2-7　プログラミング基礎

東京大学 数理・情報教育研究センター

2021年４月３０日

2025年9月1日改訂

概要

- 本節ではコンピュータにさまざまな処理を実行させるプログラムの作成に必要な基礎知識を学びます。
- 実際にプログラムを作成し、実行させないと理解は難しいと思われます。本教材のスライドには、東京大学・数理情報教育センターが提供している「Python プログラミング入門」のリンクを記載しています。必要に応じて活用してください。

本教材の目次 1/2

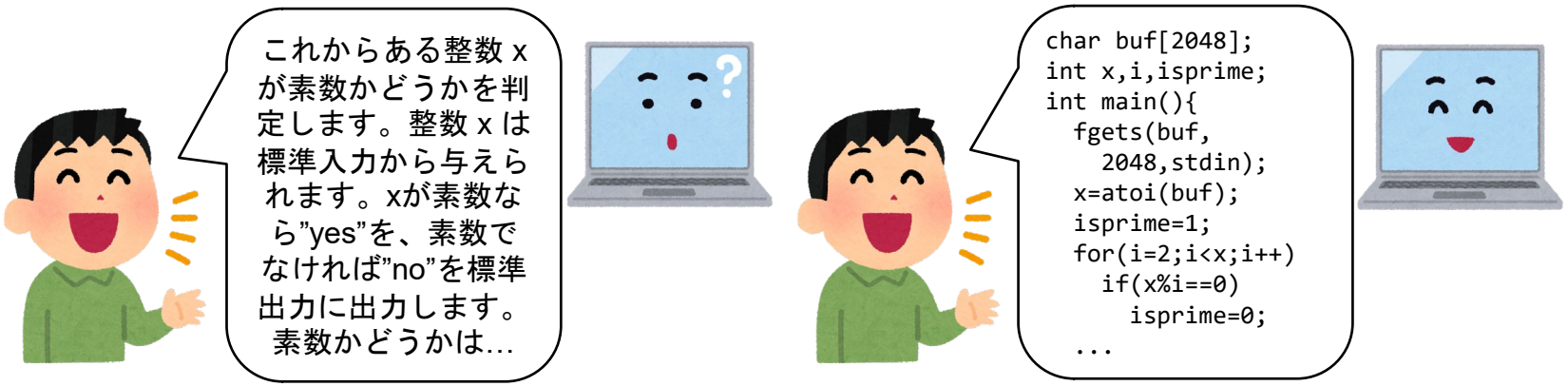
本教材の目次 2/2

プログラムとプログラミング

- コンピュータはプログラムを実行する装置です。プログラムを作成することでコンピュータに計算の指示を出すことができます。

- プログラムを作成することをプログラミングと呼びます。プログラムを作成する人をプログラマと呼びます。プログラムのことをコード、プログラミングのことをコーディングと呼ぶこともあります。

- プログ ラミングに使用する言語をプログラミング言語と呼びます。C、Java、Python、JavaScript、Rust など、現在では様々なプログラミング言語を使用することができます。

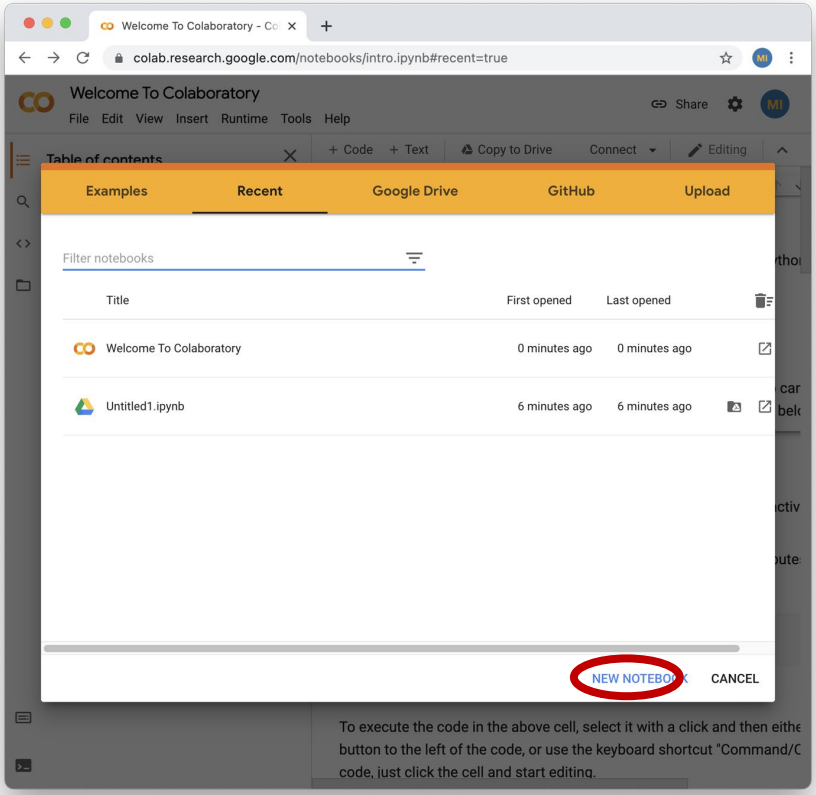


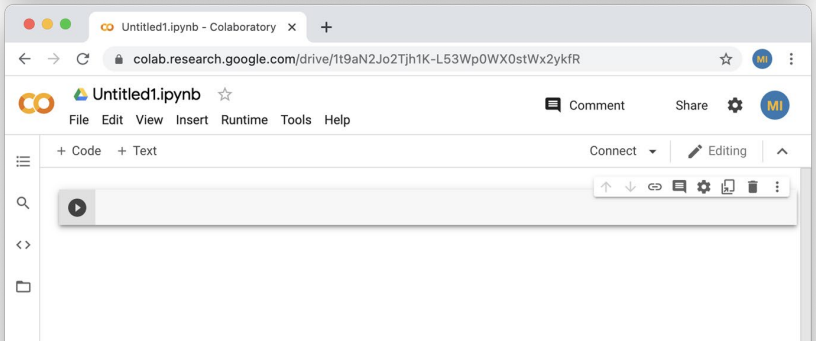
プログラミング言語 Python

- 本節では、Python を例として、プログラミングを学びます。
- Python はシンプルで読みやすい構文でありながら高度な処理を行うプログラムの作成にも適した言語です。特に統計処理や機械学習のライブラリが数多く配布されています。
- Python はオープンソースソフトウェアとして公開されており、誰でも使うことができます。

Python を使うために：Google Colaboratory

- Google の Colaboratory サービスでは、Python プログラミング環境が提供されています。Colaboratory は普段使っている Web ブラウザから利用することができます。
- 利用には Google アカウントが必要です。アカウントは個人が Gmail 用に利用しているもの、大学から Google Workspace 用に提供されているものいずれでもかまいません。
- 準備ができれば Web ブラウザで以下の URL にアクセスしてください。
  https://colab.research.google.com/
- 右上が表示されれば、画面下の NEW NOTEBOOK をクリックしてください。
- 右下の画面が表示されることを確認してください。





https://utokyo-ipp.github.io/1/1-0.html

電卓として使う

- Colaboratory 画面はいくつかのセルから構成されています。
- コードセル（灰色部分）にはプログラムを記述することができます。
- コードセルに 1+1 と入力し、セル実行ボタンをクリックすると、計算結果、2 が表示されます。
- 四則演算の加減乗除の演算子は、それぞれ +-*/を利用します。（通常プログラミング言語では除算は ➗ ではなく / が使われます。）その他、切り捨て除算(//)、余り(%)、冪乗(**)も利用できます。
- Python では整数と小数点のある数（実数）は数学的に同じ数を表す場合でも異なる形式で扱われますので表示は異なります。（整数は整数型、実数は浮動少数点型として扱われます。）
- コードセル追加ボタンで新しいコードセルを追加することができます。
  - メモなどを自由に記述できるマークダウンセルも用意されています。
- セルの実行が止まらない、すなわちセル実行ボタンが回転しつづけるときは、再度セル実行ボタンをクリックすると、プログラムを強制終了することができます。

https://utokyo-ipp.github.io/1/1-1.html

変数

- 変数とは値に名前を割り当てる仕組みす。
- 右上の例では等号 = を用いて、188.0 という値に h という名前をつけています。これを変数定義、あるいは代入と呼びます。
- 定義された変数はその後の式で利用することができます。右中の例では、先に定義した変数 h を評価しています。一方で、定義されていない変数、ここでは j を評価するとエラーとなります。
- 代入は = の右辺を評価した結果を左辺の変数に割り当てます。ここで、左辺の変数が右辺にあってもかまいません。
- 変数の値を増加（減少）させるといった単純な処理には累積代入演算子 += (-=)を使うと、プログラムが読みやすくなります。（右下）

- 右の例ではセルの # 以降にコメントを記載しています。
  Python では # 以降の内容は評価されないため自由にコメントを書くことができます

```
[22] h = 188.0          # 変数 h を定義

[23] h                  # h を評価（下に表示されます）
     188.0

[24] j                  # 未定義の変数 j を評価してみる。エラーになります。
     ---------------------------------------------------------------------------
     NameError                                 Traceback (most recent call last)
     <ipython-input-24-60ca8127080f> in <module>()
     ----> 1 j               # 未定義の変数 j を評価してみる。

     NameError: name 'j' is not defined

     SEARCH STACK OVERFLOW

[25] h = h + 200        # h に 200 を加えた結果を変数 h に代入する。

[26] h
     388.0

[27] h += 200           #累積代入演算子をでさらに 200 を加える。

[28] h
     588.0
```

https://utokyo-ipp.github.io/1/1-2.html

プログラム

- 前項では一行の処理、プログラム文、ごとに実行結果を確認していました。プログラム文を並べて記述することができます。コンピュータはプログラム文の処理を順番に実行します。
- 例えば、肥満度(BMI)は右上の式から得ることができます。
- 右中のプログラムでは最初に分母の評価結果を変数 h2 に代入し、次に除算で bmi を評価する流れです。
- 右下のプログラムは右上と同じですが、行番号を表示させています。
  - Colaboratory の場合行番号は以下のキー操作で表示/非表示を切り替えることができます。 CTL-M L (Control キーを押しながら、M、Lを続けて押します。)
- 以降の説明では行番号を表示させておこないます。

$$BMI = \frac{体重(Kg)}{身長^2(m)}$$

```
h = 180.0
w = 75.0
h2 = (h/100.0)**2
w / h2
```

```
23.148148148148145
```

```
1 h = 180.0
2 w = 75.0
3 h2 = (h/100.0)**2
4 w / h2
```

```
23.148148148148145
```

https://utokyo-ipp.github.io/1/1-2.html

関数

- 身長・体重データは人によって違いますが、BMI は同じ処理で求めることができます。関数を利用すると、異なるデータに対して同じ処理をおこなうことができます。
- 右の例では、2番目のセルで bmi2 という関数を定義しています。関数の定義は以下のように行います。

  def 関数名(引数, ...):

- () 内の引数は関数が受け取る値に割り当てる変数で、関数の中の処理で使用できます。
- 以降の行で関数の処理が記述され 3 行目の return 文が続きます。return 文は以下のように使用します。

  return 式

- 関数は return 文では式を評価し、終了します。式の評価結果は呼び出し元に返されます。これを返値（戻り値）と呼びます。
- 関数定義文以降では字下げ（インデント）されています。Python では関数がおこなう処理は字下げを続けて記述していきます。
- ３番目のセルでは関数 bmi2 を使って、BMI を評価しています。１番目のセルと同じ結果を得ています。

https://utokyo-ipp.github.io/1/1-2.html
https://utokyo-ipp.github.io/3/3-3.html

論理・比較演算と条件分岐の基礎

- 条件分岐(if)文の基本的な形は以下のように記述します。

```
if 式:
  処理1
  …
else:
  処理2
  …
```

- 右上は関数 bmax は 2 つの引数 a, b の大きい方を返す関数の例です。
- ここで a > b のような式を比較演算と呼びます。比較演算子の結果は 論理値 （bool）型で結果が真の場合 True, 偽の場合 False を返します。式が True の場合 if 文直後の「処理1」が、False の場合 else 文の後の「処理2」が実行されます。
- また、if および else 以降の行が関数定義よりももう一段深くインデントされていることに注意してください。

```
def bmax(a, b):
    if a > b:
        return a    # a > b が True の場合
    else:
        return b    # a > b が False の場合
```

```
bmax(10, 20)
```

```
20
```

- 変数 a,b,c の関係が a < b < c の関係にあることを評価したときは、論理演算子 and で以下のように書けます。

```
if a < b and b < c:
```

- ここで比較演算子 < は論理演算子 and よりも優先度が高く、先に評価されることに注意してください。プログラミングでは演算子の優先順位に注意する必要があります。
- 論理・比較演算子の一覧と意味を右下に示します。

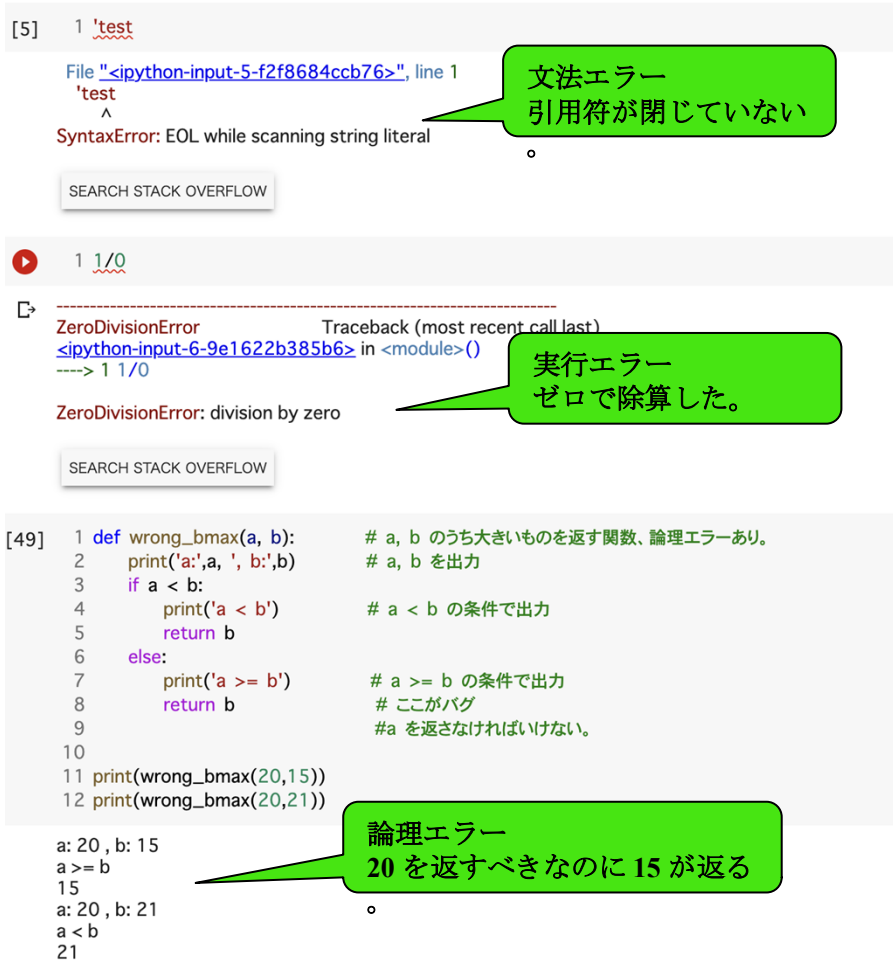
```
[35] a == b    # a, b が等しい
     a != b    # a, b が等しくない
     a <= b    # a は b 以下
     a < b     # a は b より小さい
     a >= b    # a は b 以上
     a > b     # a は b より大きい

     x and y   # x, y の論理積
     x or y    # x, y の論理和
     not x     # x の否定
```

https://utokyo-ipp.github.io/1/1-3.html

デバッグ

- プログラムに誤り（バグ）があると想定どうりに動作しないことがあります(エラー)。こういったときは、バグを取り除く作業、デバッグ、が必要になります。
- エラーには、文法エラー、実行エラー、論理エラーがあり、それぞれ対応が異なります。
- 文法エラー(Syntax Error)はプログラムの文法上の誤りが原因で、エラー出力にも SyntaxError と表示されます。エラー出力に含まれる行番号からエラー箇所を把握できます。
- 実行エラー(Runtime Error)は文法上に誤りはないものの、プログラムの処理の継続ができないことが原因です。文法エラー同様にエラーメッセージからエラーが発生箇所は把握できますが、エラー以前の処理における論理エラーの可能性もあります。例えば、a / b のような除算で、他の場所で b にゼロを代入する場合です。
- 論理エラー(Logical Error)はプログラムは正常終了するものの意図したように動作しないというエラーです。この場合、プログラムをよく見直し、エラー箇所を発見し、修正する必要があります。
- 論理エラーの発見には、処理の流れや、変数の変化を、print 文によって表示させる方法も有効です。



https://utokyo-ipp.github.io/1/1-4.html

文字列

- 文字列を使ったプログラミングについて学びます。
- Pythonでは文字列は引用符（シングルクオート）’あるいは二重引用符（ダブルクオート）”で囲んで記述します。
- 文字列には数字の並びも含まれます。これは、これまで学んだ数とは異なることに注意してください。
- 文字列の位置をインデックスで指定して文字や、スライスで指定して一部の文字列を取り出すこともできます。
- 長さゼロの文字列、すなわち空文字列、を作ることもできます。
- 二項演算子で文字列の連結(+)や繰り返し(*) が行えます。また、論理演算子 in は右辺の文字列に左辺の部分文字列が存在すると True を返します。
- その他、文字列を操作するためのメソッド（関数のようなもの）も用意されています。

```
[86]  1 'hello'            # 引用符で文字列を作る。
      'hello'

[87]  1 word1 = 'hello' # 文字列を作り、word1 に代入
      2 word1
      'hello'

[88]  1 word1[2]          #3 文字目をインデックス指定で取り出す。最初の文字のインデックスは
      'l'

[89]  1 word1[2:5]        #3-6 文字目の部分文字列をスライスで取り出す。
      'llo'

      1 'he' in word1          # 文字列 'he' が含まれていることをテスト。
      True

[92]  1 'The' in word1
      False
```

https://utokyo-ipp.github.io/2/2-1.html

リスト・タプル

- 複数の要素をまとめて取り扱うデータ構造としてリスト(list)があります。
- 他のプログラミング言語では配列と呼ばれることもあります

- リストを作るには要素をコンマで区切り、全体をかぎ括弧 [...]で囲みます。要素を持たない空リストを作ることもできます。空リストは繰り返し処理で便利です。
- 文字列と同じように、インデックス・スライスによる要素・部分リストの取り出しができます。
- インデックスを指定してリストの要素を更新することができます。
- リストを要素として持つリスト（多重リスト）を作ることもできます。

- リストに似たとデータ構造として、タプル(tuple)があります。タプルは要素を丸括弧 (...) で囲んで作ります。
- リスト同様に、インデックス・スライスが利用できます。
- リストと違って、タプルの要素を更新することはできません。

```
[118]  1 abcd = ['a', 'b', 'c', 'd']          # 一文字 4 要素のリスト
       2 abcd[2]                              # インデックスで 3 要素目を取り出す

       'c'

[120]  1 abcd[2] = 'hello'                    #3番目の要素を更新する
       2 abcd

       ['a', 'b', 'hello', 'd']

[122]  1 abcd = ['a', 'b', 'c', 'd']          # スライスを使った範囲指定
       2 abcd[1:3]

       ['b', 'c']

[124]  1 abcd2 = ('a', 'b', 'c', 'd')          # 一文字 4 要素のタプル
       2 abcd2[2]                             # インデックスで 3 要素目を取り出す

       'c'

[125]  1 abcd2[1:3]                           # スライスを使った範囲指定

       ('b', 'c')

       1 abcd2[2] = 'hello'                   #タプルの要素は変更できません。

       ---------------------------------------------------------------------------
       TypeError                                 Traceback (most recent call last)
       <ipython-input-127-3c23335ca978> in <module>()
       ----> 1 abcd2[2] = 'hello'

       TypeError: 'tuple' object does not support item assignment
```

https://utokyo-ipp.github.io/2/2-2.html

条件分岐

- 先の条件分岐文では分岐の評価式は一つだけでしたが、以下のように elif を使うことで、複雑な条件を記述することができます。

```
if 式1:
   処理1
   …
elif 式2:
   処理2
   …
elif 式3:
   処理3
   …
else:
   処理n
   …
```

- elif の式は前の分岐文、if あるいは elif 文、の式が全て False と評価された時に評価され、True のときに直後の式が実行されます。そして、全ての式が False のとき、else 文の後の式が実行されます。
- 条件分岐文を入れ子にすることでより複雑な条件の処理をおこなうことができます。インデントレベルによって動作が異なることがありますので、十分注意しなければなりません。
- if 文のelif, else は省略することができます。

```
1  def sample_ifs(a,b,c):          # 3重の入れ子の if 文プログラムの例
2      if a == 0:
3          if b == 0:
4              if c == 0:
5                  print('a,b,c 全てゼロ')
6              elif c > 0:
7                  print('a,b はゼロ、c は正')
8              else:
9                  print('a,b はゼロ、c は負')
10         elif b > 0:
11             print('a はゼロ、 b は正、 c は不明')
12         else:
13             print('a はゼロ、 b は負、 c は不明')
14     else:
15         print('a はゼロではない, b および c は不明')
16
17 sample_ifs(0,1,2)
```

```
a はゼロ、b は正、c は不明
```

https://utokyo-ipp.github.io/2/2-3.html

辞書

- 辞書はキーと値を対応づけるデータ構造です。
- キーとしては、文字列・数値・タプルなど変更不可能なデータが使えます。リスト・辞書など変更可能なデータは利用できません。一方、値はどのようなデータでも扱えます。
- 辞書を作るには、コロン(:)でキーと値の対応づけた要素をコンマで区切り、全体を波括弧{...}で囲みます。リスト同様に要素を持たない空の辞書を作ることもできます。
- 辞書の値をを得るにはリストのインデックスと同様に
  辞書[キー]
  とします。
- 辞書の一覧を得るメソッドも、キー、値、キーと値それぞれに用意されています。これらのメソッドはくり返し処理で役立ちます。

```
[88]  1 dic1 = {'cat': 3, 'dog': 3, 'elephant': 8}   # 3要素の辞書
      2 for key in dic1:                              # 辞書にわたって繰り返し、キーだけが取り出される。
      3     print('key:', key, ', value:', dic1[key])  # キーと値を表示。

key: cat , value: 3
key: dog , value: 3
key: elephant , value: 8
```

```
[90]  1 ppap = {'apple' : 3, 'pen' : 5}      #2要素の辞書
      2 ppap                                 #中身を確認

{'apple': 3, 'pen': 5}
```

```
[94]  1 ppap['apple']                        #キーを指定して取り出す。

3
```

```
[95]  1 ppap['orange']                       #登録されていないキーを指定するとエラーになる。

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-95-50119f617b71> in <module>()
----> 1 ppap['orange']                    #登録されていないキーを指定するとエラーになる。

KeyError: 'orange'

SEARCH STACK OVERFLOW
```

```
      1 ppap['apple'] = 10                   # 既存のキーを指定し値を更新する。
      2 ppap['pinapple'] = 7                 # 既存のキーを指定し値を更新する。
      3 ppap

  File "<ipython-input-97-da09a14e96e7>", line 2
    ppap['pinapple'] = 7                    #
                     ^
SyntaxError: invalid character in identifier

SEARCH STACK OVERFLOW
```

https://utokyo-ipp.github.io/3/3-1.html

繰り返し

- 繰り返しはコンピュータが最も得意とする処理で、for や while 文を用いてプログラミングできます。
- for 文による繰り返しは以下のように記述します。if 文と同様に処理のインデントが深くなっていることに注意してください。

  for 変数 in 文字列・リスト・辞書など:
  　処理

- for 文では in の後の文字列・リスト・辞書などから繰り返しの順番で要素が取り出されます。変数は繰り返し処理で利用されます。
- 繰り返しでは、文字列の場合文字、リストでは要素、辞書の場合キーが取り出されます。
- Python で繰り返し処理の回数を指定するには、range 関数を利用します。range 関数の引数として一個の整数 N を与えると、0 から N–1 までの整数を順番に取り出すことができます。

- while 文による繰り返しは以下のように記述します。

  while 式:
  　処理

- 式が True の場合処理を繰り返します。False の場合 while を抜け、以降の処理に進みます。

```
[85]  1 words = ['dog', 'cat', 'mouse']  # 3要素のリスト
      2 for w in words:                  # リストにわたって繰り返し。
      3     print(w, len(w))             # リストの内容と、長さを表示。

dog 3
cat 3
mouse 5
```

```
[87]  1 word = 'Hello'              # 与える文字列
      2 for c in word:              # 文字列に渡って繰り返し、一文字ずつ取り出される。
      3     print(c)                # 取り出しら文字を印字。

H
e
l
l
o
```

```
      1 dic1 = {'cat': 3, 'dog': 3, 'elephant': 8}   # 3要素の辞書
      2 for key in dic1:                             # 辞書にわたって繰り返し、キーだけが取り出される。
      3     print('key:', key, ', value:', dic1[key])  # キーと値を表示。

key: cat , value: 3
key: dog , value: 3
key: elephant , value: 8
```

https://utokyo-ipp.github.io/3/3-2.html

ファイル操作

- ファイル操作は大量のデータを扱うには不可欠な処理です。
- プログラムはファイル操作の前に、ファイルを開く open を実行する必要があります。open ではファイル名と読み書きなどのモードなどを引数として、ファイルオブジェクトを返します。

  f = open('ファイル名', 'モード')

- その後の処理は、ファイルオブジェクトのメソッドを通じておこないます。
- とくに指定しなければファイルはテキストモードで扱われます。一行ずつ読み込み印字するには、for 文を利用して以下のように処理します。

  ```
  for line in f:
      print(line, end='')
  ```

- 利用の終わったファイルオブジェクトはクローズメソッドを読んで閉じる必要があります。ただしファイルオープンにあたって with 文を利用すれば、ブロックから抜ける際に自動的にクローズされます。

  ```
  with open('ファイル名', 'モード') as f:
      処理
  ```

- テキストファイルの扱いには文字コードの違いに注意する必要があります。
- ファイルを扱う際にはディレクトリやフォルダといったファイル構造に注意する必要があります。ファイル構造はコンピュータのオペレーティングシステムごとに異なります。

```
[61]  1 f=open('sample.txt', 'r') # file を open する

[62]  1 f.readline()              # 一行読み込む。
'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labo
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.\n'

[63]  1 f.readline()              # もう一行読み込む。
'Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.\

[64]  1 f.readline()              # さらに、もう一行読み込む。
'Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id e

[65]  1 f.readline()              # もう一行読み込む、最終行の場合空文字列が返される。
''

[59]  1 f.close()                 # 最後に close する。

      1 with open('sample.txt', 'r') as f:   # with を使うと close を呼ばなくても良い。
      2     for line in f:
      3         print(line)

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labo

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id es
```

https://utokyo-ipp.github.io/4/4-1.html
https://utokyo-ipp.github.io/4/4-3.html

プログラムの設計

- プログラムはコンピュータに実行させるための指示であると同時に、人間が読むための文書でもあります。プログラムの読みやすさのことを可読性といいます。

- 可読性を意識してプログラミングすることには多くの利点があります。もしプログラムが読みやすければ、間違いを見つけやすいのでより複雑なプログラムも正しく動かせるようになります。また、将来必要になるメンテナンスや機能追加も容易になります。

- 読みやすいプログラムを書くコツは読みやすい文章の書き方と同じです。
  - 要点が最初にまとめて書いてある方が読みやすい → 関数やモジュールに適切な名前をつける、それぞれのまとめをコメントに書く
  - 表現は簡潔な方が読みやすい → 同じ計算をよりシンプルに書けないか模索する
  - 適当な粒度で段落に分けてある方が読みやすい → 適当な粒度で関数、クラス、モジュールなどに分割する

生成 AI の利用とプログラムの可読性

- 近年ではプログラミングでも生成 AI の活用が増加しています。
- 簡単なプロンプトでも高度なプログラムを生成できるため、開発にかかる時間を大幅に削減することができます。
- その一方で、AI が生成したプログラムが ”間違い” を含むこともあります。このようなプログラムは実行しても期待した結果は得られません。ある入力で正しく動くように見えても、別の入力では正しく動かない、ということもよくあります。
- AI 時代のソフトウェア開発では、AI が生成したプログラムに深刻な問題が含まれていないかどうかを確認できる程度のプログラミングスキルを身につけることが重要です。

関心事の分離

- プログラム設計の有名な原則に separation of concern (関心事の分離) と呼ばれるものがあります。別々に考えるようなことは別々の場所に書きましょう、ということです。

- プレゼンテーションの原則にも ワンスライド・ワンメッセージ と呼ばれるものがあります。プログラムでもプレゼンテーションでも、複数の話題が混ざると人間には読みにくいようです。

```
def calculate(xs):
    n = 0
    m = 0
    l = 0
    for x in xs:
        n += 1
        m += x
        l += x ** 2
    return l/n - (m/n)**2
```

```
def mean(xs):
    return sum(xs) / len(xs)

def variance(xs):
    E = mean(xs)
    return mean(
        [(x-E)**2 for x in xs])
```

図： どちらもデータ列の分散を計算する関数の定義
どちらの方が読みやすいでしょうか

オブジェクト指向

- オブジェクト指向プログラミングでは、仕事を複数のオブジェクトで分担するようにプログラムします。

- １つのオブジェクトはいくつかのデータと、それらのデータを操作する関数を集めたものです。どんなデータにも管理責任を負うただ１つのオブジェクトが存在するので、権限や責任について考えやすくなります。

- イメージとしては会社の中で複数の人が連携して仕事を進めるような感じです。仕事の種類によって担当者を決めてあるので、オブジェクトは自身が担当する仕事をいくつかのサブタスクに分割し、担当者に依頼することで仕事を進めます。

```
class Producer:
  def __init__(self):
    self.N = 0
  def produce(self):
    self.N = self.N*2+1
    return self.N
```

```
class Consumer:
  def consume(self, N):
    print('consume:', hex(N))
```

```
p = Producer()
c = Consumer()
for _ in range(20):
  c.consume(p.produce())
```

図： 連携して動作するクラスの例

クラス宣言

- オブジェクトは、まずクラスを宣言してからインスタンス化するという手順で作成します。
- クラス文によってクラスを宣言します。クラス文は次のような構文です。

```
class（クラス名）:
  （クラス変数やメソッドの宣言）
```

- 次にインスタンス化をします。Python ではクラスを関数のように呼び出します。

```
（クラス名）((引数1), (引数2), ...)
```

- ドット記法でオブジェクト内部のデータや関数にアクセスできます。

```
（オブジェクト）.（フィールド名）
（オブジェクト）.（メソッド名）((引数1), (引数2), ...)
```

オブジェクト内部のデータを属性（またはフィールド、インスタンス変数など）、オブジェクト内部の関数をメソッドと呼びます。

```
class Producer:
  N = 0
  def produce(self):
    self.N = self.N*2+1
    return self.N

class Consumer:
  def consume(self, N):
    print('consume:', hex(N))

p = Producer()
c = Consumer()
for i in range(20):
  c.consume(p.produce())
```