなお決まりの処理は、関数にすることでソースコードが分かりやすくなります。

第6回では、皆様に関数を自作してもらいます。ぜひ、上記を意識して取り組んでもらえればと思います。

【注意事項】

次の2つの自作関数を実行してみて、結果を比べて見てください。どうなるでしょうか。

In [ ]:

```
def display_value(num):
    print(num)

# 自作関数display_valueを呼び出す
display_value(10)
```

In [ ]:

```
def return_value(num):
    return num

# 自作関数return_valueを呼び出す
return_value(10)
```

どうでしょうか。どちらも、引数の値が画面に出力されました。一見、同じ結果に見えます。

ここで、第1回で【注意事項 (とても重要！)】としてご説明した内容を、思い出してください。

Google Colaboratoryでデータの型を取得するtype関数を実行すると、取得したデータ型が画面に出力されました。しかしこれは、Google ColaboratoryやJupyter Notebookの独自機能であり、本来は画面への出力にはprint関数を使う、とご説明しました。

今回の例も、まさにこの注意事項どおりの動作となっています。

2番目のプログラムのように、関数を実行して、その戻り値を変数に代入したりせず何も処理しないとき、Google ColaboratoryやJupyter Notebookの独自機能で、戻り値が画面に出力されます。

しかし、画面に出力するprint関数と、戻り値を返すreturn文は全く別の機能です。print関数は値を画面に出力するだけで戻り値は返しませんし、return文は戻り値を返すだけで、画面に出力はしません。しっかりと理解をするようにしましょう。

エラー、例外

Pythonにおけるプログラムのエラーには、次の2種類があります。

- 構文エラー(syntax error)
- 例外 (exception)

構文エラー

「**構文エラー**」はその名のとおり、ソースコードへ記述した処理に構文の誤りがある場合に発生します。「(」に対する「)」を書いていない、if文で条件式の末尾に「:(セミコロン)」が書かれていない、といった場合に発生するエラーです。

次のプログラムを実行してみましょう。「SyntaxError」が発生することが分かります。

In [ ]:

```
print("Start")
num_list = [1, 2, 3,
print(num_list)
```

構文エラーがあると、そもそもプログラムの実行が開始しません。

あらかじめ、構文エラーが発生しないようにソースコードを記述する必要があります。

例外

ソースコードに構文エラーがなければ、プログラムは実行を開始します。しかし、それでも実行中に何らかのエラーが発生することがあります。

このように、ソースコードの構文に問題がなくても、プログラム実行中に発生するエラーを「**例外**」と言います。

例えば、次のプログラムを実行してみてください。何が起きるでしょうか。

In [ ]:

```
div_value = 0
print(div_value)
print(100 / div_value)
```

print(div_value)の結果として0が出力されましたので、プログラムの実行は開始しています。

しかし、0で割り算しようとしたことでプログラムがエラーになりました。これは、「ZeroDivisionError」という例外です。

この例では、プログラムの構文の誤りはありません。しかし、これ以上プログラムを継続できないために例外となり、プログラムの実行は中止されます。

この例はごく簡単なものなので、「明らかに間違っているのだから、構文エラーのようにプログラム実行前の時点でエラーと分かるのでは？」と思われるかもしれません。

しかしPythonではあくまで、プログラム実行前には、構文が正しいかどうかのチェックしか行いません。

例えばデータ構造化プログラムにおいて、CSVファイルから読み込んだ値を使って割り算をするとしたら、どうでしょうか。

プログラムを開始して、実際にファイルを読み込んでみるまで、どのような値が出てくるかは分かりません。そもそもファイルのフォーマットが不正で、読み込むことさえできないかもしれません。

こういったケースから、例外は構文エラーとは異なり、プログラム実行前の時点ではエラーかどうか分からないことがイメージしていただけると思います。

例外を処理する

次のソースコードを見てみましょう。リストの文字列を順番に取り出して、浮動小数点数へ変換し、変換後の値で割り算を行います。

なお「**float関数**」は、浮動小数点数型でないデータを、浮動小数点数型に変換する関数です。例えば文字列"2.0"を引数に渡すと、浮動小数点数型の「2.0」に変換してくれます。

In [ ]:

```
for numstr in ["2.5", "0", "Number", "3.5", "4.5"]:
    num = float(numstr)
    value = 100 / num
    print(value)
```

最初の"2.5"は問題ありませんが、次の"0"でZeroDivisionErrorとなります。

【試してみましょう】リストから"0"を削除したら、どうなるでしょうか。結果を予想して、プログラムを動かして試してみましょう。

エラーへの対応

"0"や数字でない文字列が出現したら、その場でプログラムがエラー終了してよいのであれば、このままでも問題ありません。

しかし例えば、"0"や数字でない文字列が出現しても警告メッセージを出力して処理を続行し、リストの最後まで実行を続けたい("3.5"や"4.5"を処理したい)場合は、どうすればよいでしょうか。

ZeroDivisionErrorへの対応

次のようにif文を使えば、ZeroDivisionErrorが発生しないようにできます。

In [ ]:

```
for numstr in ["2.5", "0.0", "Number", "3.5", "4.5"]:
    num = float(numstr)
    if num == 0:
        print("警告：値が0です")
    else:
        value = 100 / num
        print(value)
```

ValueErrorへの対応

float関数の引数に"Number"といった数字でない文字列を渡すと、「ValueError」という例外が発生します。

このValueErrorについても、ZeroDivisionErrorと同じように、あらかじめif文で、float関数が変換できる文字列であるかをチェックすることはできます。

しかし、float関数が変換できる文字列は、次のように色々なパターンがあります。

- 3、3.0、+3、-3.0、5e3、5e-3、 3 (全角) などなど

これらをif文でチェックしようとすると、かなり大変です。もっと簡単に、チェックできないものでしょうか。

try～except

Pythonでは、プログラムを実行して例外が発生する可能性があるとき、「もしこの例外が発生したらこうする」とあらかじめ定めておくことができます。

これには次のように、「**try文**」を使用します。

```
try:
    例外が発生する可能性のある処理
except 発生しうる例外1
    例外1が発生したときの処理
except 発生しうる例外2
    例外2が発生したときの処理
            :
```

- try文に続けて、「例外が発生する可能性のある処理」を記述します。
- try文とセットで記述する「**except**」で、「例外1が発生したときの処理」を記述します。この処理を実行した後、プログラムはエラー終了することなく処理を継続します。
- 発生しうる例外が複数ある場合は、それぞれの例外ごとにexceptを複数記述します。

実際のプログラムの例を見てみましょう。

In [ ]:

```
for numstr in ["2.5", "0.0", "Number", "3.5", "4.5"]:
    try:
        num = float(numstr)
        if num == 0:
            print("警告：値が0です")
        else:
            value = 100 / num
            print(value)
    except ValueError:
        print("警告：値が数字ではありません")
```

「ZeroDivisionErrorへの対応」で示したソースコードに、try～exceptの処理を追加しています。

ValueErrorが発生したら、exceptで定義されたprint関数で警告メッセージを出力して、for文の繰り返しを継続できます。

ifとtry～exceptの使い分け

ここまでの例を見て、ZeroDivisionErrorもexceptで処理すればよいのでは？と考えたかもしれません。確かに、「発生しうる例外が複数ある場合は、それぞれの例外ごとにexceptを複数記述します。」とご説明したとおり、次のように記述することもできます。

In [ ]:

```
for numstr in ["2.5", "0.0", "Number", "3.5", "4.5"]:
    try:
        num = float(numstr)
        value = 100 / num
        print(value)
    except ValueError:
        print("警告：値が数字ではありません")
    except ZeroDivisionError:
        print("警告：値が0です")
```

ifとtry～exceptの使い分けについての明確な線引きはありませんが、基本的には、次のように考えることが多いです。

- if文でどのようにチェックしても例外が発生する可能性が残ってしまう場合は、try～exceptを使う。
- if文でチェックするとチェックが多すぎたり、ソースコードが読みづらくなったりする場合は、try～exceptを使う。
- 上記以外は、if文を使う。

ValueErrorの例は、上記の2点目に該当するといえます。

try～exceptに関するその他の機能

try文には他にも、「例外が発生しなかったときだけ動く処理」を定義するelseや、「例外が起きても起きなくても必ず動く処理」を定義するfinallyなどがあります。

```
try:
    例外が発生する可能性のある処理
except 発生しうる例外1
    例外1が発生したときの処理
except 発生しうる例外2
    例外2が発生したときの処理
                :
else:
    例外が発生しなかったときだけ動く処理
finally:
    例外が起きても起きなくても必ず動く処理
```

また、例外の詳しい情報を取得したり、例外を自作したりすることもできます。本講座ではこれ以上の詳しい説明はしませんが、try文を実際に使うときにはぜひ、例外について色々調べてみてください。

【小ネタ】

try文には「例外が発生する可能性のある処理」を記述します。ところで、そもそも、どんな例外が発生する可能性があるか、どうやって知るのでしょうか。

実は、Pythonが提供する関数や演算子などで、どのような例外が発生する可能性があるのか、公式ドキュメントを見ても書いていません。それこそPython実行環境そのもののソースコードを調査しないと、正確なところは分かりません。

そのため、次のようにする必要があります。

- 知識・経験から知っている例外についてあらかじめ処理を入れておく
- 実際にプログラムを動かしてみたら例外が起きてしまったので、そこで初めて例外処理を入れる

他のプログラム言語では、発生するエラーが公式ドキュメントに明記されているものもあります。

【注意事項】

Pythonには「**pass**」という構文があります。これは、「何もしない命令」です。例えば、ある例外が起こってもプログラムの動作に影響がないことが分かっているため、無視して処理を続行する、という場合に使用します。

次の例ではZeroDivisionErrorが起きていますが、passすることで何もせず処理を継続します。

In [ ]:

```
for numstr in ["2.5", "0", "Number", "3.5", "4.5"]:
    try:
        num = float(numstr)
        value = 100 / num
        print(value)
    except ZeroDivisionError:
        pass
    except ValueError:
        print("警告：値が数字ではありません")
```

くれぐれも気を付けていただきたいのは、「何か例外が起きるけど、よく分からないからpassして動かしちゃえ！」としないでください、ということです。

たとえば、極端なプログラムとして、下記のようなものがあります。

```
try:
    例外が発生する可能性のある処理1
    例外が発生する可能性のある処理2
                :
except:
    pass

継続処理
```

exceptに例外名を指定しないと、「すべての例外」を意味します。このプログラムはどんな例外が起こっても、無視してプログラムが継続します。しかし、もし例外が起きていたら、プログラムが期待通りの動作をしたという保証はありません。

上記のようなプログラムにはせず、例外が発生したらきちんと調査をして、exceptで適切な処理をしてください。もちろん、調査した結果、無視してよいと確認できたのであれば、passを使っても問題ありません。

予約語

Pythonでは、あらかじめ予約された語句が定められています。このような語句を変数名や関数名として使用すると、プログラムが正しく動作しないことがあるため、注意が必要です。

- 予約語の例
  - if、forなど制御文の名前
  - print、len、strなど組み込み関数の名前

上記以外にも、Pythonの機能としてあらかじめ定められた語句が予約語となっています。

次のソースコードでは、予約語のprintを変数名として使用したため、後続の処理でprint関数が使えなくなってしまう例です。プログラムを実行すると、print関数が実行できずにエラーとなります。

In [ ]:

```
# 予約語であるprintを変数名として使用
print = "abc"

# print関数が使えなくなってしまう
print("Hello, world.")
```

【注意事項】上記のソースコードを実行したあとは、必ずGoogle Colaboratoryの画面上部にあるメニューから、[ランタイム]→[ランタイムを再起動]を実行してください。この操作をしないと、本資料のソースコードで使用されているすべてのprint関数も動作しなくなります。

注意すべき用語

以上で、Pythonの基本的な構文の紹介を終わります。

最後に、Pythonにまつわる様々な用語のうち、使う人や文脈により、同じ意味でも異なる表現となるような用語をご紹介します。

ソースコード、スクリプト

第1回の「Pythonの実行環境・エディタの紹介」でご紹介したとおり、Pythonのプログラムを作成するときは、最初にエディタで「ソースコード」を作成します。

しかし人・書籍などによっては、ソースコードの代わりに「**スクリプト**」「**Pythonスクリプト**」などと呼ぶこともあります。

様々なプログラミング言語のソースコードのうち、あらかじめコンピュータが理解できる形式に変換(コンパイル)しなくても、そのまま実行できるものをスクリプトと呼びます。

Pythonのソースコードはコンパイルせずにそのまま実行できるので、ソースコードと同じ意味でスクリプトと呼ぶことがあります。

本講座では、「ソースコード」に呼び方を統一しています。

モジュール、パッケージ、ライブラリ

第1回の「Pythonの実行環境・エディタの紹介」で、Pythonは、非常に多くのさまざまな機能が「パッケージ」という形で公開されているとご紹介しました。

ここでは「パッケージ」と、それに関連する用語として「モジュール」「ライブラリ」についてご紹介します。

モジュール

Google ColaboratoryやJupyter Notebookでは、Webブラウザの画面上でソースコードを記述するのであまり意識しませんが、通常、ソースコードは拡張子「.py」のテキストファイルに記述して保存します。

この拡張子「.py」のファイルは、「**モジュール**」と呼びます。

「モジュール」は前述の「ソースコード」とも近い用語ではありますが、あまり混同されることなく、区別して使われていることが多いです。

「ソースコード」は「.py」ファイルの中身(記述された処理)、「モジュール」は「.py」ファイルそのもの、というニュアンスが一般的です。

パッケージ

「**パッケージ**」は、ある機能を提供するために必要な複数のモジュールをまとめたものです。

ライブラリ

パッケージに近い用語として「**ライブラリ**」があります。

しかし、モジュールとパッケージはPythonの公式ドキュメントで定義された用語なのですが、ライブラリという用語には厳密な定義がありません。

ライブラリは、主に次のような意味で使われます。

- パッケージよりも大きな機能の括りで、複数のパッケージをまとめたもの
- パッケージ、モジュール、関数など、様々なプログラムで汎用的に使用する前提で作られた部品の総称
- パッケージと同じ意味

使われ方がケースバイケースな用語ですが、非常によく使われる用語なので、覚えておきましょう。

本講座の資料では、Python公式ドキュメントの用語に合わせ、「モジュール」「パッケージ」の用語を使用しています。

最後に～アンケートご協力のお願い

第1回・第2回にて、Pythonの基本的な構文についてご紹介しました。ここまでの講義について、アンケートのご協力をお願いいたします。

アンケートURL (Teamsのチャットでもお知らせします)

https://docs.google.com/forms/d/e/1FAIpQLSf1b5di5sEVGbnWI1NICiOkpiE1LWWI2nWYMbB-eh3QCPpJgQ/viewform?usp=sf_link (https://docs.google.com/forms/d/e/1FAIpQLSf1b5di5sEVGbnWI1NICiOkpiE1LWWI2nWYMbB-eh3QCPpJgQ/viewform?usp=sf_link)

おまけ

下記は、何のプログラムでしょう？？

パッケージをインストールするpipコマンドを除くと、たったの4行しかありませんが・・・

In [ ]:

```
# 必要なパッケージのインストール
!pip install qrcode
```

In [ ]:

```
# プログラム本体
import qrcode
from IPython.display import display
img = qrcode.make("https://docs.google.com/forms/d/e/1FAIpQLSf1b5di5sEVGbnWI1NICiOkpiE1LWWI2nWYMbB-e
display(img)
```