

---

# Table of Contents

|                              |        |
|------------------------------|--------|
| 1. Introduction              | 1.1    |
| 2. Python Grammer            | 1.2    |
| 1.1. Hello World             | 1.2.1  |
| 1.2. Arithmetic Operations   | 1.2.2  |
| 1.3. Variable                | 1.2.3  |
| 1.4. String                  | 1.2.4  |
| 1.5. Conditional Branch      | 1.2.5  |
| 1.6. Loop - while            | 1.2.6  |
| 1.7. Loop - for              | 1.2.7  |
| 1.8. Function                | 1.2.8  |
| 1.9. Input & Cast            | 1.2.9  |
| 1.10. Data Structures - list | 1.2.10 |

---

# Introduction

本テキストは *Competitive Programming with Python* です。

Pythonで競技プログラミングを行うために必要な基礎知識やテクニックについて解説していきます。

## 注意事項

- 使用するPythonのバージョンは 3.7.0 を想定しています。
- Pythonの環境構築の方法については省略します。
- テキスト内では一部対話型環境を用いていますが、試す際は IPython や Jupyter Notebook 等のREPL環境を使用することを推奨します。
- 記載内容は理解しやすさを優先するために、厳密には不正確な内容が含まれていることがあります。
- 指摘等があれば [https://github.com/Szarny/cp\\_with\\_python](https://github.com/Szarny/cp_with_python) の Issue へお願いします。

## Author

- Tsubasa Umeuchi (<https://github.com/Szarny>)

## Python Grammer

本章では、Pythonの基礎的な文法について記述します。

なお、本章では以下のようなプログラミングに関する初步的な内容を理解していることを前提としています。

- 型
- 入出力
- 関数の概念
- 基礎的なデータ構造

## 1.1. Hello World

### Hello World

Pythonの基本出力は `print(...)` と記述します。

文字列はダブルクオート("), もしくはシングルクオート(')で囲みます。

```
>>> print("Hello, world!")
Hello, world!
```

注: プログラム中の `>>>` はコンソール上で自動的に出力されるものであるため、入力の必要はありません。

### Practice

`Hello, Python!` と出力してください。

## 1.2. Arithmetic Operations

### 演算子

Pythonで用いることのできる、主な演算子は以下の通りです。

| 演算子 | 概要       |
|-----|----------|
| +   | 加算       |
| -   | 減算       |
| *   | 乗算       |
| **  | べき乗      |
| /   | 除算       |
| //  | 除算(切り捨て) |
| %   | 剰余       |

以下のようにすることで、電卓としても活用することができます。

```
>>> 9 + 4
13
>>> 9 - 4
5
>>> 9 * 4
36
>>> 9 ** 4
6561
>>> 9 / 4
2.25
>>> 9 // 4
2
>>> 9 % 4
1
```

また、`()`を用いることで、演算の順序を入れ替えることもできます。

なお、Pythonにはインクリメント演算子(`++`)やデクリメント演算子(`--`)は存在しません。  
後述の累算代入文を利用して下さい。

### Practice

以下の計算式を計算してください。

$$3 \left( \left( 5^2 + \frac{4}{3} \right) - 33 \times 3 + 20 \right) + 281$$



## 1.3. Variable

### 変数の扱い方

Pythonにおいて、変数は宣言や型を指定することなく利用することができます。

```
>>> height = 1.7
>>> weight = 60
>>> bmi = weight / (height ** 2)
>>> print(bmi)
20.761245674740486
```

### 再代入

一度代入された変数に異なる型の値を代入することも可能です。

```
>>> my_variable = 1
>>> print(my_variable)
1
>>> my_variable = "hello"
>>> print(my_variable)
hello
>>> my_variable = 3.14
>>> print(my_variable)
3.14
```

### 未使用変数

但し、一度も代入が行われていない変数を扱おうとするとエラーが発生します。

```
>>> print(your_variable)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'your_variable' is not defined
```

### 累算代入文

前述の通り、なお、Pythonにはインクリメント演算子( `++` )やデクリメント演算子( `--` )は存在しません。

そのため、以下のような累算代入文を使用します。

```
+ = - = * = /= // = %= ** =
```

累算代入文は以下のように用います。

```
>>> age = 0
>>> age += 1
>>> print(age)
1
>>> age += 10
>>> print(age)
11
>>> age *= 2
>>> print(age)
22
```

## Practice

あなたは123456円の商品を買おうとしています。ここで、消費税率は8%とします。

商品価格を `price`、消費税率を `tax` という変数にそれぞれ代入し、購入に必要な金額を計算してください。

## 1.4. String

### 文字列の基本的な扱い方

文字列は、ダブルクオート(")，もしくはシングルクオート(')で囲むことで定義できます。

```
>>> greeting = "hello"
>>> print(greeting)
hello
```

### 文字列と演算子

+ 演算子を用いると、文字列同士をくっつけることができます。

```
>>> front = "Pyt"
>>> rear = "hon"
>>> combine = front + rear
>>> print(combine)
Python
```

\* 演算子を用いると、文字列を繰り返すことができます。

```
>>> print("hey!" * 10)
hey!hey!hey!hey!hey!hey!hey!hey!hey!
```

### 文字列長

`len` 関数を用いると、文字列の長さを取得することができます。

```
>>> print(len("hogehogefugafugapiyopiyo"))
24
```

### 文字列とインデックス

Pythonでは、文字列の一部をインデックスによって抜き出すことができます。

```
# 0-indexed であることに注意
>>> s = "Competitive Programming with Python"
>>> print(s[0])
C
>>> print(s[1])
o
```

なお、インデックスに負数を指定すると、末尾から見た時の文字を返します。

```
>>> print(s[-1])
n
```

## 文字列とスライス

文字列を抜き出すとき、スライスという記法を用いることができます。

`string[start:end:step]` というように記述します。

これは `string[start]` から `string[end-1]` までの文字列を `step` 文字ごとに切り出す という意味になります。

```
>>> string = "Programming"
>>> print(string[0:7:1])
Program
>>> print(string[0:7:2])
Porm
```

`step` が不要ないときは、省略することも可能です。

```
>>> print(string[0:9])
Programmi
```

始端や終端を指定したくないときは、その部分を空欄にしておきます。

空欄にした場合、始端なら「最初から」 終端なら「最後まで」という意味になります。

```
>>> print(string[3:])
gramming
>>> print(string[:3])
Pro
```

なお、`step` に負数を指定すると逆順になります。

```
>>> print(string[::-1])
gnimmargorP
>>> print(string[::-2])
gimroP
```

## 文字列への変数埋め込み

`str.format` を利用することで、文字列オブジェクトに対して変数に格納された値を埋め込むことができます。

文字列オブジェクト内に存在する `{}` が `format` メソッドの引数として与えられた変数の値と置換されます。

以下に実例を示します。

```
>>> height = 170
>>> weight = 60
>>> print("Height is {}, Weight is {}".format(height, weight))
Height is 170, Weight is 60
```

なお、`{}` 内にパラメタを指定することで、ゼロ埋めや桁数指定、進数指定等を行うこともできます。

## Practice

- 以下の文字列の2文字目から35文字目までを3文字区切りで抜き出した文字列はなんですか？

F3Ibr F3<fe3oo 0pP Pyuit14<bvoVwn+cfe3 1#

- 変数 `s` には文字列が格納されているとします。この時、以下のコードは同義ですか？

`s[:] s[:len(s)]`

## 1.5. Conditional Branch

### 条件式とbool値

Pythonではbool値として、真である `True` と偽である `False` が定義されています。  
`if`文等で用いる条件式は評価の結果、これらの `True` か `False` かのどちらかとなります。  
 大文字で始まる点に注意してください。

### 条件分岐

Pythonにおいて、条件分岐命令は以下のように記述します。  
 CやJavaのように `else if` ではなく、`elif` と記述する点に注意してください。

```
if 条件式1:
    条件式1がTrueの時の処理
elif 条件式2:
    条件式1がFalseかつ条件式2がTrueの時の処理
elif 条件式3:
    条件式1, 条件式2がFalseかつ条件式3がTrueの時の処理
...
elif 条件式N:
    条件式1 ~ 条件式N-1がFalseかつ条件式NがTrueの時の処理
else:
    どの条件式もFalseだった時の処理
```

### 条件式と演算子

条件式には、主に以下のような比較演算子を用います。

| 演算子                | 例                            | 概要   |
|--------------------|------------------------------|--|
| <code>==</code>    | <code>hoge == fuga</code>    | <code>hoge</code> と <code>fuga</code> が等しいなら <code>True</code>   |
| <code>!=</code>    | <code>hoge != fuga</code>    | <code>hoge</code> と <code>fuga</code> が等しくないなら <code>True</code> |
| <code>&gt;</code>  | <code>hoge &gt; fuga</code>  | <code>fuga</code> より <code>hoge</code> が大きいなら <code>True</code>  |
| <code>&gt;=</code> | <code>hoge &gt;= fuga</code> | <code>hoge</code> が <code>fuga</code> 以上なら <code>True</code>     |
| <code>&lt;</code>  | <code>hoge &lt; fuga</code>  | <code>hoge</code> より <code>fuga</code> が大きいなら <code>True</code>  |
| <code>&lt;=</code> | <code>hoge &lt;= fuga</code> | <code>hoge</code> が <code>fuga</code> 以下なら <code>True</code>     |

なお、複数の条件式を元に論理演算を行う場合は、以下の論理演算子を用います。

| 演算子              | 例                          | 概要   |
|------------------|----------------------------|--|
| <code>and</code> | <code>hoge and fuga</code> | <code>hoge</code> と <code>fuga</code> が両方 <code>True</code> なら <code>True</code>       |
| <code>or</code>  | <code>hoge or fuga</code>  | <code>hoge</code> と <code>fuga</code> の少なくとも片方が <code>True</code> なら <code>True</code> |

|     |          |                  |
|-----|----------|------------------|
| not | not hoge | hogeがFalseならTrue |
|-----|----------|------------------|

## 条件分岐の例(奇遇判定)

```
>>> oddnum = 1
>>> evennum = 2
>>>
>>> if oddnum % 2 == 0:
...     print("Even!")
... else:
...     print("Odd!")
...
Odd!
>>> if evennum % 2 == 0:
...     print("Even!")
... else:
...     print("Odd!")
...
Even!
```

## Practice

変数 `num` になんらかの値を代入し

- `num` が10未満なら `Low`
- `num` が10以上100未満なら `Middle`
- `num` が100以上なら `High`

と出力する条件分岐を記述してください。

## 1.6. Loop - while

### while文

while文は以下のように記述します。

これは、「条件式が True の間処理を繰り返す」という意味になります。

```
while 条件式:  
    処理
```

### while文の例

```
>>> num = 1  
>>> while num <= 10:  
...     print(num)  
...     num += 1  
...  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

### break

break キーワードを用いることで、その時点での最も内側のループを抜けることができます。

「ある条件が満たされた(満たされなくなった)時にループを抜けたいが、いつそうなるかは分からない」といった時に、無限ループやフラグ変数と一緒によく用いられます。

以下で実例を示します。

### breakの実例

文字列 s について、最初にピリオドかカンマが出現する部分より前の文を抽出したいとします。この時、以下のように記述することができます。

```
>>> s = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod te  
mpor incididunt ut labore et dolore magna aliqua."  
>>> idx = 0
```

```
>>> while True:  
...     if s[idx] == "," or s[idx] == ".":  
...         break  
...     idx += 1  
  
>>> print(s[:idx])  
Lorem ipsum dolor sit amet
```

## Practice

- `while` を利用して、`1` から `10000` までの総和を求めてください。
- `1+2+...` を計算するとき、総和が初めて `1000` を超えるのはどの値を足したときですか？  
`while` と `break` を利用して求めてください。

## 1.7. Loop - for

### for文

`for` も `while` と同様に、繰り返し処理を実行させる際に利用します。以下のように記述します。

```
for 変数 in 反復可能オブジェクト:  
    処理
```

ここでいう `反復可能オブジェクト` とは、読んで字のごとく繰り返しが可能なモノのことです。具体的には、文字列や後述する `range` , `list` , `tuple` 等があります。

### forと文字列

`反復可能オブジェクト` の部分に文字列をセットすると、ループが1回実行されるたびに文字列に含まれる文字が順番に1文字ずつ変数にセットされます。

```
>>> string = "hello"  
>>> for letter in string:  
...     print(letter)  
...  
h  
e  
l  
l  
o
```

### forとrange

`反復可能オブジェクト` の部分に `range` をセットすると、指定した範囲やステップの整数値がループが実行されるたびに変数にセットされます。

厳密にいうと、`range` はシーケンスを生成するオブジェクトです。

`range` の記述方法とその意味を以下に示します。

| 記述法                         | 意味   |
|-----------------------------|--|
| <code>range(N)</code>       | <code>0</code> から <code>N-1</code> まで                    |
| <code>range(N, M)</code>    | <code>N</code> から <code>M-1</code> まで                    |
| <code>range(N, M, S)</code> | <code>N</code> から <code>M-1</code> まで <code>S</code> ごとに |

以下に実例を示します。

```
>>> for i in range(5):
```

```

...     print(i)
...
0
1
2
3
4
>>>
>>> for i in range(2, 5):
...     print(i)
...
2
3
4
>>>
>>> for i in range(1, 10, 2):
...     print(i)
...
1
3
5
7
9

```

## 複数のオブジェクト(zip)

複数のオブジェクトを1つのforループの中で同時に処理したいときは `zip` 関数を用います。  
以下のように記述します。

```

for 変数1, 変数2, ..., 変数N in zip(オブジェクト1, オブジェクト2, ..., オブジェクトN):
    処理

```

以下に実例を示します。

```

>>> s1 = "abc"
>>> s2 = "def"
>>> s3 = "ghi"
>>>
>>> for l1, l2, l3 in zip(s1, s2, s3):
...     print(l1, l2, l3)
...
a d g
b e h
c f i

```

## enumerate

オブジェクトの要素だけでなく添え字も同時に取得したいときは `enumerate` 関数を用います。以下のように記述します。

```
for 添字変数, 要素変数 in enumerate(オブジェクト):
    処理
```

以下に実例を示します。

```
>>> s = "hello"
>>> for i, l in enumerate(s):
...     print(i, l)
...
0 h
1 e
2 l
3 l
4 o
```

なお、`enumerate` 関数の `start` 引数に値を指定することで、開始値を変更することができます。

```
>>> s = "hello"
>>> for i, l in enumerate(s, start=100):
...     print(i, l)
...
100 h
101 e
102 l
103 l
104 o
```

## Practice

- `for` と `range` を利用して、条件分岐を用いずに `1` から `100` の範囲に存在する偶数のみを出力してください。
- `for` と `enumerate` を利用して、文字列 `ajiemwvtwamuoimufiosufcvwtwapocuwjepmovawemwapo` の30文字目の文字を特定してください。

## 1.6. Function

### 関数の基本

Pythonにおいて、関数は `def` キーワードを用いて以下のように記述することで定義することができます。

必ずインデントを揃えないといけないことに注意してください。

```
def 関数名(引数1, 引数2, ..., 引数N):
    ...
    return 戻り値
```

### 関数の例

`succ` 関数は、受け取った値に+1した値を返却します。

```
>>> def succ(k):
...     return k + 1
...
>>> print(succ(1))
2
>>> print(succ(12345))
12346
```

### 位置引数

Pythonでは、関数の引数を指定するときに、目的の値を渡す引数名を明示することができます。関数を呼び出すときに、`関数名(引数名=値, 引数名=値, ...)` と記述することで、特定の引数にその値を渡すことを明示できます。

位置引数を利用すると、どの引数に値を渡しているのかが分かりやすくなったり、引数の順番を意識しなくても良くなったりするというメリットがあります。

```
>>> def calc_triangle_area(bottom, height):
...     return bottom * height / 2
...
>>>
>>> # 以下の3つの関数呼び出しは全て同一の意味になります
>>> print(calc_triangle_area(5, 6))
15.0
>>> print(calc_triangle_area(bottom=5, height=6))
15.0
>>> print(calc_triangle_area(height=6, bottom=5))
15.0
```

## Practice

- 引数 `k` を受け取り
  - `k` が3で割り切れれば `Fizz` を返す
  - `k` が5で割り切れれば `Buzz` を返す
  - `k` が3でも5でも割り切れれば `FizzBuzz` を返す
  - `k` が3でも5でも割り切れなければ `k` を返す

という動作を行う関数 `fizzbuzz` を作成してください。

- 上で作成した関数に対して、`for` と `range` を利用して `1` から `100` までの引数を渡し、正しく動作するかテストしてください。

## 1.9. Input & Cast

### Input

標準入力は `input` 関数を用いて行うことができます。

```
>>> s = input()
hello! # ここはユーザの入力

>>> print(s)
hello!
```

`input` 関数に引数を与えると、入力時にプロンプトとして表示させることができます。

```
>>> name = input("What's your name? : ")
What's your name? : hoge # ここはユーザの入力

>>> print("Hello, {}!".format(name))
Hello, hoge!
```

### input関数と型変換

`input` 関数によって受け取った値は全て文字列型( `str` )となります。

そのため、入力された数値をそのまま用いようとすると想定外の動作が発生します。

```
>>> not_number = input()
100 # ここはユーザの入力
>>> print(not_number + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

そのため、型変換のための関数を用いる必要があります。

型変換のための関数には、以下のようなものが存在します。

| 関数                     | 変換元                    | 変換先                                      |
|------------------------|------------------------|--|
| <code>int()</code>     | <code>str</code>       | <code>int</code>                         |
| <code>int(a, b)</code> | <code>str</code>       | b 進数である文字列 a を10進数の <code>int</code> に変換 |
| <code>str()</code>     | <code>int</code>       | <code>str</code>                         |
| <code>chr()</code>     | <code>int</code>       | 入力された数値をASCIIコードとしてみたときの文字               |
| <code>ord()</code>     | <code>str (1文字)</code> | 入力された文字に対応するASCIIコード                     |
|                        |                        |  |

|                    |                  |     |
|--------------------|------------------|-----|
| <code>oct()</code> | <code>int</code> | 8進数 |
| <code>bin()</code> | <code>int</code> | 2進数 |

今回の場合、`str` を `int` に変換したいので `int()` を用います。

```
>>> number = int(input())
100 # ここはユーザの入力
>>> print(number + 1)
101
```

## Practice

- ユーザから身長(メートル単位)と体重(キログラム単位)を標準入力で受け取り、BMIを計算するプログラムを作成してください。  
BMIは `体重 / (身長 * 身長)` で求められます。
- ユーザから標準入力で10進数の数値を受け取り、2進数、8進数、16進数に変換するプログラムを作成してください。
- 以下の2進数はある文字列のASCIIコードです。文字列に復元してください。  
`1010000 1111001 1110100 1101000 1101111 1101110`

## 1.10. Data Structures - list

### listとは

listは可変長配列です。

また、格納する型や要素数に制限はないため、(メモリが許す限り)好きなデータを好きなだけ格納させることができます。

### listの定義方法

空のlistを生成するときは、`[]` か `list()` を用います。

```
>>> empty_list1 = []
>>> print(empty_list1)
[]

>>> empty_list2 = list()
>>> print(empty_list2)
[]
```

最初からデータを持たせることも可能です。

```
>>> num_list = [1,2,3,4,5]
>>> print(num_list)
[1, 2, 3, 4, 5]
```

### listの参照方法

listの参照方法は、文字列に対するそれと同様に行うことができます。

- 特定の要素にアクセスするときは添字で指定する
- 複数の要素にアクセスするときはスライス記法(`[start:stop:step]`)を用いる
- `list`の長さ(要素数)は `len()` によって取得できる。

以下に例を示します。

```
>>> lang_list = ["Python", "Ruby", "C", "Swift", "Java", "Kotlin", "Rust"]

>>> print(lang_list[0])
Python

>>> print(lang_list[-1])
Rust

>>> print(lang_list[2:5])
```

```
[ 'C', 'Swift', 'Java']

>>> print(lang_list[::-1])
['Rust', 'Kotlin', 'Java', 'Swift', 'C', 'Ruby', 'Python']

>>> print(lang_list[::2])
['Python', 'C', 'Java', 'Rust']

>>> print(len(lang_list))
7
```

## listの操作方法

listには、以下のようなメソッドが用意されています。

| メソッド                                 | 内容  |
|--------------------------------------|---|
| <code>list.append(x)</code>          | listの末尾に <code>x</code> を追加                 |
| <code>list.insert(i, x)</code>       | listの <code>i</code> 番目に <code>x</code> を挿入 |
| <code>list.index(x)</code>           | 先頭から見てlist内で <code>x</code> が初めて現れる位置の添字を返却 |
| <code>list.count(x)</code>           | list内で <code>x</code> が現れる回数を返却             |
| <code>list.remove(x)</code>          | 先頭からみてlist内で初めて現れる <code>x</code> を削除       |
| <code>list.pop(i)</code>             | listの <code>i</code> 番目の要素を削除               |
| <code>list.sort()</code>             | listを昇順に並べ替え                                |
| <code>list.sort(reverse=True)</code> | listを降順に並べ替え                                |

また、list固有の操作ではありませんが、以下の組み込み関数は競技プログラミング内でlistに対して用いられることが多いです。

| 関数                                      | 内容                 |
|---|--------------------|
| <code>max(引数1, 引数2, ..., 引数N)</code>    | 引数の中で最大の値を返却       |
| <code>max(list)</code>                  | listの中で最大の値を返却     |
| <code>min(引数1, 引数2, ..., 引数N)</code>    | 引数の中で最小の値を返却       |
| <code>min(list)</code>                  | listの中で最小の値を返却     |
| <code>sorted(list)</code>               | listを昇順に並べ替えた結果を返却 |
| <code>sorted(list, reverse=True)</code> | listを降順に並べ替えた結果を返却 |
| <code>sum(list)</code>                  | listの要素の総和を返却      |

## listとfor

`list`は文字列と同様に反復可能オブジェクトなので、`for`と併用できます。  
記述方法は、 またもや文字列のそれと全く同じです。

```
>>> lang_list = ["Python", "Ruby", "Rust"]
>>>
>>> for lang in lang_list:
...     print(lang)
...
Python
Ruby
Rust
```

もちろん、`zip` や `enumerate` も使えます。

```
>>> lang_list = ["Python", "Ruby", "Rust"]
>>>
>>> for i, lang in enumerate(lang_list):
...     print("lang_list[{}] = {}".format(i, lang))
...
lang_list[0] = Python
lang_list[1] = Ruby
lang_list[2] = Rust
```

## Practice

- `["A", "P", "B", "o", "C", "n"]` という値を格納したリスト `L` があります。  
「listの操作方法」で示したメソッドのみを用いて `["P", "y", "t", "h", "o", "n"]` にしてください。
- 「listの操作方法」で示したメソッドを活用して、  
`5, 3, 4, 77, 5, 44, 14, 432, 543, 53, 3, 5, 77, 3, 1, 4, 6, 7, 8, 4, 5, 62, 657, 5, 4, 526, 7, 8, 9, 36, 73, 7, 22`  
 の中央値を求めてください。
- 任意のlistを引数に取り、以下の動作を行う関数を作成してください。ただし、listの要素が数値以外であることを考慮する必要はありません。
  - 偶数番目に位置する要素は加算する
  - 奇数番目に位置する要素は減算する

完成した関数に以下のlistを与え、正しい結果になるか確認してください。

- `[] -> 0`
- `[1, 2, 3] -> 2`
- `[11, -22, 33, -44, 55] -> 165`
- `[-1] -> -1`

