

情報 I

～第3編 第2章 プログラミング～

____年 ____組 ____番 ____氏名_____

【はじめに】

本章は、アルゴリズムの構築を通して **論理的思考を養うことが目的** である。アルゴリズムを考える上でツールとしてプログラミング言語 Python を使うが、プログラミング言語の習得が目的ではないので注意すること。また、共通テストも見据えて、DNCL と呼ばれる擬似言語も併記する。

※DNCL の言語仕様については、2023 年 09 月時点での公開情報に準拠する

【論理的思考を養うために身につける3つのこと】

1. コンピュータによる「演算の性質」と「自動処理の有用性」の理解
2. アルゴリズムの「表現方法」の習得
3. 事象の「モデル化」と「シミュレーション」の考え方の問題解決への活用

【Python とは】

次のような特徴があるため、授業ではプログラミング言語として「Python」を利用する

- ☐ 動かすために覚えなければならないルールが少ない
- ☐ 利用者が多いため調べたときに資料が見つけやすい
- ☐ パッケージと呼ばれる追加機能が豊富であり現在人気急上昇中の「人工知能」も手軽に利用できる
- ☐ DNCL (Daigaku Nyushi Center Language) という共通テスト用日本語プログラミング言語と互換性がある

【プログラミングを学ぶ上で】

1. 落ち着きましょう。冷静になりましょう。
2. 脳が疲れたら、たくさんの水を飲み、甘い物を食べましょう。
3. 記憶ではなく、思考と試行に時間をかけて、想造しましょう。
4. 自分なりの感想を持ちましょう。印象に残ることが理解の始まりです。
5. プログラムは、仲間と考え、自分で試行錯誤し、最後は自力で完成させましょう。

(書籍『Head First Java』より意識)

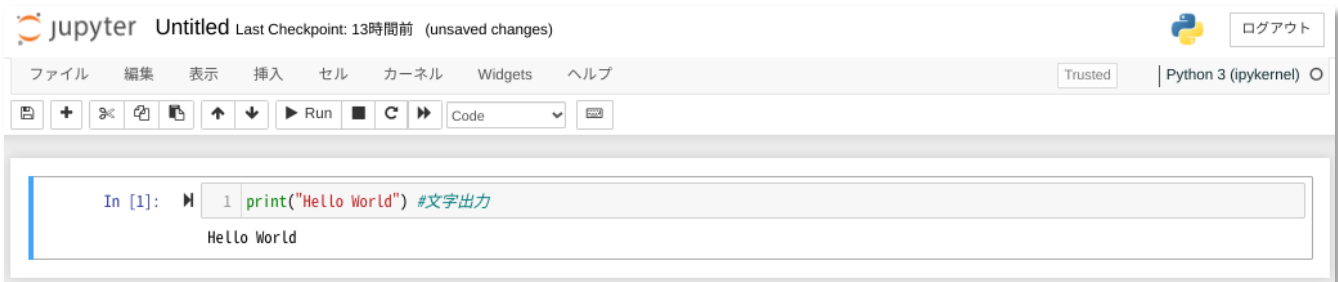
◆ プログラムの実行環境について

Python の実行

Python を実行するにあたり、授業では Chromebook にインストールした「Jupyter Notebook」を利用する。

1. Jupyter Notebook の起動

- (1) Chromebook > [アプリ一覧] > [Linux アプリ] > [Jupyter]
- (2) [MyDrive] > [Classroom] > [情報 I 2023 1 年〇組] の中の配信ファイルを開く



Jupyter Notebook の編集画面

2. ファイル名の変更

- (1) ファイル名をクリックすると変更できる。 ※授業で配信されたファイルの名前は基本的に変更しないこと

3. プログラムを書く

- (1) セルにカーソルを合わせて、セル内に命令を記述
- (2) セルの追加は、画面左上の[+]をクリック

4. プログラムの実行

- (1) [Ctrl]を押しながら[Enter]を押す。もしくは、セル左側の再生ボタン[▶]を押す

5. プログラムの保存

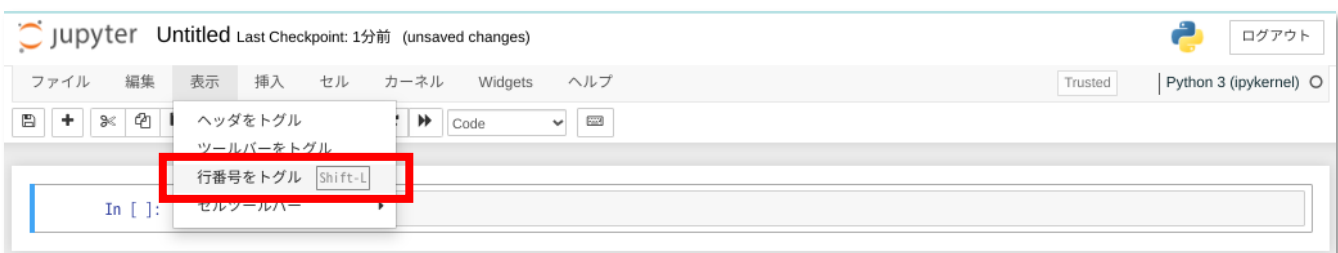
- (1) Jupyter Notebook は自動保存（5秒毎くらい）
- (2) [Ctrl]を押しながら、[S]を押すと強制保存できる

6. Jupyter Notebook の終了

- (1) 編集画面の [ファイル] > [閉じて終了] から起動中のプログラムを閉じる
- (2) ホーム画面の [終了] からアプリを閉じる

7. Jupyter Notebook の設定

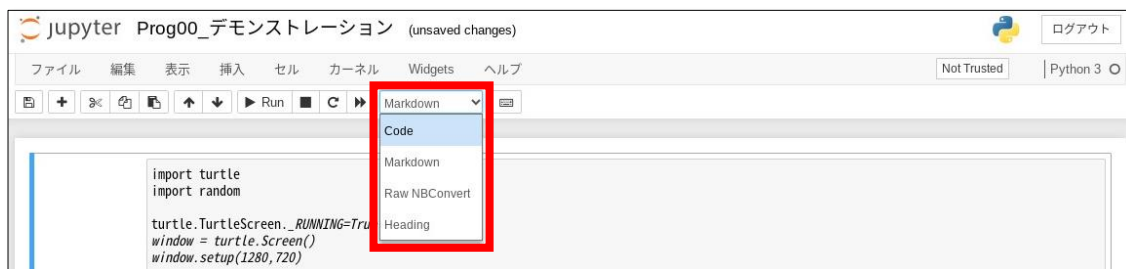
- (1) [表示]>[行番号をトグル]をクリックして、セルに行番号を表示させましょう。



8. よくあるトラブル

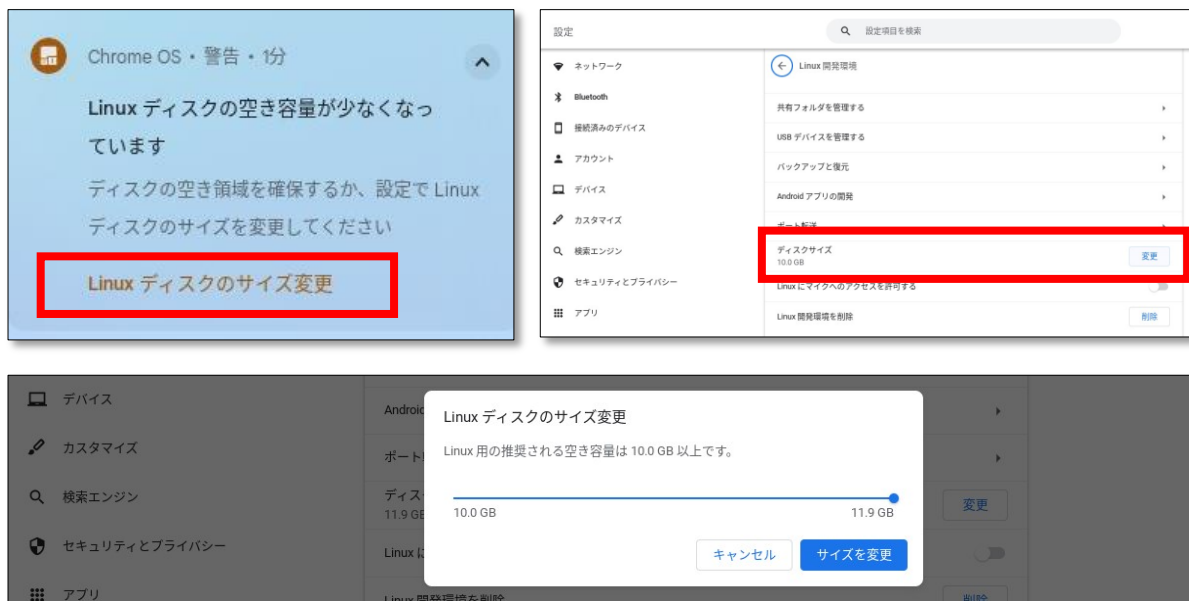
(1) アプリ起動中、セル左側の再生ボタン[▶]がなくなり、プログラムが実行できなくなる

対応方法： 画面上部のセルの種類を「Code」にしましょう。



(2) 「Linuxの空き容量が少なくなっています」と警告が表示された

対応方法： 設定からLinuxのディスクサイズを大きくしましょう



🖥️ DNCL の実行

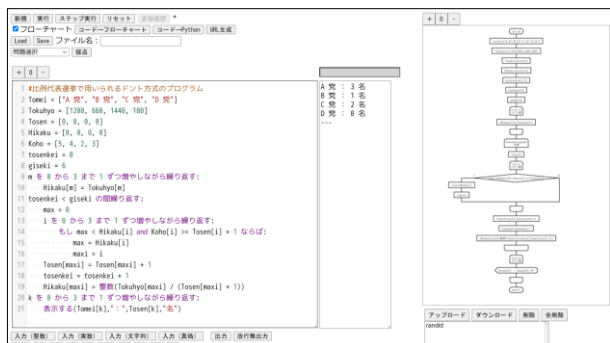
DNCL は擬似言語プログラミング言語であるため実行環境は存在しない。ただし、極めて DNCL に近い文法で日本語プログラミングができる環境として「つちのこ 2.0」および「PyPEN」が存在する。

どちらも WEB ブラウザ上で実行でき、「外部ファイルの読込」「ステップ実行」「実行途中での変数の確認」「DNCL から Python への変換」「サンプルプログラムの呼出」を提供している。共通テスト対策に役立ててほしい。

【つちのこ 2.0】

<https://t-daimon.jp/tsuchinoko/ide/>

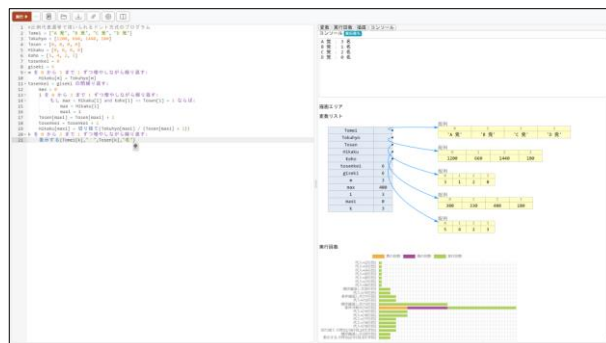
「フローチャートの自動生成」が可能



【PyPEN】

<https://watayan.net/prog/PyPEN/>

「繰り返し処理の可視化」が可能



00. プログラムとは

プログラムの構成要素は、突き詰めると次の3つに分解できます。

1. 入力（引数による値の代入など）

多くのプログラミング言語で、「値」として「数値」「文字」「配列」を処理に使用可能である。

- 数値型 : 10進数の数値データ。Pythonは「整数型」と「浮動小数点型（実数）」で扱いに差異がある。
- 文字列型 : 英字・漢字・仮名など文字データ。Pythonは「`""`（ダブルクォート）」で囲んで記述する。
- 配列構造 : 複数の値を保持する構造。Pythonは「リスト型」「タプル型」「辞書型」の3種類ある。

2. 処理（入力値の加工）

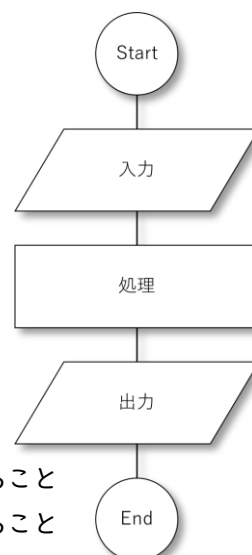
コンピュータでできる処理は、大別すると「記憶」「計算」「制御」だけである。

- 記憶 : 入力された値を「変数」としてコンピュータに記憶する。
- 計算 : 記憶した値を用いて「式」を計算する。
- 制御 : 値に応じて判断して「条件分岐」や「繰り返し」を制御する。

3. 出力（戻り値として結果を返す）

プログラムにおける出力とは、処理によって「求めたいもの」である。

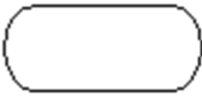
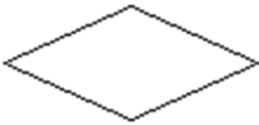







- 戻り値 : 処理によって得られた結果のことを総称して「戻り値」と呼ぶ
- 文字出力 : 「文字」や「数字」として得られた「戻り値」を画面に表示すること
- 画面描画 : 「画像」や「映像」として得られた「戻り値」を画面に表示すること
- 音声出力 : 「音声」として得られた「戻り値」をスピーカーから再生すること
- ファイル出力 : 「戻り値」を適切なファイル形式で、プログラムとは別の外部ファイルに保存すること



★フローチャート記号（JIS X 0121）

プログラムの流れを可視化するための図を「フローチャート（流れ図）」と呼び、下記の記号を使って表す。

処理の流れは、開始の「端子」から終了の「端子」に向かって、上から下、左から右に記述する決まりである。

| | | |
|------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| 端子（流れ図の開始と終了）  | 判断（条件分岐）  | 入力（標準文字入力）  |
| 処理（代入や計算など）  | ループ端【始】  | 表示（標準文字出力）  |
| 定義済処理（関数）  | ループ端【終】  | 入出力（外部データの入出力）  |

01. 標準文字出力

標準文字出力は、「コンソール」と呼ばれるウィンドウ画面に文字や数字を表示します。

Pythonでは「`print()`」という命令が用意されており、これを使うことで実行結果を画面に表示して確認します。

構文 1.1 画面に《値》を表示して改行する

| | | | |
|--------|---------------------------|------|-------------|
| Python | <code>print(《値》)</code> | DNCL | 表示する(《値》) |
|--------|---------------------------|------|-------------|

構文 1.2 画面に複数の《値》を並べて表示して改行する

| | | | |
|--------|-------------------------------------------|------|-----------------------------|
| Python | <code>print(《値1》, 《値2》, 《値3》, …)</code> | DNCL | 表示する(《値1》, 《値2》, 《値3》, …) |
|--------|-------------------------------------------|------|-----------------------------|

構文 1.3 画面に《値》を表示して、末尾に《終字》を追記する

| | | | |
|--------|-------------------------------------|------|----------------------|
| Python | <code>print(《値》, end=《終字》)</code> | DNCL | 表示する(《値》, 末尾=《終字》) |
|--------|-------------------------------------|------|----------------------|

例題 1.1

次のコードを実行して、どのような表示になるか確認しよう！

```
1 print( "Hello world" )      # 文字出力
```

```
1 print( "数字", 1, 6.02, 273 ) # 複数の値を並べて出力
```

例題 1.2

次の2つコードがそれぞれどのような表示になるか予想してから、結果を確認しよう！

| #例題 1.2a | 【あなたの予想】 | 【結果】 |
|----------------------------------------------------|----------|------|
| 1 print("大") 2 print("光") 3 print("院") | | |

| #例題 1.2b | 【あなたの予想】 | 【結果】 |
|-------------------------------------------------------------------|----------|------|
| 1 print("太", "田") 2 print("女子", end="") 3 print("高校") | | |

練習 01

自分の名前を表示するプログラムを「`print()`」を使って作ってみよう！

02. 変数

プログラミング言語における「変数」とは、「値」に名前を付ける仕組みであり、名前はその値を指し示します。

変数はアルファベットを組合せて自由に名前を設定できる。ただし、プログラミング言語に元々用意されている単語（予約語）は変数の名前（変数名）に設定できない。Python の「予約語」を以下に示す。

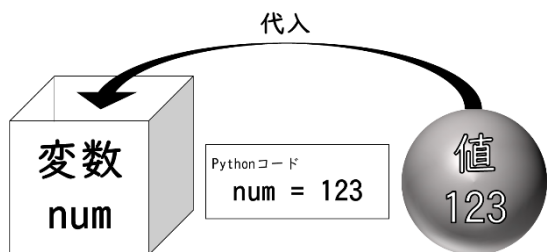
and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

構文 2.1 《変数》に《値》を代入（上書き）する

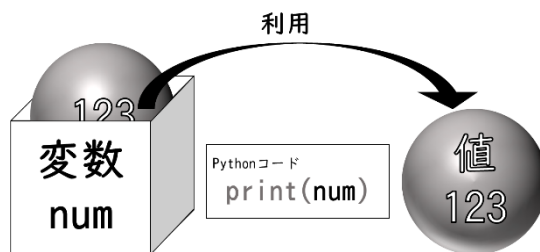
| | | | |
|--------|------------|------|------------|
| Python | 《変数》 = 《値》 | DNCL | 《変数》 = 《値》 |
|--------|------------|------|------------|

代入式（左辺に右边を上書き）

変数の利用（例：表示命令）



プログラムでの「=」（代入式）



変数名を指定すると、代入した「値」を利用できる

構文 2.2 《変数》に《値》を「実数」として代入する

| | | | |
|--------|-------------------|------|----------------|
| Python | 《変数》 = float(《値》) | DNCL | 《変数》 = 小数(《値》) |
|--------|-------------------|------|----------------|

構文 2.3 《変数》に《値》を「整数」として代入する

| | | | |
|--------|-----------------|------|----------------|
| Python | 《変数》 = int(《値》) | DNCL | 《変数》 = 整数(《値》) |
|--------|-----------------|------|----------------|

【名付けのコツ】 良い変数名・関数名は「必要な情報が含まれている」といえます。

- ☐ 広く意味を持つ名前をつけない
- ☐ 多少長くなっても必要な情報は必ず入れる
- ☐ 配列やコレクションは複数形にする
- ☐ boolean 型 (true/false) は「is」「has」「can」「should」をつけると分かりやすくなることが多い
 - ◆ is ... 状態を表す (isActive など)
 - ◆ has ... 所有を表す (hasFile など)
- ☐ 品詞を意識して、変数名なら「形容詞＋名詞」、関数名なら「動詞＋名詞」にするとわかりやすい

例題 2.1

次のコードを実行して、どのような表示になるか確認しよう！

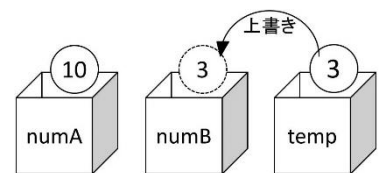
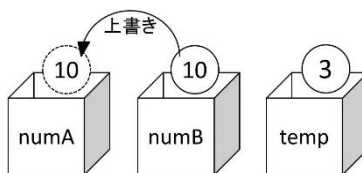
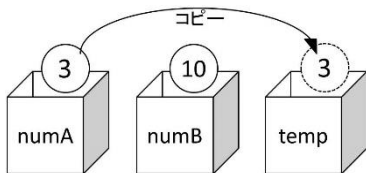
```
1 aisatsu = "こんにちは" # 変数へ文字の代入
2 print(aisatsu)          # 変数の出力
```

例題 2.2

次の2つコードがそれぞれどのような表示結果になるか予想してから、結果を確認しよう！

| #例題 2.2a | 【あなたの予想】 | 【結果】 |
|--------------------------------------------------------------|----------|------|
| 1 x = 1 2 y = 3 3 x = y 4 y = x 5 print(x , y) | | |

| #例題 2.2b | 【あなたの予想】 | 【結果】 |
|--------------------------------------------------------------------------------------------------------|----------|------|
| 1 numA = 3 2 numB = 10 3 temp = numA 4 numA = numB 5 numB = temp 6 print(numA , numB) | | |



例題 2.3

次の2つコードがそれぞれどのような表示になるか予想してから、結果を確認しよう！

| #例題 2.3a | 【あなたの予想】 | 【結果】 |
|------------------------------------------------------|----------|------|
| 1 pi = 3.141592 2 pi = int(pi) 3 print(pi) | | |

| #例題 2.3b | 【あなたの予想】 | 【結果】 |
|-------------------------------------------------|----------|------|
| 1 g = 9.80665 2 g = int(g) 3 print(g) | | |

この2つの結果より、「int()」関数は、値を [四捨五入・切り捨て・切り上げ] しているとわかる。

練習 2

変数に自分の名前を代入して表示するプログラムを組もう。利用目的を意識して変数名を決めましょう。

03. 算術計算

Python および DNCL での数式は、右辺の「計算式の結果」を左辺の「変数」に代入する文法である。

構文 3.1 《変数》に《値1》と《値2》の足し算の結果を代入する

| Python | 《変数》 = 《値1》 + 《値2》 | DNCL | 《変数》 = 《値1》 + 《値2》 |
|--------|--------------------|------|--------------------|
|--------|--------------------|------|--------------------|

| | 数学記号 | Python 記号 | 数学での書き方 | Python での書き方 |
|------------|------|-----------|------------------|--------------|
| 足し算 | + | + | $y = a + b$ | $y = a + b$ |
| 引き算 | - | - | $y = a - b$ | $y = a - b$ |
| かけ算 | × | * | $y = a \times b$ | $y = a * b$ |
| 割り算 (小数の商) | ÷ | / | $y = a \div b$ | $y = a / b$ |
| 割り算 (整数の商) | ÷ | // | | $y = a // b$ |
| あまり | mod | % | $r = a \bmod b$ | $r = a \% b$ |
| べき乗 | | ** | $y = a^b$ | $y = a ** b$ |

例題 3.1

次の5つコードの「y」がそれぞれどのような値になるか予想してから、結果を確認しよう！

| #例題 3.1a | 【あなたの予想】 | 【結果】 |
|---------------------|----------|------|
| 1 a, b, c = 2, 4, 6 | | |
| 2 y = a + b * c | | |

| #例題 3.1b | 【あなたの予想】 | 【結果】 |
|-------------|----------|------|
| 1 y = 2 | | |
| 2 y = y + 1 | | |

| #例題 3.1c | 【あなたの予想】 | 【結果】 |
|-----------------|----------|------|
| 1 a, b = 0.1, 3 | | |
| 2 y = a * b | | |

| #例題 3.1d | 【あなたの予想】 | 【結果】 |
|-----------------------|----------|------|
| 1 a, b = "123", "654" | | |
| 2 y = a + b | | |

| #例題 3.1e | 【あなたの予想】 | 【結果】 |
|-------------|----------|------|
| 1 x = "12" | | |
| 2 y = x * 4 | | |

練習 3

「 $a = 1$, $b = -5$, $c = 3$ 」のときの「 $D = b^2 - 4ac$ 」を求め、「D」を表示するプログラムを組もう！

04. 標準文字入力

これまでは完成したプログラムの変数の値を書き換えるためには、その都度ソースコードを変更する必要があった。input()関数を使うことでプログラムの実行中にキーボードからの入力の変数の値の変更が可能になり、対話的なソフトウェアの開発が可能になる。

構文 4.1 プログラム実行中に《説明文》を出して入力を受け付け、入力した文字を《変数》に代入する

Python

```
《変数》 = input( 《説明文》 )
```

DNCL

```
《変数》 = 入力( 《説明文》 )
```

構文 4.2 プログラム実行中に《説明文》を出して入力を受け付け、入力した小数を《変数》に代入する

Python

```
《変数》 = float(input( 《説明文》 ))
```

DNCL

```
《変数》 = 小数( 入力( 《説明文》 ))
```

構文 4.3 プログラム実行中に《説明文》を出して入力を受け付け、入力した整数を《変数》に代入する

Python

```
《変数》 = int(input( 《説明文》 ))
```

DNCL

```
《変数》 = 整数( 入力( 《説明文》 ))
```

例題 4.1

次のコードを数回実行して、どのような表示になるか確認しよう！

```
1 myname = input("名前を入力してね。") # 実行中に入力データを文字として変数 myname に代入する
2 print("こんにちは", myname, "さん")
```

例題 4.2

次の2つのコードについて入力値を自分で決めて、それぞれ結果がどうなるか予想してから実行して確かめよう！

| #例題 4.2a | 【あなたの予想】 | 【結果】 |
|-----------------------|--------------------|--------------------|
| 1 x = input("数字を入力。") | x = _____ と入力したとき、 | x = _____ と入力したとき、 |
| 2 y = 3 * x | y = _____ | y = _____ |
| 3 print(y) | | |

| #例題 4.2b | 【あなたの予想】 | 【結果】 |
|------------------------------|--------------------|--------------------|
| 1 x = float(input("数字を入力。")) | x = _____ と入力したとき、 | x = _____ と入力したとき、 |
| 2 y = 3 * x | y = _____ | y = _____ |
| 3 print(y) | | |

この2つの結果より、「input()」だけを使用した場合、プログラムの動作中にキーボードから入力した「値」を[整数 ・ 小数 ・ 文字]として変数に代入しているとわかる。

練習 4

プログラム実行中に「a, b, c」を入力可能にして、「 $D = b^2 - 4ac$ 」を求めて表示するプログラムを組もう！

05. 条件分岐文

条件分岐文は、「条件」が成り立つかどうかによって、実行する処理を切り替えます。

プログラムの条件式は必ず「Yes (真)」か「No (偽)」になり、「条件分岐」や「繰り返し」の制御の条件として使われる。次に示す関係演算子 (比較式) と論理演算子 (論理式) を組み合わせることで条件式を指定する。

| | | | | | |
|--------|------------|--------|--------------|---------|---------|
| A == B | A と B が等しい | A != B | A と B が等しくない | P and Q | P かつ Q |
| A <= B | A が B 以下 | A < B | A が B より小さい | P or Q | P または Q |
| A >= B | A が B 以上 | A > B | A が B より大きい | not P | P ではない |

構文 5.1

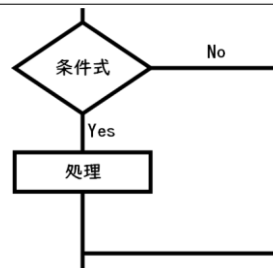
《条件式》が真の時に《処理》を実行する
(偽のときはなにもしない)

P
y
t
h
o
n

```
if 《条件》 :
    〃〃 《処理》
```

D
i
f

もし 《条件》 ならば :
〃〃 《処理》



※ 構文中の「 〃〃 」という表記は「半角スペースでの字下げ (インデント)」であるとする。

構文 5.2

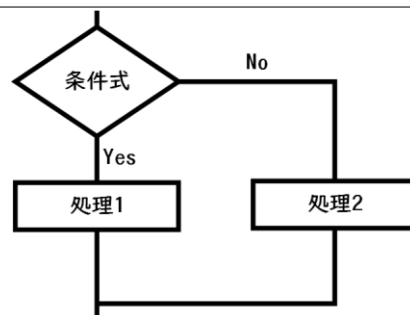
《条件式》が真のときに《処理 1》、
偽のときに《処理 2》を実行する

P
y
t
h
o
n

```
if 《条件》 :
    〃〃 《処理 1》
else :
    〃〃 《処理 2》
```

D
i
f

もし 《条件》 ならば :
〃〃 《処理 1》
そうでなければ :
〃〃 《処理 2》



構文 5.3

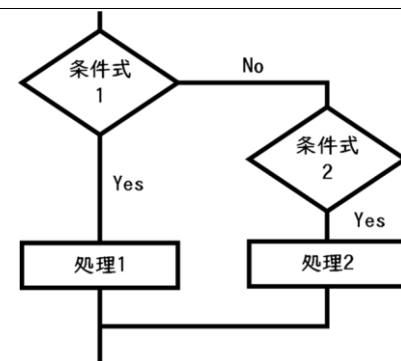
《条件式 1》が真のときに《処理 1》、
《条件式 2》が真のときに《処理 2》を実行する
(すべてが偽のときはなにもしない)

P
y
t
h
o
n

```
if 《条件 1》 :
    〃〃 《処理 1》
elif 《条件 2》 :
    〃〃 《処理 2》
```

D
i
f

もし 《条件 1》 ならば :
〃〃 《処理 1》
そうでなくもし 《条件 2》 ならば:
〃〃 《処理 2》



構文 5.4

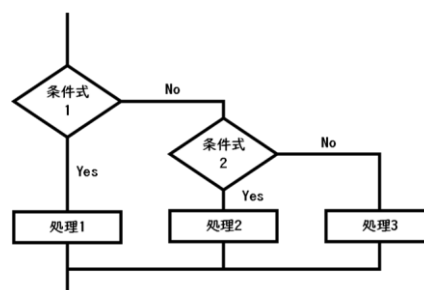
《条件式 1》が真のときに《処理 1》、《条件式 2》が真のときに《処理 2》を実行して、
すべてが偽のときに《処理 3》を実行する

P
y
t
h
o
n

```
if 《条件 1》 :
    〃〃 《処理 1》
elif 《条件 2》 :
    〃〃 《処理 2》
else :
    〃〃 《処理 3》
```

D
i
f

もし 《条件 1》 ならば :
〃〃 《処理 1》
そうでなくもし 《条件 2》 ならば:
〃〃 《処理 2》
そうでなければ
〃〃 《処理 3》



例題 5.1

次の3つコードについて、実行時に正の数・負の数・0を入力して、出力結果を記録・確認してみよう

#例題 5.1a

```

1 num = int(input("数字を入力してね。"))
2 if num > 0 :
3     print(num, "は正の数")
        
```

正: _____

負: _____

0: _____

```

graph TD
    S((S)) --> Input[入力: num]
    Input --> Decision{num > 0}
    Decision -- Yes --> Output[出力: 正]
    Output --> E((E))
    Decision -- No --> E
        
```

#例題 5.1b

```

1 num = int(input("数字を入力してね。"))
2 if num > 0 :
3     print(num, "は正の数")
4 else :
5     print(num, "は負の数")
        
```

正: _____

負: _____

0: _____

```

graph TD
    S((S)) --> Input[入力: num]
    Input --> Decision{num > 0}
    Decision -- Yes --> Output1[出力: 正]
    Decision -- No --> Output2[出力: 負]
    Output1 --> E((E))
    Output2 --> E
        
```

#例題 5.1c

```

1 num = int(input("数字を入力してね。"))
2 if num > 0 :
3     print(num, "は正の数")
4 elif num < 0 :
5     print(num, "は負の数")
6 else :
7     print(num, "は正でも負でもない")
        
```

正: _____

負: _____

0: _____

```

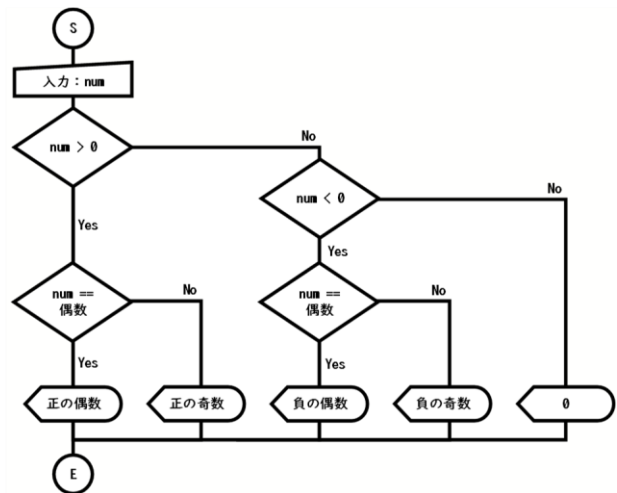
graph TD
    S((S)) --> Input[入力: num]
    Input --> Decision1{num > 0}
    Decision1 -- Yes --> Output1[出力: 正]
    Decision1 -- No --> Decision2{num < 0}
    Decision2 -- Yes --> Output2[出力: 負]
    Decision2 -- No --> Output3[出力: 0]
    Output1 --> E((E))
    Output2 --> E
    Output3 --> E
        
```

字下げのことをプログラミングでは「インデント」と呼びます。多くの言語でインデントはコードの見やすさを整えるためのものですが、PythonやDNCLではインデントによって条件分岐や繰返しの範囲を決定します。プログラミングの際にはインデントによく注意してコードを読み書きするようにしましょう。

例題 5.2

次のコードについて入力値を変えながら数回実行してみよう

```
1 num = int(input("数字を入力してね。"))
2 if num > 0 :
3     if num % 2 == 0 :
4         print(num, "は正の偶数")
5     else:
6         print(num, "は正の奇数")
7 elif num < 0 :
8     if num % 2 == 0 :
9         print(num, "は負の偶数")
10    else :
11        print(num, "は負の奇数")
12 else:
13    print(num, "は正でも負でもない")
```

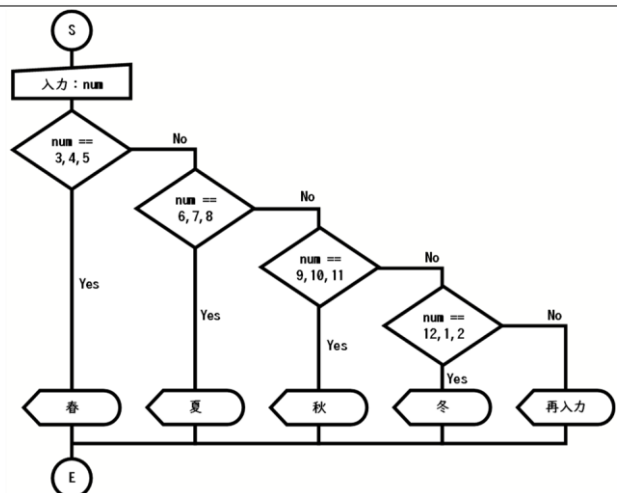


例題のように条件分岐中に条件分岐を記述できる。プログラミングでは一般にこのような入れ子構造のことを「ネスト」と呼ぶ。ネストはインデントを深めることになるため、4, 6, 9, 11 行目を 4 文字分下げしている。「条件分岐」や「繰り返し」でネストする際は、インデントの数を間違えないように注意しましょう。

例題 5.3

次のコードについて入力値を変えながら数回実行してみよう

```
1 month = int(input("誕生月を入力してね。"))
2 if 3 <= month and month <= 5 :
3     print("春生まれ")
4 elif 6 <= month and month <= 8 :
5     print("夏生まれ")
6 elif 9 <= month and month <= 11 :
7     print("秋生まれ")
8 elif month == 12 or month == 1 or month == 2 :
9     print("冬生まれ")
10 else :
11    print("1 から 12 までの整数を入力してね。")
```

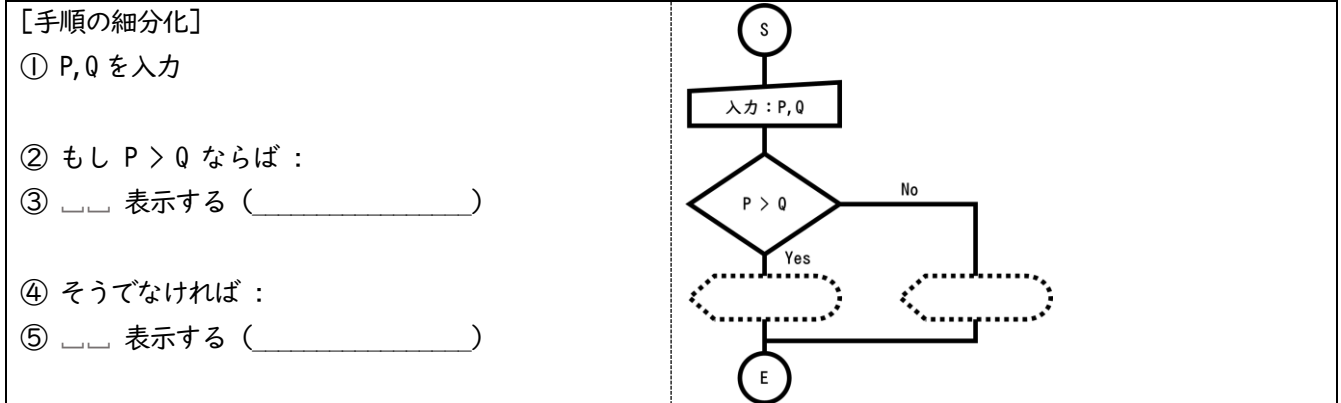


数学では $3 \leq x \leq 5$ のような、3 つ以上の値に不等号つけて比べることがある。一方、多くのプログラミング言語では値の比較は同時に 2 つまでしかできない。故に、上の例題では論理式を使って分解している。

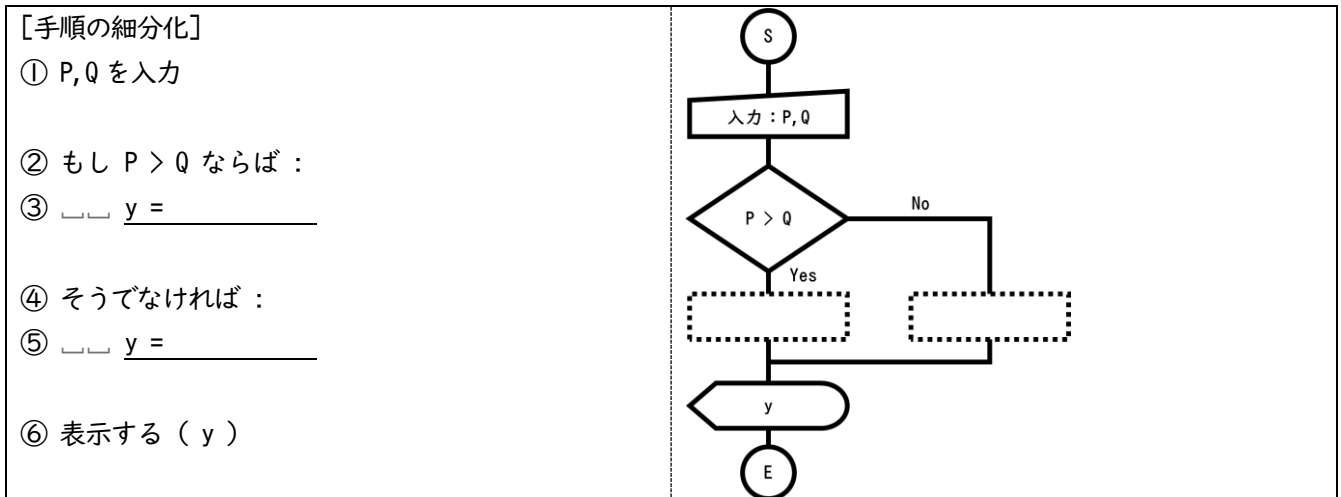
$$3 \leq num \leq 5 \rightarrow 3 \leq num \text{ かつ } num \leq 5$$

次の各プログラムについて、手順の細分化をして、フローチャートを完成させた上で作成しよう！

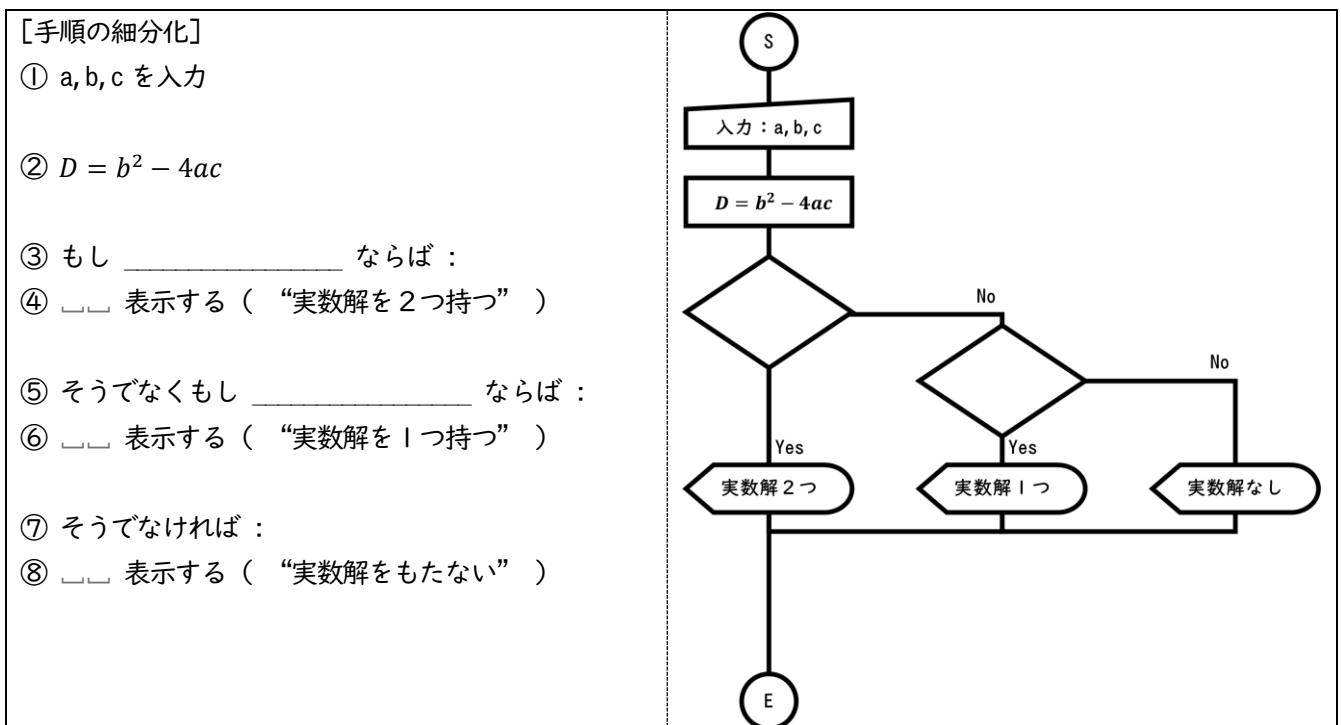
(1) 整数P,Qを入力すると、大きい方の数字を表示する



(2) 整数P,Qを入力すると、数字の差の絶対値を表示する



(3) 2次方程式 $ax^2 - bx + c = 0$ について、判別式 $D = b^2 - 4ac$ から実数解をどのように持つか表示する



06. 順次繰返し文

順次繰返し文は、「変数」の値を増やしながら、「処理」を繰返し実行します

構文 6.1 《変数》を 0 から《終値》まで1 ずつ増やしながら、《処理》を繰返し実行する

Python
for 《変数》 in range(《終値》):
 __ 《処理》

DNCL
《変数》を《終値》まで増やしながら繰返す :
 __ 《処理》

構文 6.2 《変数》を《始値》から《終値》まで1 ずつ増やしながら、《処理》を繰返し実行する

Python
for 《変数》 in range(《始値》, 《終値》):
 __ 《処理》

DNCL
《変数》を《始値》から《終値》まで増やしながら繰返す :
 __ 《処理》

構文 6.3 《変数》を《始値》から《終値》まで《増分》ずつ増やしながら、《処理》を繰返し実行する

Python
for 《変数》 in range(《始値》, 《終値》, 《増分》):
 __ 《処理》

DNCL
《変数》を《始値》から《終値》まで《増分》ずつ増やしながら繰返す :
 __ 《処理》

※ 《変数》の範囲について、Python では 《始値》以上《終値》未満、DNCL では 《始値》以上《終値》以下である

例題 6.1

次の2つのコードについて、それぞれどのような表示になるか予想してから、結果を確認しよう！

| #例題 6.1a | 【あなたの予想】 | 【結果】 |
|----------------------------------------------------|----------|------|
| 1 for i in range(5): 2 __ print(i, end=",") | | |

ヒント (6.1a): print()命令の「end オプション」については、p05の構文 1.2を確認しよう。

| #例題 6.1b | 【あなたの予想】 | 【結果】 |
|-------------------------------------------------------------------------------|----------|------|
| 1 for i in range(1 , 5): 2 __ x = 2 * i - 1 3 __ print(x , end=",") | | |

ヒント (6.1b): 変数 i の値が1以上5未満の範囲で1ずつ増えながら、2行目と3行目の処理を実行している。

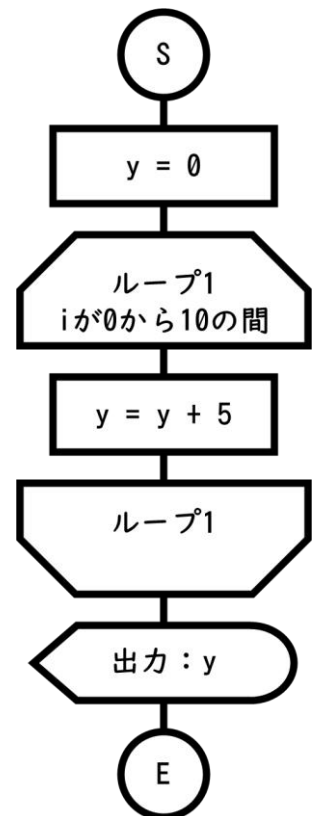
例題 6.2

次の2つのコードについて、ループを繰り返すごとに処理内の「変数の値」と「右辺の式」がどのように変化しているか、表を完成させよう。

#例題 6.2a

```
1 y = 0
2 for i in range( 11 ):
3     y = y + 5
4 print( "y =", y )
```

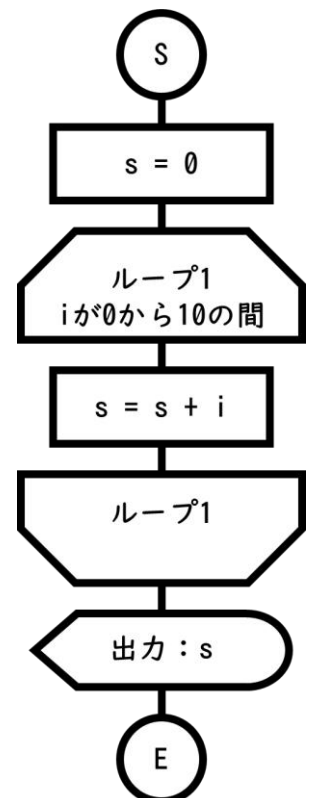
| i | y | = y + 5 |
|----|----|----------|
| 0 | 5 | = 0 + 5 |
| 1 | 10 | = 5 + 5 |
| 2 | 15 | = 10 + 5 |
| 3 | | = 15 + 5 |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |



#例題 6.2b

```
1 s = 0
2 for i in range( 11 ):
3     s = s + i
4 print( s )
```

| i | s | = s + i |
|----|---|---------|
| 0 | 0 | = 0 + 0 |
| 1 | 1 | = 0 + 1 |
| 2 | 3 | = 1 + 2 |
| 3 | | = 3 + 3 |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |



プログラムの「=」は等式ではなく、代入式（上書き）であることを再確認しよう！

例題 6.3

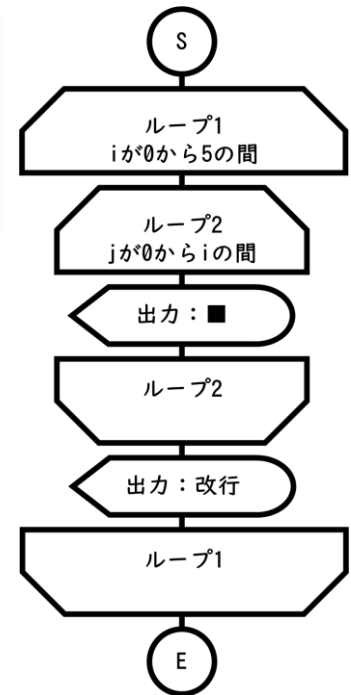
次のコードを読み、i 行 j 列の表をコードの通り塗りつぶしましょう。(■が塗る、□が空白)

#例題 6.3a

```
1 for i in range(6):
2     for j in range(i):      # j ループの範囲に i を利用
3         print("■",end="")  # 改行せずに■を表示
4     print("")              # 改行
```

| 予想 | | j | | | | | |
|----|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| i | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| | 4 | | | | | | |
| | 5 | | | | | | |

| 結果 | | j | | | | | |
|----|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| i | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| | 4 | | | | | | |
| | 5 | | | | | | |



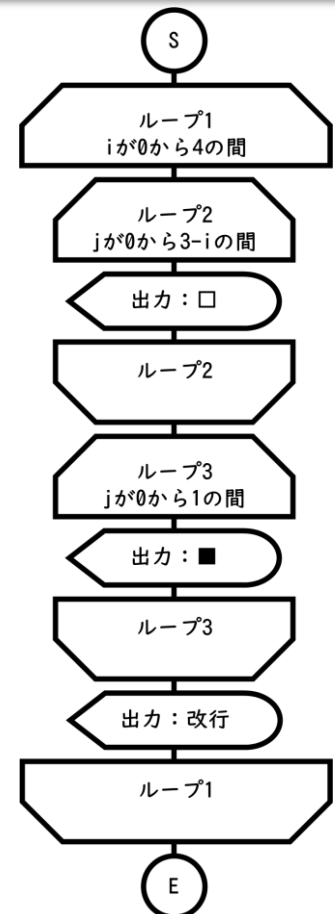
フローチャートを指でなぞりながら、各ループの変数がどのように変化しているか確認しましょう。

#例題 6.3b

```
1 for i in range(5):
2     for j in range(4-i):
3         print("□",end="")  # 改行せずに□ (空白) を表示
4     for j in range(2):
5         print("■",end="")  # 改行せずに■を表示
6     print("")              # 改行
```

| 予想 | | j | | | | | |
|----|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| i | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| | 4 | | | | | | |
| | 5 | | | | | | |

| 結果 | | j | | | | | |
|----|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| i | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| | 4 | | | | | | |
| | 5 | | | | | | |



多重ループにすることで、外側のループの値に応じて内側のループの指定範囲を変動させることができる。これを「可変長ループ」と呼び、「データの並び替え」などのアルゴリズムでよく使うテクニックである。

練習 6

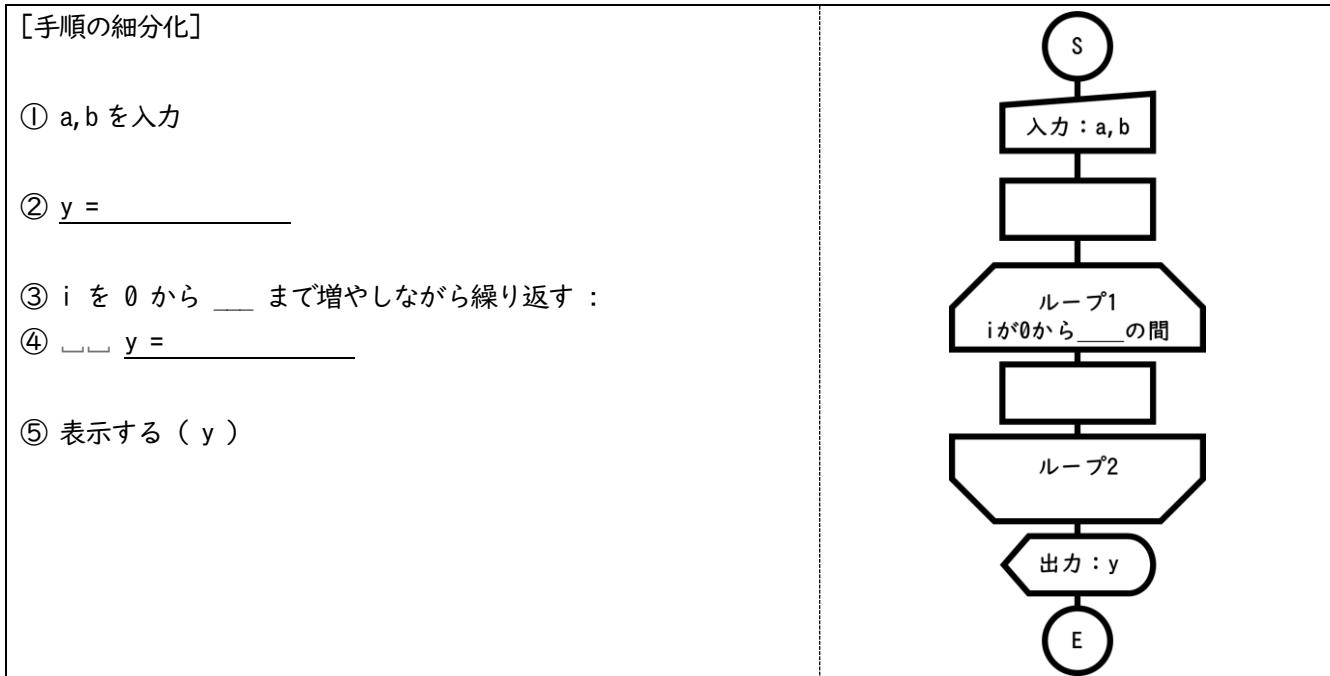
次の各プログラムについて、手順の細分化をして、フローチャートを完成させた上で作成しよう！

- (1) 自然数 a と b を入力して、繰り返し構文を使って $y = a^b$ を計算するプログラム (**を使った演算を禁ずる)

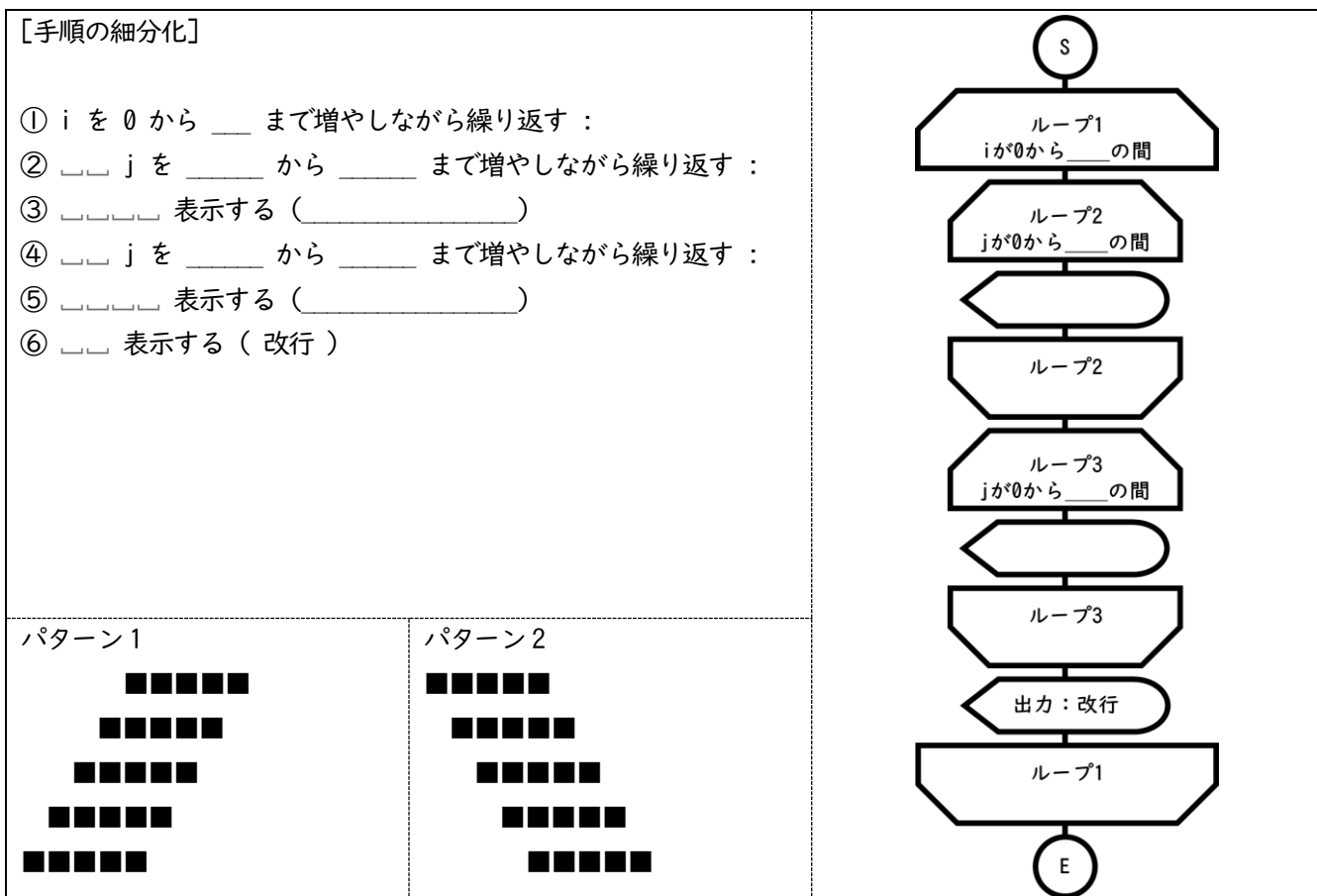
$$y = a^b$$

$$= 1 \times a \times a \times a \cdots \times a$$

※ a を b 回掛け合わせている



- (2) 5 マス 5 段の平行四辺形を表示するプログラム (パターン 1, 2 のどちらかならば OK)



ヒント (練 6.2) : 例題 6.3b をベースに書き換えると簡単に作れる

07. 条件繰返し文

条件繰返し文は「条件」が成り立つ間、「処理」を繰返し実行します。なお、「処理」を実行する前に「条件」が成り立つかどうか判定されるため、「処理」が 1 回も実行されないことがあります。

構文 7.1 《条件式》が真である間、《処理》を繰返し実行する

Python

```
while 《条件》 :  
    __ 《処理》
```

D N C L

```
《条件》の間繰繰り返す :  
    __ 《処理》
```

例題 7.1

次のコードについて、(a)は飽きるまで「Y」と入力、(b)は好きに入力して飽きたら「Y」と入力しよう。

```
#例題 7.1a  
1 answer, num = "Y", 1  
2 while answer == "Y":  
3     __ print( num , end=" : " )  
4     __ answer = input("数字を2倍しますか? (Y/N) :")  
5     __ num = num * 2
```

```
#例題 7.1b  
1 answer = "N"  
2 while answer != "Y":  
3     __ answer = input("勇者よ、世界をお救いください! (Y/N) :")  
4     __ print("ありがとう!")
```

例題 7.1a は繰返す処理への「継続条件」を満たすか入力値を判断していたのに対して、例題 7.1b は繰返す処理への「終了条件」を満たすか入力値を判断している。While に渡す条件式は継続条件の必要があるため、`answer != "Yes"` という「≠」を用いることで終了条件を継続条件にしている。

例題 7.2

次のコードについて、何回か実行して異なる桁数の数字を入力して、結果を確認してみよう！

```
1 num = int(input("各桁に分解したい整数を入力してね。"))  
2 while num > 0 :  
3     __ x = num % 10  
4     __ num = num // 10  
5     __ print( x , end="," )
```

例題 7.2 は入力値によって桁数が異なるため、for のように回数を指定して処理を繰返すことができない。このようにプログラムの開始時点で繰返す回数が不明な事象に対して while は有用である。

練習 7

例題 7.2 をベースに、入力した数値の各桁の数の総和を求めるプログラムを作成しよう！

08. 関数

これまで、出力命令には `print()`、入力命令には `input()` など既にいくつかの関数を扱ってきた。これらのように再利用性の高い命令は関数として最初から設定されている。そして、関数は「関数定義」により自作できる。関数を作ると機能単位で処理を分けることが容易になりコード全体が読みやすくなる。

構文 8.1 関数定義の記述方法

Python

```
def 《関数名》(《引数1》, 《引数2》, ~):  
    __ 《処理》  
    __ return 《戻り値》
```

D N C L

関数《関数名》(《引数1》, 《引数2》, ~):
 __ 《処理》
 __ 《戻り値》を返す

引数（ひきすう）は、関数が受け取る値を指し示す変数のことです。

「:」以降は、関数定義の本体であり、関数の処理を記述する部分として「return」へと続きます。
return は、続く式の評価結果「戻り値」を、関数の呼び出し元に返して、関数を終了します。

構文 8.2 《数値》を2進数で表示する

Python

```
bin(《数値》)
```

D N C L

二進で表示する(《数値》)

構文 8.3 《最小値》以上《最大値》以下のランダムな整数を生成して《変数》に代入する

Python

```
import random  
《変数》= random.randint(《最小値》, 《最大値》)
```

D N C L

《変数》= 整数乱数(《最小値》, 《最大値》)

構文 8.4 《最小値》以上《最大値》以下のランダムな実数を生成して《変数》に代入する

Python

```
import random  
《変数》= random.uniform(《最小値》, 《最大値》)
```

D N C L

《変数》= 実数乱数(《最小値》, 《最大値》)

乱数を生成する関数はPython標準の命令ではなくモジュールという拡張機能に含まれている。「import」を使うことで作成済みの関数を呼び出すことができ、プログラムの幅が広がっていく。

例題 8.1

次のコードを何回か実行して結果を確認してみよう！

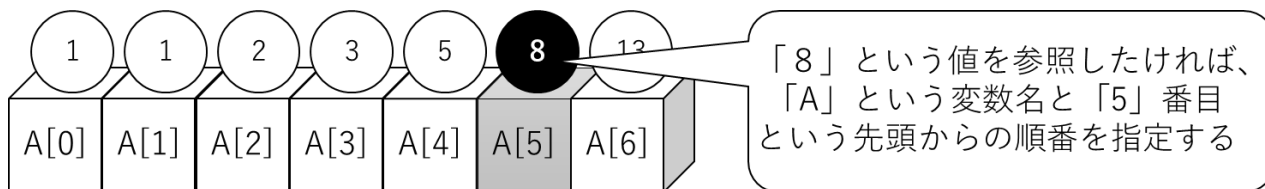
```
1 import random          # モジュールの読み込み  
2 r = random.randint(1,100) # random モジュール内の randint() 関数を呼び出している  
3 print(r)
```

練習 8

1 以上 1000 以下の奇数をランダムに表示するプログラムを組んでみよう！

09. 配列構造

大量のデータを取り扱う際に、すべての変数に名前を付けることは実質不可能であるため、「リスト」と呼ばれる構造の変数を使う。これは、1つの変数名で複数の値を保存でき、使用する際は先頭から何番目のデータかを番号で指定して呼び出す。この考え方はPython以外にも多くのプログラム言語に存在する。



構文 9.1 《配列》に複数の《値》を代入する

| | | | |
|--------|--------------------------------------------|---------|--------------------------------------------|
| Python | <code>《配列》 = [《値1》, 《値2》, 《値3》 ~]</code> | D N C L | <code>《配列》 = [《値1》, 《値2》, 《値3》 ~]</code> |
|--------|--------------------------------------------|---------|--------------------------------------------|

構文 9.2 指定した《位置》の《配列》の値を《変数》に代入する

| | | | |
|--------|----------------------------------|---------|----------------------------------|
| Python | <code>《変数》 = 《配列》[《位置》]</code> | D N C L | <code>《変数》 = 《配列》[《位置》]</code> |
|--------|----------------------------------|---------|----------------------------------|

構文 9.3 《配列》に含まれる値の個数を《変数》に代入する

| | | | |
|--------|---------------------------------|---------|---------------------------------|
| Python | <code>《変数》 = len(《配列》)</code> | D N C L | <code>《変数》 = 要素数(《配列》)</code> |
|--------|---------------------------------|---------|---------------------------------|

構文 9.4 《配列》の値を順番に《変数》に代入しながら処理を実行する

| | | | |
|--------|-----------------------------------------------|---------|------------------------------------------------|
| Python | <pre>for 《変数》 in 《配列》 : __ 《処理》</pre> | D N C L | <code>《配列》の要素《変数》について繰り返す : __ 《処理》</code> |
|--------|-----------------------------------------------|---------|------------------------------------------------|

構文 9.5 《配列》から指定した位置の値を削除する

| | | | |
|--------|-------------------------------|---------|------------------------------|
| Python | <code>del 《配列》[《位置》]</code> | D N C L | <code>削除 《配列》[《位置》]</code> |
|--------|-------------------------------|---------|------------------------------|

構文 9.6 《配列》の指定した《位置》へ《値》を挿入する

| | |
|--------|---------------------------------------|
| Python | <code>《配列》.insert(《位置》, 《値》)</code> |
|--------|---------------------------------------|

構文 9.7 《配列》の末尾へ《値》を追加する

| | |
|--------|---------------------------------|
| Python | <code>《配列》.append(《値》)</code> |
|--------|---------------------------------|

構文 9.8 《配列 A》の末尾へ《配列 B》を連結する

| | |
|--------|--------------------------------------|
| Python | <code>《配列 A》.extend(《配列 B》)</code> |
|--------|--------------------------------------|

例題 9.1

次のコードについて、上から順番に実行していくとどのように表示されるか予想してから、結果を確認しよう！

| | #例題 9.1 | 【あなたの予想】 | 【結果】 |
|---|---------------------------------------------------------------------------------------|----------|------|
| a | <code>a = [1, 1, 2, 3, 5, 8, 13, 21, 34]</code> <code>print(a)</code> | | |
| b | <code>print(a[6])</code> | | |
| c | <code>print(len(a))</code> | | |
| d | <code>for i in range(len(a)) :</code> <code> print(a[i] , end=" ")</code> | | |
| e | <code>a.append(55)</code> <code>print(a)</code> | | |
| f | <code>a[7] = 1000</code> <code>print(a)</code> | | |

例題 9.2

次のコードについて、何回か実行して、結果を確認しよう！

```

1 import random
2 r = []                                # 空の配列を生成
3 for i in range(10):
4     r.append( random.randint(1,100) ) # 配列の末尾に乱数を追記
5 print(r)

```

練習 9

次の各プログラムについて、手順の細分化をして、フローチャートを完成させた上で作成しよう！

(1) 要素数 10 個の乱数配列を生成・表示して、リストの中から最大値を見つけ出すプログラム

| 【手順の細分化】 | 【フローチャート】 |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|
| ① 要素数 10 個の乱数配列 r を生成する ② 表示する (r) ③ _____ ④ i を _____ から _____ まで増やしながら繰り返す : ⑤ ____ もし _____ ならば : ⑥ ____ ____ _____ ⑦ 表示する (_____) | |

(2) 要素数 10 個の乱数配列を生成・表示して、リストを昇順に並び替えるプログラム

| 【手順の細分化】 | 【フローチャート】 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|
| ① 要素数 10 個の乱数配列 r を生成する ② 表示する (r) ③ i を _____ から _____ まで増やしながら繰り返す : ④ ____ j を _____ から _____ まで増やしながら繰り返す : ⑤ ____ ____ もし _____ ならば : ⑥ ____ ____ ____ 入れ替える (_____ , _____) ⑦ 表示する (r) | |

※乱数配列の各要素の値は 1 以上 100 以下とする。

◆ エラーについて

プログラムは曖昧な表現を避けるため「正確」に記述しないと動きません。

【構文エラー】

```
In [1]: 1 123+
        Input In [1]
        123+
        ^
SyntaxError: invalid syntax
```

エラー箇所は、矢印や色で強調される。
強調箇所の周辺を含めてよく読みましょう！

コードは間違った文法で実行しても警告が出て動きません。これを「**構文エラー**」と呼びます。
Jupyter Notebook をはじめとする多くのプログラミングの実行環境ではエラー箇所を教えてください。
エラーが表示されたら内容をよく読んで修正しましょう。

【実行時エラー】

```
In [1]: 1 123/0
ZeroDivisionError                                Traceback (most recent call last)
Input In [1], in <cell line: 1>()
----> 1 123/0
ZeroDivisionError: division by zero
```

プログラムの文法は正しいが内容の不備を警告するエラーもあり、「**実行時エラー**」と呼びます。例えば、計算の処理中に 0 で割り算を行うと実行時エラーが発生します。文法だけではなく論理的に正しいのかも考えるようにしましょう！試行錯誤することがプログラミングの醍醐味です。

【よくあるエラーメッセージの一覧】

| エラーメッセージ | 意味 |
|------------------------------------------------------------|-----------------------|
| <i>invalid syntax</i> | 文法の間違い |
| <i>invalid character in identifier</i> | 無効な文字（全角文字など）が存在する |
| <i>expected an indented block</i> | インデント（段落）が必要 |
| <i>unindent does not match any outer indentation level</i> | インデントが揃っていない |
| <i>unexpected indented</i> | インデントが不要 |
| <i>unexpected EOF while parsing</i> | プログラムが書きかけ途中 |
| <i>name 'XXXX' is not defined</i> | 'XXXX' は定義されていない |
| <i>expected 1 arguments, got 0</i> | 関数の引数が足りない |
| <i>ValueError</i> | 関数の引数に不適切な値を入れている |
| <i>TypeError</i> | 数字を文字で割り算してしまう等の型の不一致 |
| <i>ModuleNotFoundError</i> | モジュールの import ができていない |
| <i>Out of range</i> | リストの範囲外を見ようとしてエラー |

※他にもエラーは沢山あるので見つけたら調べて自分で解決する習慣をつけましょう。