令和2年度「専修学校による地域産業中核的人材養成事業」

AIプログラミングⅠ教材

Society5.0実現のためのIT技術者養成モデルカリキュラム開発と実証事業

令和２年度「専修学校による地域産業中核的人材養成事業」

AIプログラミングⅠ教材

Society5.0 実現のための IT 技術者養成モデルカリキュラム開発と実証事業

目次

シラバス …… 1
第 1 回：Python ことはじめ …… 3
第 2 回：変数とオブジェクト …… 14
第 3 回：組み込み関数と単一データのプログラミング …… 27
第 4 回：イテラブルなデータを使用したプログラミング …… 37
第 5 回：セット、辞書、タプルを使用したプログラミング …… 48
第 6 回：python による基本的なフロー制御 …… 58
第 7 回：python による基本的なフロー制御 2 …… 68
第 8 回：関数を利用したプログラミング …… 80
第 9 回：クラスを利用したプログラミング …… 93
第 10 回：モジュールを利用したプログラミング …… 107
第 11 回：継承 / 名前空間とスコープ / 例外処理 …… 121
第 12 回：標準ライブラリ …… 141
第 13 回：標準ライブラリ 2 …… 157
第 14 回：NumPy を使用したプログラミング …… 175
第 15 回：Matplotlib によるデータの可視化 …… 205

<table>
<tr><td>科目名</td><td colspan="5">AIプログラミング Ⅰ</td><td>週合計駒数</td><td>4駒</td><td>作成日</td><td></td></tr>
<tr><td rowspan="2">区分</td><td>必修</td><td rowspan="2">開講時期</td><td>1年次</td><td>週講義駒数</td><td>1駒</td><td>総時間数</td><td>120時間</td><td rowspan="2">担当教員</td><td rowspan="2"></td></tr>
<tr><td>講義・実習</td><td>後期</td><td>週実習等駒数</td><td>3駒</td><td>総単位数</td><td>5単位</td></tr>
</table>

目標	概要
Pythonの基本文法の習得、標準ライブラリを活用したプログラムを実装出来ることを目標とする。	Pythonの基本文法、標準ライブラリ・外部ライブラリの利用方法を学習する。

履修前提	テキスト・参考文献
※選択・エクステンションのみ記入	オリジナルテキスト

評価方法	関連科目
小テスト／中間テスト／期末テスト、提出課題、授業に取り組む姿勢(出席率、授業態度)	AIプログラミングⅡ、機械学習Ⅰ・Ⅱ・Ⅲ、データマイニング、AIシステム開発

回	学習目標	学習項目	理解度確認
1	Pythonの特徴について説明出来る。Pythonにおけるリテラルの種類と用途について説明出来る。	Pythonの歴史と発展を踏まえた上で、Pythonの動作原理、Pythonの特徴、Pythonのプログラミング・実行方法、コメントとインデント、ソースコードのエンコーディングについて学習する。Pythonにおけるリテラルについても学習する。	小テスト、練習問題
2	変数を利用したプログラムを作成出来る。Pythonにおけるオブジェクトについて説明出来る。	変数とオブジェクト:変数の使用目的を踏まえた上で、Pythonにおける変数のメカニズム、宣言と代入、識別子とキーワードについて学習します。併せて、Pythonにおけるオブジェクトとその仕組みについても学習する。	小テスト、練習問題
3	単一データを使用したプログラムを作成出来る。	データ操作(1):built-in関数とオブジェクトの種類について理解した上で、単一データの基本操作について学習する。	小テスト、練習問題
4	イテラブルなデータを使用したプログラムを作成出来る。	データ操作(2):複数の値を持つデータのうち、シーケンシャルなオブジェクトとイテラブルなオブジェクト基本操作について学習する。	小テスト、練習問題
5	セット、辞書、タプルを使用したプログラムを作成出来る。	データ操作(3):セット、辞書、タプルの違いを理解し、それぞれの形式におけるプログラミングについて学習する。	小テスト、練習問題
6	基本的なフロー制御を利用したプログラムを作成出来る。	フロー制御(1):構造化定理に基づく基本制御構造とフローを理解した上で、Pythonにおけるフロー制御の方法について学習する。具体的には、if文、条件演算子、while文、for文について学習する。	小テスト、練習問題
7	Python独自のフロー制御を利用したプログラムを作成出来る。	フロー制御(2):breakとcontinue、フロー制御で利用されるbuilt-in関数について学習する。	小テスト、練習問題
8	関数を利用したプログラムを作成出来る。	関数:Pythonにおける関数の特徴を踏まえた上で、関数定義、関数呼び出し方法、関数オブジェクトについて学習する。	小テスト、練習問題
9	クラスを利用したプログラムを作成出来る。	クラスとモジュール(1):クラスの作成意義を踏まえた上で、クラス定義、クラスのアトリビュートとメソッド、インスタンス化、クラスメソッド、staticメソッドについて学習する。	小テスト、練習問題
10	モジュールを利用したプログラムを作成出来る。	クラスとモジュール(2):モジュールの作成意義を踏まえた上で、モジュール定義、モジュールのインポート方法、モジュールの作成方法についても学習する。併せて、パッケージについても学習する。	小テスト、練習問題
11	継承を利用したプログラムを作成出来る。名前空間とスコープについて説明出来る。例外処理を利用したプログラムを作成出来る。	継承、名前空間とスコープ、例外処理:継承の目的および長所と短所を確認した上で、継承の実現方法について学習する。Pythonにおける名前空間とスコープのメカニズム、例外処理の方法について学習する。	小テスト、練習問題

12	学習目標 Pythonの標準ライブラリを使用して、システム操作、数学関連のプログラムを作成出来る。Pythonにおける文字列のメカニズムについて説明出来る。	学習項目 標準ライブラリ(1):Pythonの標準ライブラリを使用したシステム操作、数値計算について学習する。また、Pythonにおける文字列のメカニズムについても学習する。
	理解度確認: 小テスト、練習問題	
13	学習目標 Pythonの標準ライブラリを使用して、日付操作、正規表現、ファイル入出力を利用したプログラムを作成出来る。	学習項目 標準ライブラリ(2):Pythonの標準ライブラリを使用した日付操作、正規表現、ファイル入出力について学習する。
	理解度確認: 小テスト、練習問題	
14	学習目標 NumPyを使用して線型代数の計算を実装できる。	学習項目 外部ライブラリ(1):機械学習を学ぶ上で必要な線型代数の基本(ベクトル・行列・テンソルの定義と演算、幾何学的意味)について学習する。併せて、NumPyを利用した線型代数の計算を実装する。
	理解度確認: 小テスト、練習問題	
15	学習目標 Matplotlibを使用してデータの可視化が出来る。	学習項目 外部ライブラリ(2):Matplotlibによる数値の可視化方法について学習する。
	理解度確認: 小テスト、練習問題	

第1回：Pythonことはじめ

アジェンダ

- Pythonとは
- 分析環境の構築
- 演習

全15回の講義について

- Pythonの基本文法習得と実装スキルの習得を目標とする。
  - Pythonの基本文法の習得を目指す
  - Pythonの標準ライブラリ、外部ライブラリを活用したプログラミングスキルの習得を目指す

Pythonとは

- 「元々はAmoebaの使用言語であるABC言語に例外処理やオブジェクト指向を対応させるために作られた言語である。」（Wikipedia）
- Pythonはデータサイエンスの分野でよく使用される言語です。Pythonには以下の特徴があります。
  - 文法がシンプルでコーディング量が少なくて済む。
  - 数値計算、統計処理で活用できるライブラリが豊富。
  - Webアプリ、API開発に活用できるライブラリも豊富。
- 数多くのコミュニティがPython による開発をしていますが、以下のことは苦手です。
  - スマホアプリ開発。
  - ゲームプログラミング。

Jupyter

- WebブラウザからPythonなどのプログラミング言語を実行させる環境
- もともとは、Pythonのインタラクティブシェルをより強力にしたIPythonが始まり
  - セル単位でのプログラム実行とタブ補完
  - オブジェクトの調査とシェルコマンドの実行
  - etc…
- IPythonをWeb経由に実行させられるものとして、IPython Notebookが登場
  - テーブルやグラフ、数式の埋め込み
  - Markdownの埋め込み
  - .ipynb形式により、分析過程自体を再実行性を担保して保存と共有が可能に
- IPython NotebookをPython以外の言語にも対応させたものがJupyter Notebook
  - Python
  - R
  - Julia

分析環境の構築

分析環境の構築

- 前提としてはPythonは3系の最新版である3.6系
- 構築方法としては下記2つ
  - 素のPythonに必要となるライブラリをインストールしていく方法
    - Pythonの環境を自力で構築できる方向け
  - Anaconda(miniconda)を利用する方法
    - Pythonの環境を自力で構築する自身がない方向け
    - インストーラを使って必要なソフトウェアー式のインストーラを行う

1. 素のPythonからインストール

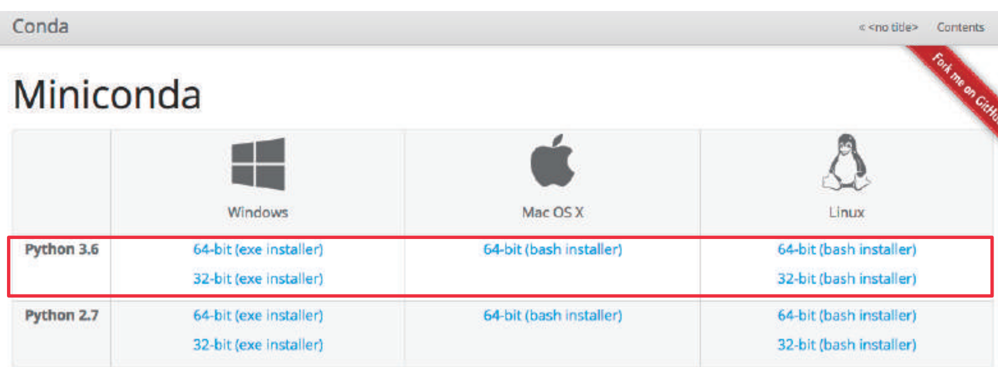
- Pythonにはパッケージ管理システムであるpipが標準で付属しているため、pipを利用して各種ライブラリをインストールする
  - Python3系そのもののインストールは割愛

```
$ pip install jupyter numpy pandas scipy matplotlib scikit-learn
```

2. Anaconda(miniconda)を利用してインストール

- AnacondaはPython及び分析に必要なライブラリー式を提供してくれるパッケージ
- minicondaはAnacondaの中で分析に必要な最小限のみ集めた軽量のディストリビューション
  - Anacondaのダウンロードページ(https://conda.io/miniconda.html)から使用しているOSに合わせたインストーラを取得してダウンロードする
  - 今回はPython3.6系が前提のため、Python3.6系のインストーラを利用する

2. Anacondaを利用してインストール（続き）



3. minicondaの使い方

- 環境を作る

```
$ conda create -n analytics
```

- 環境のアクティベート

```
$ activate analytics
```

- 環境内でのパッケージインストール

```
$ pip(conda) install jupyter numpy pandas scipy matplotlib scikit-learn
```

- 環境の終了

```
$ deactivate analytics
```

以後の作業

- Jupyter notebook上での作業を推奨とします。
- コマンドを実行したディレクトリをルートとしてjupyterが起動します。

```
$ jupyter notebook
```

演習

演習1：Pythonの実行（ターミナル編）

- ターミナルからpythonを起動してください。
- Pythonのバージョンを確認してください。

```
— python — 80×24
Last login: Fri Nov 15 07:52:16 on ttys000
p(anaconda-2.4.0) (base) $ python
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hi!!")
Hi!!
>>>
```

```
bash — 80×24
(anaconda-2.4.0) (base) $ python --version
Python 3.7.3
(anaconda-2.4.0) (base)
```

演習2：Pythonの実行（Jupyter notebook編）

- ターミナルからJupyter notebookを実行してください。
- 「Chapter01_introduction.ipynb」を開いてください。

リテラル

- Pythonにはリテラルという表記法があります。
- “[ダブルコーテーション]や’ [シングルコーテーション]で囲ったものを文字列リテラルといいます。
- 小数や整数を記述したものを数値リテラルといいます。

演習3：文字列リテラルの記述

- 「こんにちは」という文字列を出力する処理を実装してください。

演習4：文字列リテラルの記述

- 「今日の天気は‘晴れ’です。」というように、文字列にシングルコーテーションを含む文字列リテラルを出力する処理を実装してください。
- 「今日の天気は”晴れ”です。」というように、文字列にダブルコーテーションを含む文字列リテラルを出力する処理を実装してください。

コードに記載するコメント

- Pythonでコードにコメントを挿入する際は「#」（シャープ）を使います。「#」以降に書く文章は、その行の終わりまで無視されます。
- 複数行のコメントを挿入する際には「 ‘ 」（シングルクォーテーション）または「“」（ダブルクォーテーション）を3つでコメントアウトする部分を囲みます。

演習5：コメントの記載

- コメントを1行記載してください。
- 複数行のコメント文を記載してください。

インデント

- 後の章で学習するif 文や for 文など、内部に別の文を持つ文を複合文といいます。
- if 文では条件式が真の場合に内部の処理が実行されますが、どこからどこまでの文を実行するのかを示すのに使われるのがブロックです。Javaなどでは、下記の様に{…}で表現されます。

```
if (条件式) {
    ブロック内の処理1
    ブロック内の処理2
}
```

- Python では特別な文字を使わずに、同じインデントがされている文を同じブロックとして扱います。

```
if 条件式:
    ブロック内の処理1
    ブロック内の処理2
```

第2回：変数とオブジェクト

アジェンダ

- 前回までの講義の振り返り
- 変数とは
- オブジェクトとは

前回までの講義の振り返り

Pythonとは

- 「元々はAmoebaの使用言語であるABC言語に例外処理やオブジェクト指向を対応させるために作られた言語である。」（Wikipedia）
- Pythonはデータサイエンスの分野でよく使用される言語です。Pythonには以下の特徴があります。
  - 文法がシンプルでコーディング量が少なくて済む。
  - 数値計算、統計処理で活用できるライブラリが豊富。
  - Webアプリ、API開発に活用できるライブラリも豊富。
- 数多くのコミュニティがPython による開発をしていますが、以下のことは苦手です。
  - スマホアプリ開発。
  - ゲームプログラミング。

Jupyter

- WebブラウザからPythonなどのプログラミング言語を実行させる環境
- もともとは、Pythonのインタラクティブシェルをより強力にしたIPythonが始まり
  - セル単位でのプログラム実行とタブ補完
  - オブジェクトの調査とシェルコマンドの実行
  - etc…
- IPythonをWeb経由に実行させられるものとして、IPython Notebookが登場
  - テーブルやグラフ、数式の埋め込み
  - Markdownの埋め込み
  - .ipynb形式により、分析過程自体を再実行性を担保して保存と共有が可能に
- IPython NotebookをPython以外の言語にも対応させたものがJupyter Notebook
  - Python
  - R
  - Julia

リテラル

- Pythonにはリテラルという表記法があります。
- “[ダブルコーテーション]や’[シングルコーテーション]で囲ったものを文字列リテラルといいます。
- 小数や整数を記述したものを数値リテラルといいます。

コードに記載するコメント

- Pythonでコードにコメントを挿入する際は「#」(シャープ)を使います。「#」以降に書く文章は、その行の終わりまで無視されます。
- 複数行のコメントを挿入する際には「‘」(シングルクォーテーション)または「“」(ダブルクォーテーション)を3つでコメントアウトする部分を囲みます。

インデント

- 後の章で学習するif 文や for 文など、内部に別の文を持つ文を複合文といいます。
- if 文では条件式が真の場合に内部の処理が実行されますが、どこからどこまでの文を実行するのかを示すのに使われるのがブロックです。Javaなどでは、下記の様に{…}で表現されます。

```
if (条件式) {
   ブロック内の処理1
   ブロック内の処理2
}
```

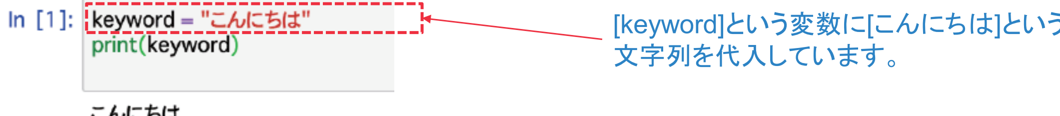
- Python では特別な文字を使わずに、同じインデントがされている文を同じブロックとして扱います。

```
if 条件式:
   ブロック内の処理1
   ブロック内の処理2
```

変数とは

変数とは

- （Pythonなどのプログラム処理において）変数とは文字列や数値、計算式を任意のキーワードで保持する仕組みです。
- 技術的な言葉で言い換えると、「メモリ上のデータにアクセスしやすいようにつける名前」です。



演習1：変数の記述

- 「こんにちは」という文字列を変数に代入し、その変数を出力する処理を実装してください。

変数の型

- Pythonでは変数に値を代入すると、自動で型を判定してくれます。
- 変数の型はtype()関数で確認することができます。

演習2：変数の型の確認

- 変数に文字列、数値を代入してください。
- それぞれの変数の型を確認してください。

演習3：型の自動変換

- 整数型と浮動小数点型を足した場合の型を確認してください。

変数の型変換

- Pythonの変数には文字列型、整数型、浮動小数点型などがありますが、相互に型の変換を行うことができます。
- 例えばstr()関数を使用すると文字列型に、int()関数を使用すると整数型に、float()関数を使用すると浮動小数点型に変換することができます。

```
# 文字列型を整数型に変換します。
param_1 = "1"
print(param_1, type(param_1))

param_2 = int(param_1)
print(param_2, type(param_2))

# 文字列型を浮動小数点型に変換します。
param_3 = "1.23"
print(param_3, type(param_3))

param_4 = float(param_3)
print(param_4, type(param_4))
```

変数の型変換

- “1–23”という文字列を浮動小数点型に変換しようとしたり、“1.23”という文字列を整数型に変換しようとすると、エラーが発生します。型変換は適切な表現のみが実行可能です。

```
param_ = "1.23"
# エラーが発生します。
print(type(int(param_)))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-39906922da24> in <module>
      1 param_ = "1.23"
      2 # エラーが発生します。
----> 3 print(type(int(param_)))

ValueError: invalid literal for int() with base 10: '1.23'
```

演習4：型の変換

- 文字列型、整数型、浮動小数点型間の変換を実行してください。
- ″1.23″という文字列を整数型に変換する処理を実行してください（エラーが発生します）。
- ″1-23″という文字列を浮動小数点型に変換する処理を実行してください（エラーが発生します）。

変数の命名規則

- 変数の命名に使用できるのは下記の文字です。
  - 小文字の英字
  - 大文字の英字
  - 数字
  - アンダースコア
- 数字は変数の名前の先頭には使えません。

```
# 記号の使用
~aa = 1
```

```
  File "<ipython-input-26-d086c52fd5b3>", line 1
    ~aa = 1
       ^
SyntaxError: can't assign to operator
```

```
# 先頭に数字を使用
1a = "chars"
```

```
  File "<ipython-input-27-a502d951e202>", line 2
    1a = "chars"
     ^
SyntaxError: invalid syntax
```

演習5：命名規則

- 命名規則に従わない変数名でエラーが発生することを確認してください。

予約語と組み込み関数名

- 変数名はルールに反しなければ、基本的には自由につけることができます。
- ただし「予約語」や「組み込み関数名」と同じ名前をつけるのは避けたほうがよいです。

```
param_int = 1
type = float
print(type(param_int))
```

```
1.0
```

例：[type]という変数にfloatを代入したため、[type]関数は型を取得する関数ではなく、浮動小数点型に変換する関数となってしまいました。

演習6：予約語、組み込み関数名の確認

- 予約語を確認してください。

```
print(__import__('keyword').kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'globa
l', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

- 組み込み関数名を確認してください。

```
print(dir(__builtins__))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildPro
cessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ell
ipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOErro
r', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupE
rror', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'Ov
erflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'Runti
meError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'Timeo
utError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'Unicode
Warning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__import__',
'__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'cl
assmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'filter', 'float', 'forma
t', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'lis
t', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

オブジェクトとは

オブジェクトとは

Python公式ドキュメントより（ https://docs.python.org/ja/3/reference/datamodel.html#objects-values-and-types ）

- Python における オブジェクト (object) とは、データを抽象的に表したものです。Python プログラムにおけるデータは全て、オブジェクトまたはオブジェクト間の関係として表されます。
- すべての属性は、同一性 (identity)、型、値をもっています。同一性 は生成されたあとは変更されません。これはオブジェクトのアドレスのようなものだと考えられるかもしれません。'is' 演算子は2つのオブジェクトの同一性を比較します。id() 関数は同一性を表す整数を返します。

オブジェクトとは

- 具体的には「整数や文字列、リスト、タプルなどのデータ」はオブジェクトです。

```
param_string = "文字列です。"
param_int = 1
param_float = 1.0

print(type(param_string))
print(type(param_int))
print(type(param_float))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
```

文字列型、整数型、浮動小数点型の変数やリテラルは
オブジェクトです。

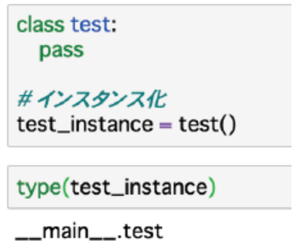
```
list_ = [0.0, 1.0, 2.0]
print(type(list_))
```

```
<class 'list'>
```

リストもオブジェクトです。

オブジェクトとは

- クラスオブジェクトをインスタンス化したものもオブジェクトです。



```
class test:
    pass

# インスタンス化
test_instance = test()
```

```
type(test_instance)
```

```
__main__.test
```

参考

- 「Chapter02_variable_object.ipynb」の参考に記載されているコードを実際に動かしてみて、コードの挙動を確認してみましょう。

第3回：組み込み関数と単一データのプログラミング

アジェンダ

- 前回までの講義の振り返り
- 組み込み関数とは

前回までの講義の振り返り

変数とは

- （Pythonなどのプログラム処理において）変数とは文字列や数値、計算式を任意のキーワードで保持する仕組みです。
- 技術的な言葉で言い換えると、「メモリ上のデータにアクセスしやすいようにつける名前」です。

変数の型変換

- Pythonの変数には文字列型、整数型、浮動小数点型などがありますが、相互に型の変換を行うことができます。
- 例えばstr()関数を使用すると文字列型に、int()関数を使用すると整数型に、float()関数を使用すると浮動小数点型に変換することができます。

```
# 文字列型を整数型に変換します。
param_1 = "1"
print(param_1, type(param_1))

param_2 = int(param_1)
print(param_2, type(param_2))

# 文字列型を浮動小数点型に変換します。
param_3 = "1.23"
print(param_3, type(param_3))

param_4 = float(param_3)
print(param_4, type(param_4))
```

変数の型変換

- “1-23”という文字列を浮動小数点型に変換しようとしたり、“1.23”という文字列を整数型に変換しようとすると、エラーが発生します。型変換は適切な表現のみが実行可能です。

```
param_ = "1.23"
# エラーが発生します。
print(type(int(param_)))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-39906922da24> in <module>
      1 param_ = "1.23"
      2 # エラーが発生します。
----> 3 print(type(int(param_)))

ValueError: invalid literal for int() with base 10: '1.23'
```

変数の命名規則

- 変数の命名に使用できるのは下記の文字です。
  - 小文字の英字
  - 大文字の英字
  - 数字
  - アンダースコア
- 数字は変数の名前の先頭には使えません。

```
# 記号の使用
~aa = 1
```

```
  File "<ipython-input-26-d086c52fd5b3>", line 1
    ~aa = 1
       ^
SyntaxError: can't assign to operator
```

```
# 先頭に数字を使用
1a = "chars"
```

```
  File "<ipython-input-27-a502d951e202>", line 2
    1a = "chars"
     ^
SyntaxError: invalid syntax
```

予約語と組み込み関数名

- 変数名はルールに反しなければ、基本的には自由につけることができます。
- ただし「予約語」や「組み込み関数名」と同じ名前をつけるのは避けたほうがよいです。

```
param_int = 1
type = float
print(type(param_int))
```

```
1.0
```

例：[type]という変数にfloatを代入したため、[type]関数は型を取得する関数ではなく、浮動小数点型に変換する関数となってしまいました。

オブジェクトとは

Python公式ドキュメントより（ https://docs.python.org/ja/3/reference/datamodel.html#objects-values-and-types ）

- Python における オブジェクト (object) とは、データを抽象的に表したものです。Python プログラムにおけるデータは全て、オブジェクトまたはオブジェクト間の関係として表されます。
- すべての属性は、同一性 (identity)、型、値をもっています。同一性 は生成されたあとは変更されません。これはオブジェクトのアドレスのようなものだと考えられるかもしれません。'is' 演算子は2つのオブジェクトの同一性を比較します。id() 関数は同一性を表す整数を返します。

オブジェクトとは

- 具体的には「整数や文字列、リスト、タプルなどのデータ」はオブジェクトです。

```
param_string = "文字列です。"
param_int = 1
param_float = 1.0

print(type(param_string))
print(type(param_int))
print(type(param_float))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
```

文字列型、整数型、浮動小数点型の変数やリテラルは
オブジェクトです。

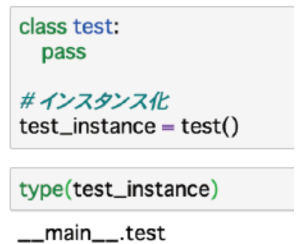
```
list_ = [0.0, 1.0, 2.0]
print(type(list_))
```

```
<class 'list'>
```

リストもオブジェクトです。

オブジェクトとは

- クラスオブジェクトをインスタンス化したものもオブジェクトです。


```
class test:
    pass

# インスタンス化
test_instance = test()
```
```
type(test_instance)
```
```
__main__.test
```

組み込み関数とは

組み込み関数とは

- 標準で用意されている関数のことを「組み込み関数と呼びます。特別な設定を行わなくても利用することが可能です。
- 組み込み関数名は下記のように確認することができます。

```
print(dir(__builtins__))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildPro
cessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ell
ipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOErro
r', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupE
rror', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'Ov
erflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'Runti
meError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'Timeo
utError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'Unicode
Warning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__import__',
'__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'cl
assmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'filter', 'float', 'forma
t', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'lis
t', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

組み込み関数の例

- abs()
  - 数値の絶対値を返却する関数です。
- all()
  - 引数に渡した要素がすべて真であればTrueを返します。それ以外はFalseです。
- any()
  - OR条件で引数に渡した要素のいずれかが真であればTrueを返します。それ以外はFalseです。
- bin()
  - 引数に整数を渡すと、頭に“0b”のついた2進数表現の文字列に変換します。※整数型ではなく、文字列を返すところに注意してください。
- bool()
  - 引数の値を評価して真ならTrue、偽ならFalseを返却します。

組み込み関数の例

- complex()
  - 複素数を扱うクラスです。引数には複素数表現の文字列か、第一引数に実数部、第二引数に虚数部の数値を渡します。
- dict()
  - 辞書クラスです。引数から新しい辞書オブジェクトを生成して返します。
- divmod()
  - 2つの引数をとって、その整数での割り算の結果と余りのペアを返却します。
- enumerate()
  - 引数にイテレーション可能なオブジェクトを渡すとenumerateオブジェクトを返します。
  - enumerateオブジェクトはイテレータであり、_next_()を呼び出す度に呼び出した回数と元のオブジェクトの要素のタプルを返します。

組み込み関数の例

- eval()
  - 引数をPythonの式として評価します。
- exec()
  - 引数に渡した文を実行することができます。
- filter()
  - 引数で渡すリストなどのiterableから条件に合う値だけをフィルタリングすることができます。条件は関数として渡します。
- float()
  - 浮動小数点型のオブジェクトを生成します。
- format()
  - 第1引数にフォーマットしたい値を、第2引数にフォーマット指定文字列を渡すことで変換後の値が返却されます。

組み込み関数の例

- frozenset()
  - iterableを引数にとって、frozensetオブジェクトを返却します。基本的にset型と同じように集合演算を扱えます。
- help()
  - Pythonオブジェクトの説明を見ることができる非常に便利な関数です。
- hex()
  - 引数に整数を渡すと、頭に0xをつけた16進数の文字列に変換します。。
- int()
  - 整数型クラスのインスタンスオブジェクトを返却します。
- isinstance()
  - 指定したオブジェクトが、あるクラスのインスタンスかどうかを判定しTrue/Falseを返却します。

組み込み関数の例

- len()
  - リストや辞書のサイズを取得する関数です。
- list()
  - listオブジェクトを生成します。
- map()
  - iterable の全ての要素に function を適用し，その結果から構成されるイテレータを返します。
- max()
  - 2 つ以上の引数の中で最大のものを返します。
- min()
  - 2 つ以上の引数の中で最小のものを返します。

組み込み関数の例

- next()
  - iterator の次の要素を取得します。
- oct()
  - 整数 x を 8 進文字列に変換します。
- pow(x, y[, z])
  - x の y 乗を返すします。引数として z が与えられると、x の y 乗に対する z の剰余を返します。
- range(start, stop[, step])
  - range型オブジェクトを生成します。
- reversed()
  - 要素を逆順に取り出すイテレータを返します。

組み込み関数の例

- sorted()
  - iterable の要素を並べ替えた新たなリストを返します。
- sum(iterable[, start])
  - startとiterableの各要素を左から右へ合計します。start のデフォルトは 0 です。

第4回：イテラブルなデータを使用したプログラミング

アジェンダ

- 前回までの講義の振り返り
- イテラブル、シーケンスとは

前回までの講義の振り返り

組み込み関数とは

- 標準で用意されている関数のことを｢組み込み関数と呼びます。特別な設定を行わなくても利用することが可能です。

- 組み込み関数名は下記のように確認することができます。

```
print(dir(__builtins__))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildPro
cessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ell
ipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOErro
r', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupE
rror', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'Ov
erflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'Runti
meError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'Timeo
utError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'Unicode
Warning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__import__',
'__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'cl
assmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'filter', 'float', 'forma
t', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'lis
t', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

組み込み関数の例

- abs()
  - 数値の絶対値を返却する関数です。
- all()
  - 引数に渡した要素がすべて真であればTrueを返します。それ以外はFalseです。
- any()
  - OR条件で引数に渡した要素のいずれかが真であればTrueを返します。それ以外はFalseです。
- bin()
  - 引数に整数を渡すと、頭に“0b”のついた2進数表現の文字列に変換します。※整数型ではなく、文字列を返すところに注意してください。
- bool()
  - 引数の値を評価して真ならTrue、偽ならFalseを返却します。

組み込み関数の例

- complex()
  - 複素数を扱うクラスです。引数には複素数表現の文字列か、第一引数に実数部、第二引数に虚数部の数値を渡します。
- dict()
  - 辞書クラスです。引数から新しい辞書オブジェクトを生成して返します。
- divmod()
  - 2つの引数をとって、その整数での割り算の結果と余りのペアを返却します。
- enumerate()
  - 引数にイテレーション可能なオブジェクトを渡すとenumerateオブジェクトを返します。
  - enumerateオブジェクトはイテレータであり、_next_()を呼び出す度に呼び出した回数と元のオブジェクトの要素のタプルを返します。

組み込み関数の例

- eval()
  - 引数をPythonの式として評価します。
- exec()
  - 引数に渡した文を実行することができます。
- filter()
  - 引数で渡すリストなどのiterableから条件に合う値だけをフィルタリングすることができます。条件は関数として渡します。
- float()
  - 浮動小数点型のオブジェクトを生成します。
- format()
  - 第1引数にフォーマットしたい値を、第2引数にフォーマット指定文字列を渡すことで変換後の値が返却されます。

組み込み関数の例

- frozenset()
  - iterableを引数にとって、frozensetオブジェクトを返却します。基本的にset型と同じように集合演算を扱えます。
- help()
  - Pythonオブジェクトの説明を見ることができる非常に便利な関数です。
- hex()
  - 引数に整数を渡すと、頭に0xをつけた16進数の文字列に変換します。。
- int()
  - 整数型クラスのインスタンスオブジェクトを返却します。
- isinstance()
  - 指定したオブジェクトが、あるクラスのインスタンスかどうかを判定しTrue/Falseを返却します。

組み込み関数の例

- len()
  - リストや辞書のサイズを取得する関数です。
- list()
  - listオブジェクトを生成します。
- map()
  - iterable の全ての要素に function を適用し，その結果から構成されるイテレータを返します。
- max()
  - 2 つ以上の引数の中で最大のものを返します。
- min()
  - 2 つ以上の引数の中で最小のものを返します。

組み込み関数の例

- next()
  - iterator の次の要素を取得します。
- oct()
  - 整数 x を 8 進文字列に変換します。
- pow(x, y[, z])
  - x の y 乗を返すします。引数として z が与えられると、x の y 乗に対する z の剰余を返します。
- range(start, stop[, step])
  - range型オブジェクトを生成します。
- reversed()
  - 要素を逆順に取り出すイテレータを返します。

組み込み関数の例

- sorted()
  - iterable の要素を並べ替えた新たなリストを返します。
- sum(iterable[, start])
  - startとiterableの各要素を左から右へ合計します。start のデフォルトは 0 です。

イテラブル、シーケンスとは

組み込みデータ型

- Pythonの組み込みデータ型にはイミュータブル（immutable）、ミュータブル（mutable）、イテラブル（iterable）、シーケンス（sequence）、マッピング（mapping）などがあります。

組み込み型	変更不可 (immutable)	変更可 (mutable)	反復可 (iterable)	シーケンス (sequence)	マッピング (mapping)
bool	○				
int	○				
float	○				
complex	○				
str	○		○	○	
list		○	○	○	
tuple	○		○	○	
range	○		○	○	
dict		○	○		○
set		○	○		
bytes	○		○	○	
bytearray		○	○	○	
file object	○		○		

参照： https://gammasoft.jp/blog/python-built-in-types/

反復可(iterable)

- イテラブルとは、要素を1つずつ返すことができるオブジェクトです（for文で1つずつ要素を取り出せるオブジェクト）。

```
test_list = [0, 1, 2, 3, 4]

for e in test_list:
    print(e)
```

```
0
1
2
3
4
```

反復可(iterable)

- イテラブルなデータはfor文のみでなく、nextメソッドによって状態を進めることができます。
- イテレーションが不可能な状態になる(最後の要素に達する)とStopIterationという例外を発生させます。

```
test_list = [1, 2, 3]
# イテレータを作成します。
itr_test = iter(test_list)
# 要素にアクセスします。
print(next(itr_test))
print(next(itr_test))
print(next(itr_test))
# 最後の要素に達した後、もう一度nextを実行すると、エラーが発生します。
print(next(itr_test))
```

```
1
2
3

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-4-b760965137f7> in <module>
      7 print(next(itr_test))
      8 # 最後の要素に達した後、もう一度nextを実行すると、エラーが発生します。
----> 9 print(next(itr_test))

StopIteration:
```

演習1：for文によるイテラブルなデータへのアクセス

- リストの要素をfor文で取り出すプログラムを実装してください。
- 文字列の各文字をfor文で取り出すプログラムを実装してください。

演習2：イテレーターによるイテラブルなデータへのアクセス

- リストの要素をイテレーターのnext関数で取り出すプログラムを実装してください。

リストとイテレータによるアクセスの違い

- イテレータによるアクセスをする場合、どの要素が参照されているかという「状態」を保持することができます。一度すべての要素にアクセスした後に再度アクセスを試みても、アクセスすることができません。
- 一方、リストは状態を持たないため、再度リストのはじめからすべての要素にアクセスできます。

```
test_list = [0, 1, 2]
# イテレータを作成します。
itr_test = iter(test_list)

# 一度、すべての要素にアクセスします。
for e in itr_test:
    print(e)

# 再度すべての要素にアクセスを試みますが、アクセスできません。
for e in itr_test:
    print(e)
```

```
0
1
2
```

```
test_list = [0, 1, 2]

# 一度、すべての要素にアクセスします。
for e in test_list:
    print(e)

# 再度すべての要素にアクセスすることができます。
for e in test_list:
    print(e)
```

```
0
1
2
0
1
2
```

演習3：リストとイテレータによるアクセスの違い

- リストとイテレータによるアクセスの違いを確認してください。

シーケンス（sequence）

- 整数のインデックスを指定して要素にアクセスできるデータ型です。
- 組み込み関数len()で要素数（長さ）を取得できます。また、文字列（str）もシーケンスです。
- シーケンスはすべて「反復できる（イテラブル）」オブジェクトです。

```
vegetables = ["potato", "tomato", "onion"]
# 2番目の要素にアクセスを試みます。
print(vegetables[1])
```

```
tomato
```

```
sentence = "今日はよく晴れていますね。"
# 9文字目にアクセスを試みます。
print(sentence[8])
```

```
い
```

演習4：シーケンスなデータへのアクセス

- シーケンスなデータに対し、インデックスを指定して要素にアクセスしてください。

第5回：セット、辞書、タプルを使用したプログラミング

アジェンダ

- 前回までの講義の振り返り
- セット、辞書、タプルとは

前回までの講義の振り返り

組み込みデータ型

- Pythonの組み込みデータ型にはイミュータブル（immutable）、ミュータブル（mutable）、イテラブル（iterable）、シーケンス（sequence）、マッピング（mapping）などがあります。

組み込み型	変更不可 (immutable)	変更可 (mutable)	反復可 (iterable)	シーケンス (sequence)	マッピング (mapping)
bool	○				
int	○				
float	○				
complex	○				
str	○		○	○	
list		○	○	○	
tuple	○		○	○	
range	○		○	○	
dict		○	○		○
set		○	○		
bytes	○		○	○	
bytearray		○	○	○	
file object	○		○		

参照： https://gammasoft.jp/blog/python-built-in-types/

反復可(iterable)

- イテラブルとは、要素を1つずつ返すことができるオブジェクトです（for文で1つずつ要素を取り出せるオブジェクト）。

```
test_list = [0, 1, 2, 3, 4]

for e in test_list:
    print(e)
```

```
0
1
2
3
4
```

反復可(iterable)

- イテラブルなデータはfor文のみでなく、nextメソッドによって状態を進めることができます。
- イテレーションが不可能な状態になる（最後の要素に達する）とStopIterationという例外を発生させます。

```
test_list = [1, 2, 3]
# イテレータを作成します。
itr_test = iter(test_list)
# 要素にアクセスします。
print(next(itr_test))
print(next(itr_test))
print(next(itr_test))
# 最後の要素に達した後、もう一度nextを実行すると、エラーが発生します。
print(next(itr_test))
```

```
1
2
3

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-4-b760965137f7> in <module>
      7 print(next(itr_test))
      8 # 最後の要素に達した後、もう一度nextを実行すると、エラーが発生します。
----> 9 print(next(itr_test))

StopIteration:
```

リストとイテレータによるアクセスの違い

- イテレータによるアクセスをする場合、どの要素が参照されているかという「状態」を保持することができます。一度すべての要素にアクセスした後に再度アクセスを試みても、アクセスすることができません。
- 一方、リストは状態を持たないため、再度リストのはじめからすべての要素にアクセスできます。

```
test_list = [0, 1, 2]
#イテレータを作成します。
itr_test = iter(test_list)

#一度、すべての要素にアクセスします。
for e in itr_test:
    print(e)

#再度すべての要素にアクセスを試みますが、アクセスできません。
for e in itr_test:
    print(e)
```

```
0
1
2
```

```
test_list = [0, 1, 2]

#一度、すべての要素にアクセスします。
for e in test_list:
    print(e)

#再度すべての要素にアクセスすることができます。
for e in test_list:
    print(e)
```

```
0
1
2
0
1
2
```

シーケンス（sequence）

- 整数のインデックスを指定して要素にアクセスできるデータ型です。
- 組み込み関数len()で要素数（長さ）を取得できます。また、文字列（str）もシーケンスです。
- シーケンスはすべて「反復できる（イテラブル）」オブジェクトです。

```
vegetables = ["potato", "tomato", "onion"]
#2番目の要素にアクセスを試みます。
print(vegetables[1])
```

```
tomato
```

```
sentence = "今日はよく晴れていますね。"
#9文字目にアクセスを試みます。
print(sentence[8])
```

```
い
```

セット、辞書、タプルとは

セット（set）とは

- セットは集合型ともいわれる型で、要素を波カッコ{}でくくります。
- セットは一意な要素のみを保持します。

```
test_set = {1, 2, 3, 4, 5}
print(test_set)

# 重複した要素を持っています。
test_set = {1, 2, 3, 3, 4, 5, 5, 5}
print(test_set)
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

セット（set）とは

- セットは要素の順序を保たないという特徴も持っています。
- 順序を保たないため、インデックス指定をして要素を取り出す、といったことはできません。

```
# test_setに記載した順番で表示されるわけではありません。
test_set = {1, 2, 'b', 5, 3, 4}
print(test_set)
```

```
{1, 2, 3, 4, 5, 'b'}
```

演習1：セット（set）

- setでは値が一意に保持されることを確認してください。
- setでは順番が保証されないことを確認してください。

辞書（dictionary）とは

- 辞書もリストのように複数のデータを保持するものですが、リストと違うのは各要素を「キー」と「値」のペアで保持します。
- 辞書では「:」でキーと値を分けます。:の左側が要素のキー、右側が要素の値です。全体は集合と同じく波カッコ{}でくくります。

```
test_dic = {1:"potato", 2:"tomato", 3:"onion"}
print(test_dic)
```

```
{1: 'potato', 2: 'tomato', 3: 'onion'}
```

辞書（dictionary）とは

- 値には文字列や数値、リストなどPythonオブジェクトであればなんでも使えます。
- キーはイミュータブル（中身が変更できないタプルや文字列、数値など）が使えます。

```
# キーにリストを設定してみます。
test_dic_error = {[0, 1]:"potato", 2:"tomato", 3:"onion"}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-74db26fee7ce> in <module>
      1 # キーにリストを設定してみます。
----> 2 test_dic_error = {[0, 1]:"potato", 2:"tomato", 3:"onion"}

TypeError: unhashable type: 'list'
```

キーにミュータブル（中身が変更できるリストなど）を設定しようとすると、エラーが発生します。

辞書（dictionary）とは

- 辞書型もセット型と同じく中身の順序が保持されません。そのため、リストやタプルのように格納位置で要素を指定することはできません。辞書型ではキーを使って要素を指定します。
- 辞書の更新や削除なども、キーを指定することで実行できます。

```
test_dic = {1:"potato", 2:"tomato", 3:"onion"}
print(test_dic)
```

```
{1: 'potato', 2: 'tomato', 3: 'onion'}
```

```
print(test_dic[2])
```

```
tomato
```

```
test_dic[2] = "sold out"
print(test_dic)
```

```
{1: 'potato', 2: 'sold out', 3: 'onion'}
```

演習2：辞書（dictionary）

- 辞書型でデータを保持してください。
- キーにミュータブルな値を使い、エラーが出ることを確認してください。
- キーを指定して値を取り出す処理を実行してください。
- 要素を上書きしてください。

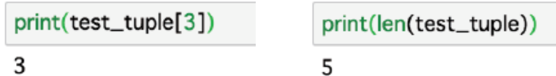
タプル（tuple）とは

- タプルはリストにそっくりなシーケンス型です。リストが四角カッコ記号[]でくくるのに対して、タプルは丸カッコ()でくくります。

```
test_tuple = (0, 1, 2, 3, 4)
print(test_tuple)
```

```
(0, 1, 2, 3, 4)
```

- リストと同様に、インデックスで値を参照したり、要素数を参照することができます。



タプル（tuple）とは

- タプルはリストと似ているところが多数ありますが、リストと違い一度作ると中身を変更することができません。

```
test_tuple[3] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-5cb1fb3466d0> in <module>
----> 1 test_tuple[3] = 5

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

- タプルは変更できませんが、リストより処理速度が速いという特徴があります。中身を変更しなくて良いリストを何度も参照するよりは、タプルのほうが計算が早くなります。

演習3：

- タプルでデータを保持してください。
- リストと同様に、インデックスを指定して値が参照できることを確認してください。
- リストと同様に要素数が参照できることを確認してください。
- タプルを変更しようとするとエラーが発生することを確認してください。

第6回：pythonによる基本的なフロー制御

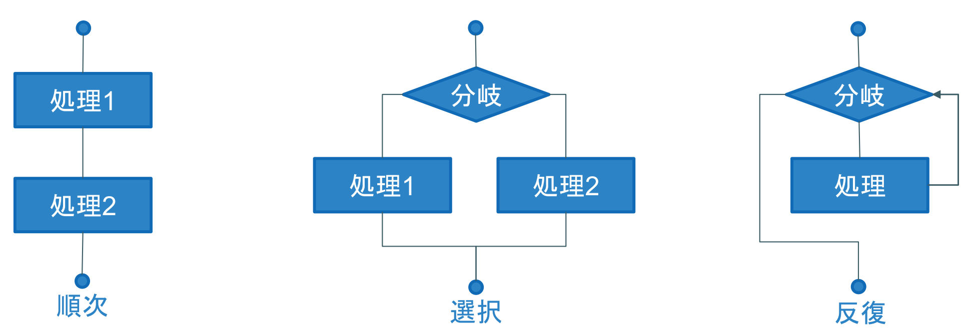
アジェンダ

- 構造化定理とは
- Pythonによる基本的なフロー制御

構造化定理とpythonによる基本的なフロー制御

構造化定理

- [Wikipediaより] 3つだけの手段で手続きを構築するならば、フローチャートがあらゆる計算可能関数を表現できることを述べている。3つだけの手段(制御構造)は、以下のものである。
  - ある一つの手続きを実行し、それからもう一つの手続きを実行する(順次)
  - ブール演算式の値に従って2つの手続きのどれか1つを実行する(選択)
  - ブール演算式が真である限り、手続きを繰り返し実行する(反復)



Pythonによる分岐処理：if文

- if文はプログラム中で条件分岐を行いたいときに使います。if文は与えられた条件が下記のいずれなのかを確認する処理です。
  - Yes（True：真）なのか
  - No（False：偽）なのか

- if文の基本的な構文は下記の通りです。

```
if [条件式]:
    [条件式がTrueのときに行う処理]
```

演習1：if文による分岐

- if文において条件を満たす場合に処理を実行するプログラムを実装してください。

Pythonによる分岐処理：if else文

- ここまでは条件式がTrueになるときの処理を学習しました。今回はTrueのときだけでなく、条件式がFalseになったときに動く処理も考慮します。

- If else文の基本的な構文は下記の通りです。

```
if [条件式]:
    [条件式がTrueのときに行う処理]
else:
    [条件式がFalseのときに行う処理]
```

演習2：if else文による分岐

- If else文において、条件を満たさない場合にも処理を実行するプログラムを実装してください。

Pythonによる分岐処理：if elif文

- 「if文の条件は満たさないが、別の条件を満たしたらelseとは別の処理がしたい。」という場合にif elif文を使います。

- If elif文の基本的な構文は下記の通りです。

```
if [条件式1]:
    [条件式がTrueのときに行う処理]
elif [条件式2]:
    [elifの条件式2がTrueのときに行う処理]
else:
    [if文の条件式1もelifの条件式2もどちらも
Falseのときに行う処理]
```

演習3：if elif文による分岐

- If elif文による分岐を実行するプログラムを実装してください。

Pythonによる分岐処理：条件演算子（三項演算子）

- Pythonには、if文を一行で記述できる条件演算子（三項演算子）と呼ばれる書き方があります。
- 条件演算子の基本的な構文は下記の通りです。

  条件式が真のときに評価される式 if 条件式 else 条件式が偽のときに評価される式

- 条件によって値を切り替える場合は、それぞれの値をそのまま記述します。

  条件式が真のときに返す値 if 条件式 else 条件式が偽のときに返す値

演習4：条件演算子による分岐

- 条件演算子による処理（文字列を返す）を実装してください。
- 条件演算子による処理（式を実行する）を実装してください。

Pythonによる反復処理：for文

- Pythonにおいて反復処理を実行する方法として、for文と呼ばれる書き方があります。
- For文の基本的な構文は下記の通りです。

```
for 変数 in オブジェクト:
    実行する処理
```

Pythonによる反復処理：for文とif文の組み合わせ

- 反復処理の中に分岐処理を実装すると、分岐処理を繰り返し実行することができます。

```
# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

# for文でループさせながら、判定対象の数値を1つずつ処理します。
for x in x_list:
  if x / 4 >= 3:# 「xを4で割った値が3以上か?」を判定します。
    print("3以上です。")
  elif x / 4 >= 2:# 「xを4で割った値が2以上か?」を判定します。
    print("2以上です。")
  else:
    print("その他です。")
```

```
その他です。
その他です。
その他です。
その他です。
その他です。
その他です。
その他です。
2以上です。
2以上です。
2以上です。
2以上です。
3以上です。
```

For文の中にif文を記載します。

演習5：for文による反復処理

- 条件演算子による処理（文字列を返す）を実装してください。
- 条件演算子による処理（式を実行する）を実装してください。

演習6：for文とif文の組み合わせ

- 分岐処理と反復処理を組み合わせて実装してください。。

Pythonによる反復処理：while文

- while文は指定した条件式が真の間は処理を繰り返し実行します。
- while文の基本的な構文は下記の通りです。

```
while 条件式:
    条件式が真の時に実行する文
```

Pythonによる反復処理：while文

- while文は指定した条件式が真の間は処理を繰り返し実行します。条件式がいつまでも真のままになる実装をしてしまうと、処理が永遠に続いてしまいます。

```
counter = 0

# カウンターが3未満の間は反復処理を実行します。
while counter < 3:
    print(counter)
    # カウンターをインクリメントします。
    counter += 1
```

```
0
1
2
```

ココでカウンターをインクリメントしないと、「counter < 3」が偽になることがないため、処理が永遠に続いてしまいます。

演習7：while文による反復処理

- while文による反復処理を実装してください。

第7回：pythonによる基本的なフロー制御2

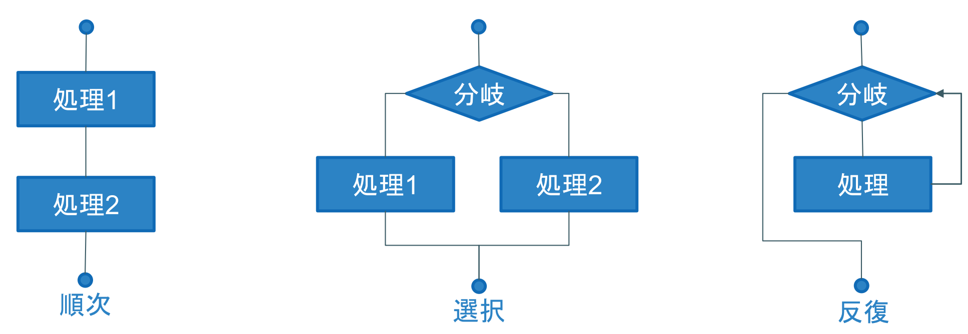
アジェンダ

- 前回までの講義の振り返り
- Pythonによるフロー制御

前回までの講義の振り返り

構造化定理

- [Wikipediaより] 3つだけの手段で手続きを構築するならば、フローチャートがあらゆる計算可能関数を表現できることを述べている。3つだけの手段（制御構造）は、以下のものである。
  - ある一つの手続きを実行し、それからもう一つの手続きを実行する（順次）
  - ブール演算式の値に従って2つの手続きのどれか1つを実行する（選択）
  - ブール演算式が真である限り、手続きを繰り返し実行する（反復）



Pythonによる分岐処理：if文

- if文はプログラム中で条件分岐を行いたいときに使います。if文は与えられた条件が下記のいずれなのかを確認する処理です。
  - Yes（True：真）なのか
  - No（False：偽）なのか

- if文の基本的な構文は下記の通りです。

```
if [条件式]:
    [条件式がTrueのときに行う処理]
```

Pythonによる分岐処理：条件演算子（三項演算子）

- Pythonには、if文を一行で記述できる条件演算子（三項演算子）と呼ばれる書き方があります。
- 条件演算子の基本的な構文は下記の通りです。

```
条件式が真のときに評価される式 if 条件式 else 条件式が偽のときに評価される式
```

- 条件によって値を切り替える場合は、それぞれの値をそのまま記述します。

```
条件式が真のときに返す値 if 条件式 else 条件式が偽のときに返す値
```

Pythonによる反復処理：for文

- Pythonにおいて反復処理を実行する方法として、for文と呼ばれる書き方があります。
- For文の基本的な構文は下記の通りです。

```
for 変数 in オブジェクト:
    実行する処理
```

Pythonによる反復処理：while文

- while文は指定した条件式が真の間は処理を繰り返し実行します。
- while文の基本的な構文は下記の通りです。

```
while 条件式:
    条件式が真の時に実行する文
```

pythonによるフロー制御

リスト内包表記

- 条件演算子とforループ、リストを組み合わせた「リスト内包表記」という書き方があります。
- For文のみを使用した場合に比べ、コーディング量を抑えることができます（複雑なアルゴリズムを記載すると可読性が低下するというデメリットもあります）。

```
# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

# for文でループさせながら、判定対象の数値を1つずつ処理します。
result_list = [x * 3 if x / 4 >= 3 else x * 2 for x in x_list]
print(result_list)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 36]
```

```
# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

# リスト内包表記を使用せず、同じロジックを実行します。
result_list = []
for x in x_list:
    if x / 4 >= 3:# 「xを4で割った値が3以上か?」を判定します。
        result_list.append(x * 3)
    else:
        result_list.append(x * 2)
print(result_list)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 36]
```

演習1：リスト内包表現

- 条件演算子とfor文を使用してリスト内包表記を実装してください。
- 同じロジックを、リスト内包表現を使用せずに実装してください。

反復処理の応用：break

- 反復処理を途中で終了したい場合にbreakを使用します。
- 通常は、反復処理を終了する条件を判定するif文と組み合わせて使用されます。

```
# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

for x in x_list:
    if x >= 6:# 「xが6以上か?」を判定します。
        print(str(x) + "で処理を中断しました。")
        break
    print(x)
```

```
1
2
3
4
5
6で処理を中断しました。
```

演習2：breakによる反復処理の中断

- breakを使用して反復処理を中断する処理を実装してください。

反復処理の応用：continue

- continueは、continueの後続処理をスキップし反復処理を継続することができます。

```
# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

for x in x_list:
    if x % 2 == 0:
        print("偶数です。", x)
        # continue以降のスキップされ反復処理が継続されます。
        continue
    print("奇数です。", x)
```

```
奇数です。 1
偶数です。 2
奇数です。 3
偶数です。 4
奇数です。 5
偶数です。 6
奇数です。 7
偶数です。 8
奇数です。 9
偶数です。 10
奇数です。 11
偶数です。 12
```

演習3：continueによる反復処理のスキップ

- 反復処理の中でcontinueを使用する処理を実装してください。

反復処理で使用される組み込み関数：range

- 反復数を指定した書き方ができます。
- 5回繰り返すfor文を書きたい場合は、以下のようにソースコードを記述します。これは5個の数値（0から始まる）を生成し、順番に読んでいきます。

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

```
0
1
2
3
4
```

反復処理で使用される組み込み関数：range

- その他にrangeは数値の生成数と開始値を指定することができます。
- 下記の例の場合、10個の数値（0から始まる）を生成し、6から順番に読み込んでいます。

```
for i in range(6, 10):
    print(i)
```

```
6
7
8
9
```

- rangeは数値の生成数と開始値以外に、増加値も指定することができます。

```
for i in range(0, 10, 3):
    print(i)
```

```
0
3
6
9
```

演習4：rangeを使用した反復処理

- rangeを使用した反復処理を実装してください。

演習5：rangeを使用した反復処理

- rangeの数値の生成数と開始値を指定した反復処理を実装してください。
- rangeの数値の生成数、開始値、増加値を指定した反復処理を実装してください。

反復処理とelseの組み合わせ

- 反復処理の後にelseを書くことができます。反復処理などの最中にbreakされなかった場合のみ、elseは実行されます。

```
# breakで反復処理を抜け場合は、elseは実行されない。

# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

for x in x_list:
    if x >= 6:# 「xが6以上か?」を判定します。
        break
    print(x)
else:
    print("処理が完了しました!")
```

```
1
2
3
4
5
```

```
# breakで反復処理を抜けなかった場合は、elseが実行される。

# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

for x in x_list:
    if x >= 15:# 「xが6以上か?」を判定します。
        break
    print(x)
else:
    print("処理を完了しました!")
```

```
1
2
3
4
5
6
7
8
9
10
11
12
処理を完了しました!
```

反復処理とelseの組み合わせ

- 下記のように、反復処理の後にelseを書くことによって実装量を大きく削減できる場合があります。

演習6：elseの実行条件

- 反復処理においてelseに記載された処理が実行される条件を確認してください（breakで反復処理を抜けるときはelseが実行されないことを確認してください）。

演習7：反復処理とelseを組み合わせた実装量の削減

- 反復処理とelseを組み合わせることで実装量が削減できることを確認してください。

第8回：関数を利用したプログラミング

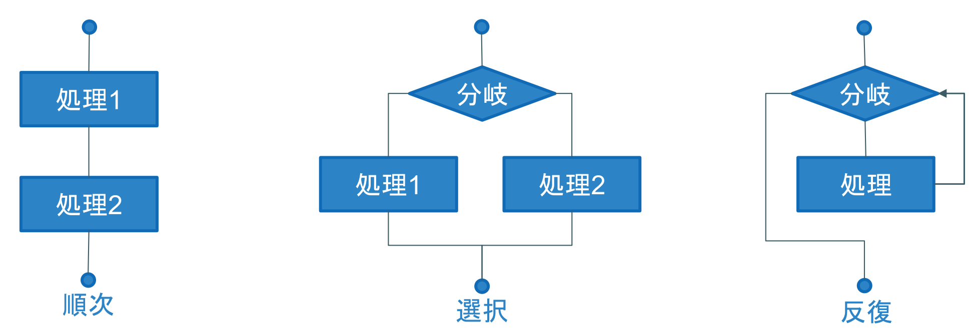
アジェンダ

- 前回までの講義の振り返り
- Pythonにおける関数

前回までの講義の振り返り

構造化定理

- [Wikipediaより] 3つだけの手段で手続きを構築するならば、フローチャートがあらゆる計算可能関数を表現できることを述べている。3つだけの手段(制御構造)は、以下のものである。
  - ある一つの手続きを実行し、それからもう一つの手続きを実行する(順次)
  - ブール演算式の値に従って2つの手続きのどれか1つを実行する(選択)
  - ブール演算式が真である限り、手続きを繰り返し実行する(反復)



Pythonによる分岐処理：if文

- if文はプログラム中で条件分岐を行いたいときに使います。if文は与えられた条件が下記のいずれなのかを確認する処理です。
  - Yes（True：真）なのか
  - No（False：偽）なのか

- if文の基本的な構文は下記の通りです。

```
if [条件式]:
    [条件式がTrueのときに行う処理]
```

Pythonによる分岐処理：条件演算子（三項演算子）

- Pythonには、if文を一行で記述できる条件演算子（三項演算子）と呼ばれる書き方があります。

- 条件演算子の基本的な構文は下記の通りです。

```
条件式が真のときに評価される式 if 条件式 else 条件式が偽のときに評価される式
```

- 条件によって値を切り替える場合は、それぞれの値をそのまま記述します。

```
条件式が真のときに返す値 if 条件式 else 条件式が偽のときに返す値
```

Pythonによる反復処理：for文

- Pythonにおいて反復処理を実行する方法として、for文と呼ばれる書き方があります。
- For文の基本的な構文は下記の通りです。

```
for 変数 in オブジェクト:
    実行する処理
```

Pythonによる反復処理：while文

- while文は指定した条件式が真の間は処理を繰り返し実行します。
- while文の基本的な構文は下記の通りです。

```
while 条件式:
    条件式が真の時に実行する文
```

リスト内包表記

- 条件演算子とforループ、リストを組み合わせた「リスト内包表記」という書き方があります。
- For文のみを使用した場合に比べ、コーディング量を抑えることができます（複雑なアルゴリズムを記載すると可読性が低下するというデメリットもあります）。

```
# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

# for文でループさせながら、判定対象の数値を1つずつ処理します。
result_list = [x * 3 if x / 4 >= 3 else x * 2 for x in x_list]
print(result_list)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 36]
```

```
# 判定対象の数値をリストにします。
x_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

# リスト内包表記を使用せず、同じロジックを実行します。
result_list = []
for x in x_list:
    if x / 4 >= 3:# 「xを4で割った値が3以上か?」を判定します。
        result_list.append(x * 3)
    else:
        result_list.append(x * 2)
print(result_list)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 36]
```

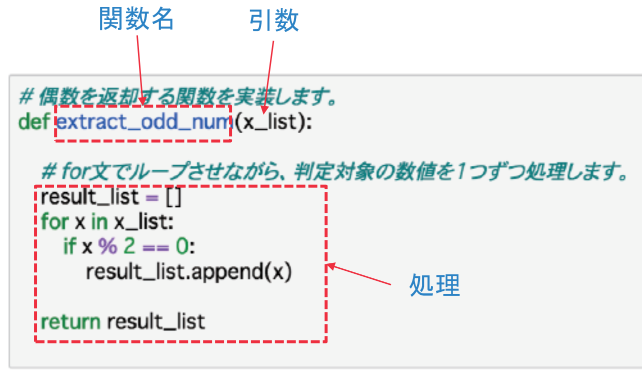
pythonにおける関数

関数とは

- 第7回までは組み込み関数（Pythonで予め用意された関数）について説明してきました。Pythonでは、独自のアルゴリズムを再利用可能な形で関数にできます。
- 関数は引数を与えることができます。また戻り値を設定することもできます。

関数の定義

- 関数は関数名、引数、処理を記載します。
- 引数は複数渡すことができます。



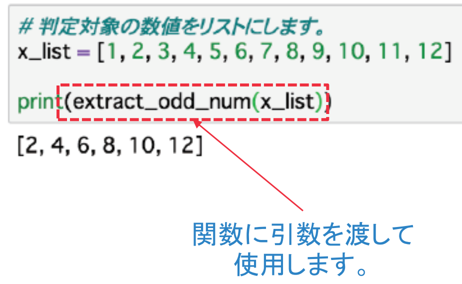

```
# 偶数を返却する関数を実装します。
def extract_odd_num(x_list):

    # for文でループさせながら、判定対象の数値を1つずつ処理します。
    result_list = []
    for x in x_list:
        if x % 2 == 0:
            result_list.append(x)

    return result_list
```

関数の呼び出し

- 定義した関数を実行する際は、引数を渡すようにします。
- 一度インスタンスを作成した関数は、引数を変えて何度でも実行できます。



演習1：関数の定義と呼び出し

- 数値リストを引数として渡し、偶数リストを返却する関数を実装してください。

演習2：複数の変数の引き渡し

- 要素数の同じ2つの数値リストの、同じ要素番号の数値を足す関数を実装してください。

キーワード引数

- 関数を呼び出す際、キーワードを指定して引数を渡すことができます。
- 関数の順番を気にせず引数を指定できるなど、便利な使い方ができます。

```
# 文字列を結合して返す関数を実装します。
def concat_strs(name, food):
    message = name + " likes " + food + " very much!"
    return message
```

```
print(concat_strs(food="steak", name="Taro"))
```

```
Taro likes steak very much!
```

演習3：キーワード引数の使い方

- 関数を呼び出す際に、キーワードを指定して引数を渡してください。

位置引数とキーワード引数の混在

- 位置を指定する引数とキーワード引数は混在させることができます。
- 混在させる場合、先に位置引数を記載する必要があります。

```
# キーワード引数を先に書くと、エラーが発生します。
print(concat_strs(food="steak", "Taro"))
```

```
  File "<ipython-input-31-7a3a90bf0fde>", line 2
    print(concat_strs(food="steak", "Taro"))
                                   ^
SyntaxError: positional argument follows keyword argument
```

```
# 位置引数を先に書くと、続いてキーワード引数を書くことができます。
print(concat_strs("Taro", food="steak"))
```

```
Taro likes steak very much!
```

演習4：位置引数とキーワード引数の混在

- 位置を指定した引数を先に記載し、キーワード引数は後に書く必要があることを確認してください。

引数の初期値

- 関数の中で引数の初期値を指定することができます。
- 関数呼び出し時に引数を指定すると、その値が優先されます。

```
# 文字列を結合して返す関数を実装します。
def concat_strs(name="Mike", food="pizza"):
    message = name + " likes " + food + " very much!"
    return message
```

```
# 引数をすべて省略します。
print(concat_strs())
# 引数を一部だけ指定します。
print(concat_strs(food="steak"))
# 位置引数とキーワード引数を混在させます。
print(concat_strs("Tommy", food="steak"))
```

```
Mike likes pizza very much!
Mike likes steak very much!
Tommy likes steak very much!
```

演習5：引数の初期値

- 関数の中で引数の初期値を指定し、その挙動を確認してください。

可変長引数

- 「*」を前において引数を定義すると、引数の個数を可変長とすることができます。
- 可変長の引数はタプルで渡されます。

```
# 文字列を結合して返す関数を実装します。
def concat_strs(name="Mike", *foods):
    message = name + " likes " + str(foods) + " very much!"
    return message
```

```
# 食べ物の名前を複数引数として渡します。
print(concat_strs("Tommy", "steak", "Sushi"))
```

```
Tommy likes ('steak', 'Sushi') very much!
```

演習6：可変長引数

- 可変長引数で変数を指定してください。

関数オブジェクト

- Pythonにおいて、関数も変数に格納してオブジェクトとして扱うことができます。

```
# 文字列を結合して返す関数を実装します。
def concat_strs(name="Mike", food="pizza"):
    message = name + " likes " + food + " very much!"
    return message
```

```
# 関数オブジェクトとして変数fに格納します。
f = concat_strs
# 関数オブジェクトを実行します。
print(f(food="steak"))
```

```
Mike likes steak very much!
```

演習7：関数オブジェクト

- 関数をオブジェクトとして実行してください。

第9回：クラスを利用したプログラミング

アジェンダ

- 前回までの講義の振り返り
- Pythonにおけるクラス

前回までの講義の振り返り

関数とは

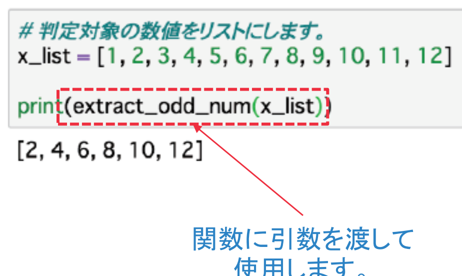
- 第7回までは組み込み関数（Pythonで予め用意された関数）について説明してきました。Pythonでは、独自のアルゴリズムを再利用可能な形で関数にできます。
- 関数は引数を与えることができます。また戻り値を設定することもできます。

関数の定義

- 関数は関数名、引数、処理を記載します。
- 引数は複数渡すことができます。

関数の呼び出し

- 定義した関数を実行する際は、引数を渡すようにします。
- 一度インスタンスを作成した関数は、引数を変えて何度でも実行できます。



キーワード引数

- 関数を呼び出す際、キーワードを指定して引数を渡すことができます。
- 関数の順番を気にせず引数を指定できるなど、便利な使い方ができます。

```
# 文字列を結合して返す関数を実装します。
def concat_strs(name, food):
    message = name + " likes " + food + " very much!"
    return message
```

```
print(concat_strs(food="steak", name="Taro"))
```

```
Taro likes steak very much!
```

位置引数とキーワード引数の混在

- 位置を指定する引数とキーワード引数は混在させることができます。
- 混在させる場合、先に位置引数を記載する必要があります。

```
# キーワード引数を先に書くと、エラーが発生します。
print(concat_strs(food="steak", "Taro"))
```

```
  File "<ipython-input-31-7a3a90bf0fde>", line 2
    print(concat_strs(food="steak", "Taro"))
                                   ^
SyntaxError: positional argument follows keyword argument
```

```
# 位置引数を先に書くと、続いてキーワード引数を書くことができます。
print(concat_strs("Taro", food="steak"))
```

```
Taro likes steak very much!
```

引数の初期値

- 関数の中で引数の初期値を指定することができます。
- 関数呼び出し時に引数を指定すると、その値が優先されます。

```
# 文字列を結合して返す関数を実装します。
def concat_strs(name="Mike", food="pizza"):
    message = name + " likes " + food + " very much!"
    return message
```

```
# 引数をすべて省略します。
print(concat_strs())
# 引数を一部だけ指定します。
print(concat_strs(food="steak"))
# 位置引数とキーワード引数を混在させます。
print(concat_strs("Tommy", food="steak"))
```

```
Mike likes pizza very much!
Mike likes steak very much!
Tommy likes steak very much!
```

可変長引数

- 「*」を前において引数を定義すると、引数の個数を可変長とすることができます。
- 可変長の引数はタプルで渡されます。

```
# 文字列を結合して返す関数を実装します。
def concat_strs(name="Mike", *foods):
    message = name + " likes " + str(foods) + " very much!"
    return message
```

```
# 食べ物の名前を複数引数として渡します。
print(concat_strs("Tommy", "steak", "Sushi"))
```

```
Tommy likes ('steak', 'Sushi') very much!
```

関数オブジェクト

- Pythonにおいて、関数も変数に格納してオブジェクトとして扱うことができます。

```
# 文字列を結合して返す関数を実装します。
def concat_strs(name="Mike", food="pizza"):
    message = name + " likes " + food + " very much!"
    return message
```

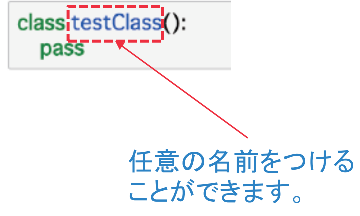
```
# 関数オブジェクトとして変数fに格納します。
f = concat_strs
# 関数オブジェクトを実行します。
print(f(food="steak"))
```

```
Mike likes steak very much!
```

pythonにおけるクラス

クラスとは

- クラスとは、オブジェクトを生成するうえで使われる″型″のようなもので、オブジェクトの定義です。クラスからインスタンスを生成し、処理を行うことができます。
- ″class″キーワードの後に任意のクラス名を記載し、その下にメソッドや変数を記載します。



コンストラクタ

- 「_init_」という名前の関数で定義します。コンストラクタは、インスタンスが作成されるときに実行される処理です。
- どのようにインスタンスを生成するか、どのようなデータを持たせるか、という定義を記載します。
- 引数に記載しているselfとは、インスタンス自身を指します。

```
class catClass():
    #コンストラクタの中でメッセージを出力します。
    def __init__(self):
        print("コンストラクタが実行されました。")
```

演習1：クラスの実装

- 任意の名前のクラスを実装してください。

演習2：コンストラクタの実装

- コンストラクタを実装し、動作を確認してください。

アトリビュート（属性）

- 属性とはオブジェクトに存在する変数やメソッドのことです。
- 名前と値を持っており、ディクショナリに似ています。例えば変数なら変数名と変数値を持っており、メソッドならメソッド名とメソッド本体を持っています。

```
class catClass():
    def __init__(self, message):
        self.name = message
```

「name」という名前の属性を
定義しています。

```
cat_instance = catClass("Tama")
print(cat_instance.name)
```

```
Tama
```

アトリビュートの動的な操作

- インスタンス作成後もアトリビュートは操作できます。

演習3：属性の実装

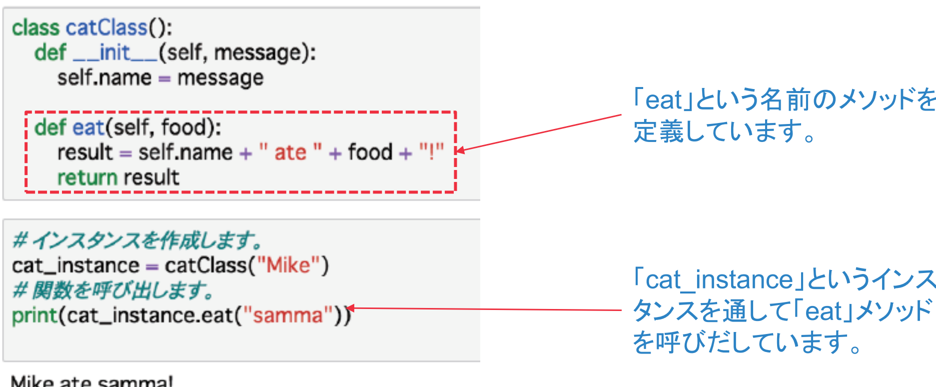
- nameという属性を定義し、インスタンス作成時にmessageを設定する処理を実装してください。

演習4：属性の動的な操作

- インスタンス作成後に属性を設定する処理を実装してください。

メソッド

- メソッドとは、クラス内に定義された関数のことです。メソッドは定義されたクラスからのみ呼び出すことが出来ます。



演習5：メソッドの実装

- クラス内でメソッドを実装し、クラスのインスタンスを経由して呼び出してください。

インスタンス変数とクラス変数

- インスタンス変数とは、それぞれのインスタンスごとに独立した変数です。例えば複数のインスタンスを生成した場合、それぞれのインスタンス変数は別のものとして扱われます。

```
class catClass():
    def __init__(self, message):
        self.name = message
```

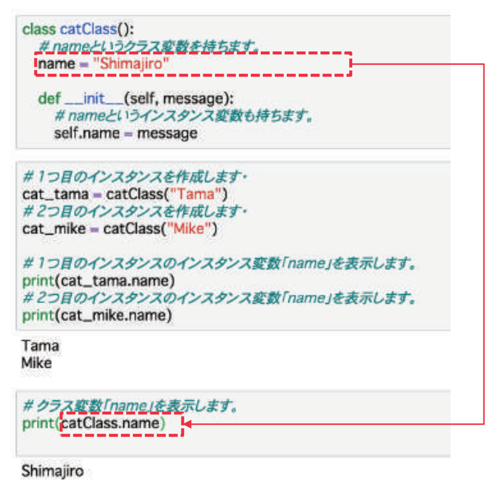
```
# 1つ目のインスタンスを作成します・
cat_tama = catClass("Tama")
# 2つ目のインスタンスを作成します・
cat_mike = catClass("Mike")

# 1つ目のインスタンスのインスタンス変数「name」を表示します。
print(cat_tama.name)
# 2つ目のインスタンスのインスタンス変数「name」を表示します。
print(cat_mike.name)
```

```
Tama
Mike
```

インスタンス変数とクラス変数

- クラス変数は全てのインスタンス間で共通した値を持ちます。クラス変数はインスタンスを生成することなく参照することができます。



クラスから直接クラス変数を
参照しています。

演習6：インスタンス変数の参照

- インスタンス変数がインスタンスごとに違う値を保持することを確認してください。

演習7：クラス変数の参照

- クラス変数とインスタンス変数へのアクセスを実行してください。

インスタンスメソッド/クラスメソッド/スタティックメソッド

- メソッドにはインスタンスメソッド、クラスメソッド、スタティックメソッドの3つがあります。
- 引数の与え方、デコレータの書き方がそれぞれ異なります。

メソッドの種類	説明	第1引数
インスタンスメソッド	インスタンスのデータを使って処理を実行する。	呼び出しに使ったインスタンスがselfに設定される。
クラスメソッド	クラスに関連した処理を実行する。	呼び出しに使ったクラスがclsに設定される。
スタティックメソッド	クラス、インスタンスとは関係のない処理を実行する。	クラスやインスタンスは設定されない。

演習8：インスタンスメソッド、クラスメソッド、スタティックメソッドの呼び出し

- インスタンスメソッド、クラスメソッド、スタティックメソッドを実装してください。

第10回：モジュールを利用したプログラミング

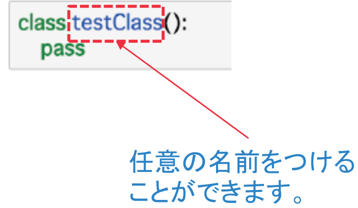
アジェンダ

- 前回までの講義の振り返り
- Pythonにおけるモジュール
- モジュールの作成

前回までの講義の振り返り

クラスとは

- クラスとは、オブジェクトを生成するうえで使われる”型”のようなもので、オブジェクトの定義です。クラスからインスタンスを生成し、処理を行うことができます。
- ”class”キーワードの後に任意のクラス名を記載し、その下にメソッドや変数を記載します。



コンストラクタ

- 「_init_」という名前の関数で定義します。コンストラクタは、インスタンスが作成されるときに実行される処理です。
- どのようにインスタンスを生成するか、どのようなデータを持たせるか、という定義を記載します。
- 引数に記載しているselfとは、インスタンス自身を指します。

```
class catClass():
    # コンストラクタの中でメッセージを出力します。
    def __init__(self):
        print("コンストラクタが実行されました。")
```

アトリビュート（属性）

- 属性とはオブジェクトに存在する変数やメソッドのことです。
- 名前と値を持っており、ディクショナリに似ています。例えば変数なら変数名と変数値を持っており、メソッドならメソッド名とメソッド本体を持っています。

```
class catClass():
    def __init__(self, message):
        self.name = message
```

「name」という名前の属性を
定義しています。

```
cat_instance = catClass("Tama")
print(cat_instance.name)
```

```
Tama
```

アトリビュートの動的な操作

- インスタンス作成後もアトリビュートは操作できます。

メソッド

- メソッドとは、クラス内に定義された関数のことです。メソッドは定義されたクラスからのみ呼び出すことが出来ます。

```
class catClass():
    def __init__(self, message):
        self.name = message

    def eat(self, food):
        result = self.name + " ate " + food + "!"
        return result
```

「eat」という名前のメソッドを定義しています。

```
# インスタンスを作成します。
cat_instance = catClass("Mike")
# 関数を呼び出します。
print(cat_instance.eat("samma"))
```

「cat_instance」というインスタンスを通して「eat」メソッドを呼びだしています。

```
Mike ate samma!
```

インスタンス変数とクラス変数

- インスタンス変数とは、それぞれのインスタンスごとに独立した変数です。例えば複数のインスタンスを生成した場合、それぞれのインスタンス変数は別のものとして扱われます。

```
class catClass():
    def __init__(self, message):
        self.name = message
```

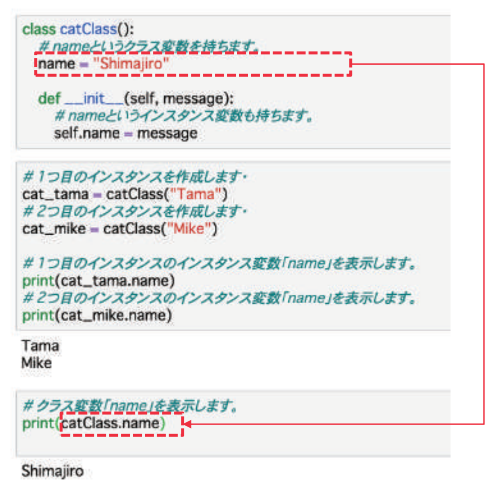
```
# 1つ目のインスタンスを作成します・
cat_tama = catClass("Tama")
# 2つ目のインスタンスを作成します・
cat_mike = catClass("Mike")

# 1つ目のインスタンスのインスタンス変数「name」を表示します。
print(cat_tama.name)
# 2つ目のインスタンスのインスタンス変数「name」を表示します。
print(cat_mike.name)
```

```
Tama
Mike
```

インスタンス変数とクラス変数

- クラス変数は全てのインスタンス間で共通した値を持ちます。クラス変数はインスタンスを生成することなく参照することができます。



クラスから直接クラス変数を
参照しています。

インスタンスメソッド/クラスメソッド/スタティックメソッド

- メソッドにはインスタンスメソッド、クラスメソッド、スタティックメソッドの3つがあります。
- 引数の与え方、デコレータの書き方がそれぞれ異なります。

メソッドの種類	説明	第1引数
インスタンスメソッド	インスタンスのデータを使って処理を実行する。	呼び出しに使ったインスタンスがselfに設定される。
クラスメソッド	クラスに関連した処理を実行する。	呼び出しに使ったクラスがclsに設定される。
スタティックメソッド	クラス、インスタンスとは関係のない処理を実行する。	クラスやインスタンスは設定されない。

pythonにおけるモジュール

モジュールとは

- 何度も利用する関数、その関数と関連のある関数など、関連性のあるプログラムの部品を1つのファイルにまとめたものをモジュールといいます。モジュールにすることで、下記のメリットが得られます。
  - 頻繁に利用するコードを1つにまとめる事ができ、コードのメンテナンスが楽になる。
  - 関連するコードを一まとまりにできるので、コードの意図がわかりやすくなる。
  - まとめたものを関数とすれば、他のプログラムから利用できるようになる。

モジュールのインポート

- まずは標準ライブラリの使い方を例に取ります。
- 「 import [モジュール名] 」と記載することで、使用したいモジュールをインポートすることができます。
- モジュールのインポート後は、モジュールに含まれる関数などを参照することができます。

```
# randomモジュールをインポートします。
import random
```

```
# 1から5の間でランダムに1つの整数を表示します。
print(random.randint(1, 5))
```

```
5
```

指定した名前でモジュールをインポート

- 「 import [モジュール名] as [任意の名前] 」と記載することで、モジュールに任意の名前をつけてインポートすることができます。
- モジュール名が長過ぎる際に省略したり、プログラミング中でモジュールに何かしらの意味を持たせたい場合などに名前をつけます。

```
# randomモジュールをインポートします。
import random as rnd
```

```
# 1から5の間でランダムに1つの整数を表示します。
print(rnd.randint(1, 5))
```

```
2
```

モジュールから特定の関数だけをインポート

- 「 from [モジュール名] import [関数名] 」と記載することで、モジュールに含まれる任意の関数のみをインポートすることができます。
- 関数を指定することで、不要な関数のインポートを避けることができます。

```
# randomモジュールからrandintのみをインポートします。
from random import randint
```

```
# 1から5の間でランダムに1つの整数を表示します。
print(randint(1, 5))
```

```
4
```

演習1：モジュールのインポート

- randomモジュール（疑似乱数を生成する）をインポートし、任意の関数を使用してください。

演習2：任意の名前によるモジュールのインポート

- randomモジュール（疑似乱数を生成する）をインポートする際に、任意の名前をつけてください。

演習3：任意の関数のみをインポート

- randomモジュール（疑似乱数を生成する）からrandint関数のみをインポートしてください。

モジュールの作成

モジュールの作成

- モジュールは自作することができます。Jupyter notebookによるモジュールの作成方法と利用方法について説明します。

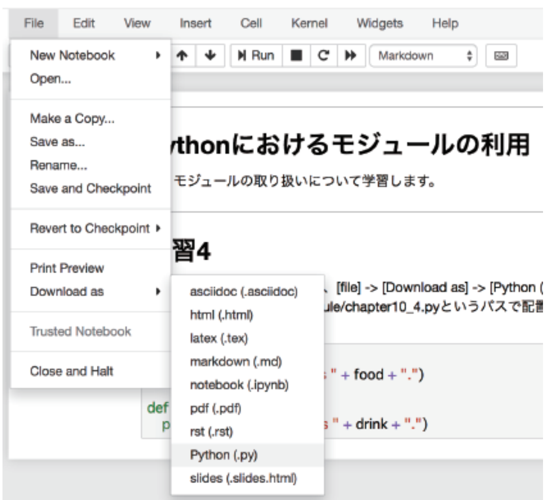
- まずは関数を実装します。

```
def eat(food):
    print("My favorite food is " + food + ".")

def drink(drink):
    print("My favorite drink is " + drink + ".")
```

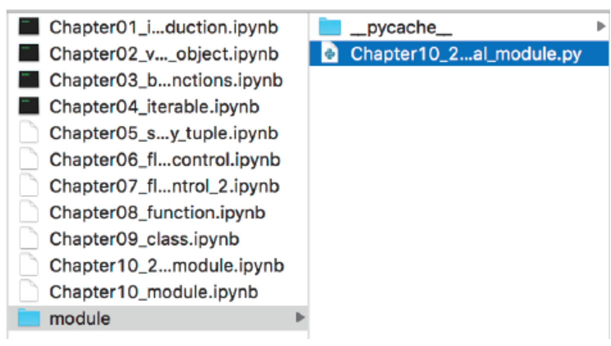
モジュールの作成

- 任意の関数を実装した後、[file] -> [Download as] -> [Python (.py)]でPythonファイルを出力してください。



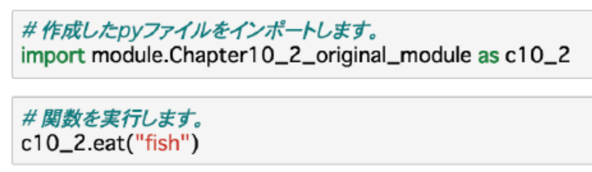
モジュールの作成

- 出力したファイルをmodule/Chapter10_2_original_module.pyというパスで配置してください。（※pyファイルの名前は任意です。）



モジュールの作成

- 使用元のプログラムにおいて作成したモジュールインポートし、関数を参照します。



演習4：モジュールの作成

- モジュールを作成し、外部のプログラムから関数を参照してください。

パッケージ

- 複数のモジュールを1つのフォルダにまとめて管理することができます。このように1つのまとまりで管理するモジュール群をパッケージといいます。

Chapter01_i...duction.ipynb
Chapter02_v..._object.ipynb
Chapter03_b...nctions.ipynb
Chapter04_iterable.ipynb
Chapter05_s...y_tuple.ipynb
Chapter06_fl...control.ipynb
Chapter07_fl...ntrol_2.ipynb
Chapter08_function.ipynb
Chapter09_class.ipynb
Chapter10_2...module.ipynb
Chapter10_3...module.ipynb
Chapter10_module.ipynb
module

__pycache__
Chapter10_2_original_module.py
Chapter10_3_original_module.py

パッケージ内のモジュールのインポート

- 「.」(ドット)でパッケージ名とモジュール名をつないでインポート文を記載します。

```
# 作成したpyファイルをインポートします。
import module.Chapter10_2_original_module as c10_2
import module.Chapter10_3_original_module as c10_3
```

```
# 1つめのモジュールの関数を実行します。
c10_2.drink("beer")
c10_3.hobby("walking")
```

```
My favorite drink is beer.
My hobby is walking.
```

演習5：パッケージ内のモジュールの参照

- 自作パッケージ内の2つのモジュールの関数を参照してください。

第11回：継承/名前空間とスコープ/例外処理

アジェンダ

- 前回までの講義の振り返り
- 継承
- 名前空間とスコープ
- 例外処理

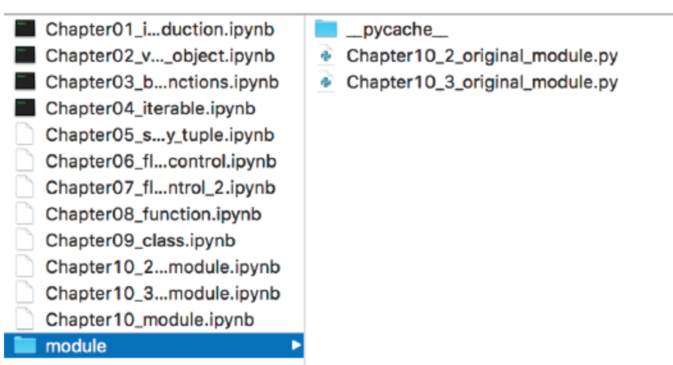
前回までの講義の振り返り

モジュールとは

- 何度も利用する関数、その関数と関連のある関数など、関連性のあるプログラムの部品を1つのファイルにまとめたものをモジュールといいます。モジュールにすることで、下記のメリットが得られます。
    - 頻繁に利用するコードを1つにまとめる事ができ、コードのメンテナンスが楽になる。
    - 関連するコードを一まとまりにできるので、コードの意図がわかりやすくなる。
    - まとめたものを関数とすれば、他のプログラムから利用できるようになる。

パッケージ

- 複数のモジュールを1つのフォルダにまとめて管理することができます。このように1つのまとまりで管理するモジュール群をパッケージといいます。



継承とは

- 既存クラスの変数、メソッドを利用し、追加や変更したい部分だけを定義して新しいクラスを作ることを継承といいます。
- 継承されるクラスのことを親クラス（スーパークラス/基底クラス）といいます。
- 親クラスを継承して作ったクラスのことを子クラス（サブクラス/派生クラス）といいます。

継承とは

- 子クラスの「()」の中に親クラスのクラス名を記載すると、子クラスにおいて親クラスを継承することができます。

```
# 親クラスを作成します。
class catParent():
    def __init__(self, message):
        self.name = message
        print("My name is " + self.name + ".")

    def eat(self, food):
        result = self.name + " ate " + food + "!"
        return result
```

```
# 親クラスを継承して子クラスを作成します。
class catChild(catParent):
    pass
```

```
# 親クラスの挙動を確認します。
catParent("Tama")
```

```
My name is Tama.
<__main__.catParent at 0x108b2b9b0>
```

```
# 子クラスの挙動を確認します。
catChild("Mike")
```

```
My name is Mike.
<__main__.catChild at 0x108b2bf28>
```

継承とは

- 子クラスの中では、独自の実装をすることができます。

```
# 親クラスを作成します。
class catParent():
    def __init__(self, message):
        self.name = message
        print("My name is " + self.name + ".")

    def eat(self, food):
        result = self.name + " ate " + food + "!"
        return result
```

```
# 親クラスを継承して子クラスを作成します。
class catChild(catParent):
    # 子クラス独自のメソッドを実装します。
    def walk(self, message):
        print("I am walking around " + message + "!")
```

```
# 子クラスのインスタンスを作成します。
cat_child = catChild("Tama")
# 子クラスのメソッドを参照します。
cat_child.walk("sea park")
```

```
My name is Tama.
I am walking around sea park!
```

演習1：親クラスの継承

- 親クラスを継承した子クラスを作成してください。

演習2：子クラスの実装

- 親クラスを継承した子クラスにおいて、独自のメソッドを実装してください。

オーバーライド

- 親クラスに実装されたメソッドを、子クラスにおいて上書きすることができます。
- 子クラスにおいて、親クラスのメソッド名と同じ名前でメソッドを実装すると、右図のようにメソッドを上書きすることができます。

```
# 親クラスを作成します。
class catParent():
    def __init__(self, message):
        self.name = message
        print("My name is " + self.name + ".")

    def eat(self, food):
        print(self.name + " ate " + food + "!")
```

```
# 親クラスを継承して子クラスを作成します。
class catChild(catParent):
    # 親クラスのメソッド「eat」を上書きします。
    def eat(self, food):
        print(self.name + " has eaten " + food + "!")
```

```
# 親クラスのeatを参照します。
cat_parent = catParent("Tama")
cat_parent.eat("fish")
```

```
My name is Tama.
Tama ate fish!
```

```
# 子クラスのeatを参照します。
cat_child = catChild("Mike")
cat_child.eat("fish")
```

```
My name is Mike.
Mike has eaten fish!
```

演習3：オーバーライド

- 子クラスにおいて親クラスのメソッドを上書きしてください。

親クラスのメソッドの参照

- 子クラスの中で[super().親クラスのメソッド名]と記載すると、子クラスから親クラスのメソッドを参照することができます。

```
# 親クラスを作成します。
class catParent():
    def __init__(self, message):
        self.name = message
        print("My name is " + self.name + ".")

    def eat(self, food):
        print(self.name + " ate " + food + "!")
```

```
# 親クラスを継承して子クラスを作成します。
class catChild(catParent):
    def action(self, food):
        # 親クラスのメソッド「eat」を参照します。
        super().eat(food)
```

```
# 子クラスのactionを通して、親クラスのeatを参照します。
cat_child = catChild("Mike")
cat_child.action("fish")
```

```
My name is Mike.
Mike ate fish!
```

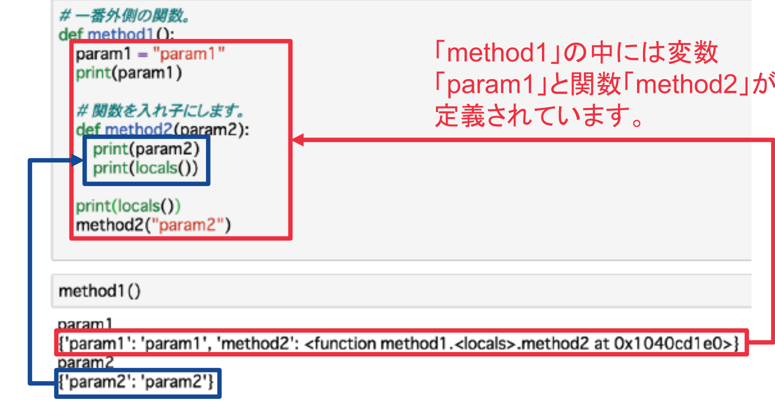
演習4：親クラスのメソッドの参照

- 子クラスにおいて親クラスのメソッドを上書きしてください。

名前空間とスコープ

名前空間とは

- 他の名前（変数や関数の名前）と重複せず、単一の名前として使うことができる範囲のことを名前空間といいます。



演習5：名前空間

- 関数の中に関数や変数を定義し、名前空間の範囲を確認してください。

スコープとは

- スコープとは、変数が有効な範囲のことをいいます。
- Pythonのスコープは4つに分けられます。それぞれLocal scope、Enclosing (function 's) scope、Global Scope、Built-in scopeといいます。

Local scope

- ローカルスコープが指す範囲は関数の中です。変数がローカルスコープに属する条件は「変数が関数内で定義された」場合のみです。

```
def method3(a):
    b = 2

    # 空の関数を定義します。
    def f():
        pass

    # 空のクラスを定義します。
    class g:
        pass

    # モジュールのインポート
    import os as h

    print(locals())
```

```
# method3を実行します。
method3(1)
```

```
{'a': 1, 'b': 2, 'f': <function method3.<locals>.f at 0x111ec90d0>, 'g': <class '__main__.method3.<locals>.g'>, 'h': <module 'os' from '//anaco
nda3/lib/python3.7/os.py'>}
```

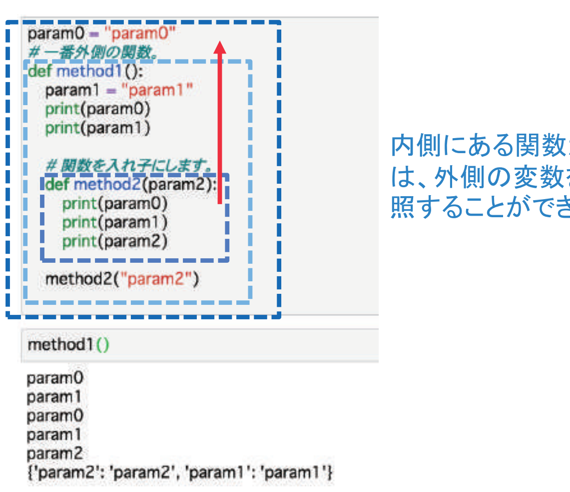
Global scope

- グローバル（モジュール）スコープは、最上位のスコープであり、ファイル（モジュール）内からであればどこからでもアクセスできます。

「method3」の内外から「param3」を参照できます。

Enclosing (function's) scope

- 関数の中に関数が定義されている場合、より内側のスコープから外側の変数が参照できます。



内側にある関数からは、外側の変数を参照することができます。

Built-in scope

- このスコープに属する「int」、「str」、「len」、「range」などの関数は、明示的に定義されていなくとも、どこからでも使うことができます。

演習6：ローカルスコープ

- 任意の関数を実装し、関数内のローカルスコープを確認してください。

演習7：グローバルスコープ

- グローバルスコープとローカルスコープの参照できる範囲の違いを確認してください。

演習8：Enclosing (function‘s) scope

- 入れ子になった関数で定義されている変数の参照範囲を確認してください。

例外処理

Pythonにおける例外処理

- Pythonで例外（実行中に検出されたエラー）をキャッチした際に実行する処理にはtryやexceptを使います。
- 例外が発生しても途中で終了させずに処理を継続させることもできます。
- また、elseやfinallyを使うことで終了時の処理を設定することも可能です。

例外処理：try-except

- [except 例外型]と実装することで、発生した例外をキャッチすることができます。

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print('Error')
```

```
Error
```

- [except 例外型 as 変数名]と実装することで、例外を変数に格納して取りまわることができます。

```
try:
    print(10 / 0)
except ZeroDivisionError as e:
    print(e)
    print(type(e))
```

```
division by zero
<class 'ZeroDivisionError'>
```

複数の例外の取り扱い

- 「except」節は複数実装することができます。それぞれ違う例外に対しての処理を実装します。

```
def divide_each(param_a, param_b):
    try:
        print(param_a / param_b)
    except ZeroDivisionError as e: # ここでは0で除算してしまったときに発生する例外をキャッチします。
        print(e)
    except TypeError as e: # ここでは数値計算に適していない型を検出したときに発生する例外をキャッチします。
        print(e)
```

```
# 0で除算した場合。
divide_each(10, 0)
```

```
division by zero
```

```
# 文字列型を使用した場合。
divide_each('10', '0')
```

```
unsupported operand type(s) for /: 'str' and 'str'
```

複数の例外の取り扱い

- 1つのexcept節の中で、複数の例外をタプルで実装することもできます。

```
def divide_each(param_a, param_b):
    try:
        print(param_a / param_b)
    except (ZeroDivisionError, TypeError) as e: # ここで2つの例外をキャッチします。
        print(e)
```

```
# 0で除算した場合。
divide_each(10, 0)
```

```
division by zero
```

```
# 文字列型を使用した場合。
divide_each('10', '0')
```

```
unsupported operand type(s) for /: 'str' and 'str'
```

演習9：try-exceptを使用した例外処理

- tryとexceptを使って例外をキャッチするプログラムを実装してください。
- 例外を変数に格納して取りまわせることも確認してください。

演習10：複数の例外の取り扱い

- 複数の例外をキャッチするプログラムを実装してください。
- 1つのexcept節にタプルで複数の例外を実装できることも確認してください。

正常終了時の処理else

- (try節で例外が発生せず)正常終了した場合に実行する処理をelse節に指定できます。
- 例外が発生してexceptでキャッチした場合は、else節の処理は実行されません。

```
def divide_each(param_a, param_b):
    try:
        print(param_a / param_b)
    except (ZeroDivisionError, TypeError) as e: # ここで2つの例外をキャッチします。
        print(e)
    else:
        print("finish!!")
```

```
# 正常終了した場合。
divide_each(10, 2)
```

```
5.0
finish!!
```

```
# 0で除算した場合。
divide_each(10, 0)
```

```
division by zero
```

演習11：正常終了時の処理

- 正常終了時の処理をelse節に実装してください。

終了時に実行する処理

- 例外発生の有無によらず、最後に実行させせたい処理は「finally」節に実装することができます。
- 「else」と「finally」は併用することができます。

```
def divide_each_else_finally(param_a, param_b):
    try:
        print(param_a / param_b)
    except (ZeroDivisionError, TypeError) as e: # ここで2つの例外をキャッチします。
        print(e)
    else:
        print("no error!")
    finally:
        print("finish!!")
```

```
# 正常終了した場合。
divide_each_else_finally(10, 2)
```

```
5.0
no error!
finish!!
```

```
# 0で除算した場合。
divide_each_else_finally(10, 0)
```

```
division by zero
finish!!
```

演習12：終了時に実行する処理

- finally節を実装し挙動を確認してください。
- elseとfinallyをともに実装し、挙動を確認してください。

例外の無視

- 「except」節に「pass」と記載すると、例外をキャッチした場合でも無視して処理を続行することができます。
- バグの温床になるため、仕様として明確な理由がない限り推奨できない実装方法です。

```
def divide_each_ignore(param_a, param_b):
    try:
        print(param_a / param_b)
    except (ZeroDivisionError, TypeError) as e: # ここで2つの例外をキャッチします。
        # 例外をキャッチしても無視します。
        pass
    print("try-except処理が終了しました。")
```

```
# 0で除算した場合。
divide_each_ignore(10, 0)
```

try-except処理が終了しました。

```
# 文字列型を使用した場合。
divide_each_ignore('10', '0')
```

try-except処理が終了しました。

演習13：例外の無視

- passを使用して例外を無視するプログラムを実装してください。

第12回：標準ライブラリ

アジェンダ

- 標準ライブラリとは
- 標準ライブラリによるシステム操作
- 標準ライブラリによる数値計算
- 文字列の取り扱い

標準ライブラリとは

- Pythonではprint()やlen()などのような組み込み関数の他に、標準ライブラリと呼ばれるモジュール群が付属しています。
- モジュールとは一連の関連した関数を埋め込んだPythonプログラムのことです。

標準ライブラリの種類

- 標準ライブラリを使用するとテキスト処理、バイナリデータ処理、日付型データ処理など様々な操作が実行できます。

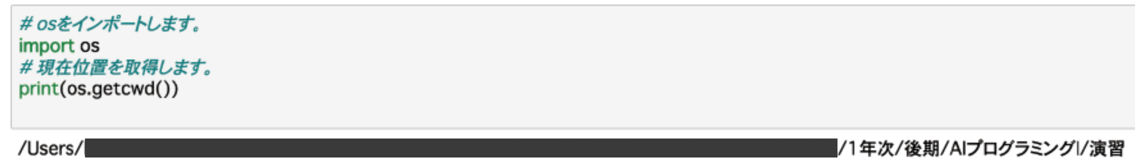
Python標準ライブラリの例: https://docs.python.org/ja/3/library/index.html

- テキスト処理サービス
  - string --- 一般的な文字列操作
  - re --- 正規表現操作
  - difflib --- 差分の計算を助ける
  - textwrap --- テキストの折り返しと詰め込み
  - unicodedata --- Unicode データベース
  - stringprep --- インターネットのための文字列調製
  - readline --- GNU readline のインタフェース
  - rlcompleter --- GNU readline向け補完関数
- バイナリデータ処理
  - struct --- バイト列をパックされたバイナリデータとして解釈する
  - codecs --- codec レジストリと基底クラス
- データ型
  - datetime --- 基本的な日付型および時間型
  - calendar --- 一般的なカレンダーに関する関数群
  - collections --- コンテナデータ型
  - collections.abc --- コレクションの抽象基底クラス
  - heapq --- ヒープキューアルゴリズム
  - bisect --- 配列二分法アルゴリズム
  - array --- 効率のよい数値アレイ
  - weakref --- 弱参照
  - types --- 動的な型生成と組み込み型に対する名前
  - copy --- 浅いコピーおよび深いコピー操作
  - pprint --- データ出力の整然化
  - reprlib --- もう一つの repr() の実装
  - enum --- 列挙型のサポート

標準ライブラリによるシステム操作

ファイルパスの取得

- 標準ライブラリosを使用すると、ファイルパスを取得することができます。



フォルダの作成

- 標準ライブラリosを使用すると、フォルダを作成することができます。

現在位置の移動

- 標準ライブラリosを使用すると、現在位置を移動することができます。

```
# osをインポートします。
import os
# 新しいフォルダに移動します。
os.chdir("./chapter12_test_folder")
```

```
# 現在位置を取得します。
print(os.getcwd())
```

```
/Users/[redacted]/AIプログラミングI/演習/chapter12_t
est_folder
```

ファイルのコピー、移動

- 標準ライブラリshutilを使用すると、ファイルコピーや移動ができます。

```
# shutilをインポートします。
import shutil
# 現在位置を取得します。
os.getcwd()

# フォルダを2つ作成します。
os.system("mkdir folder1")
os.system("mkdir folder2")

# folder1に移動します。
os.chdir("./folder1")
# test.txtを作成します。
os.system("echo test, test, test > 'test.txt'")

# folder1内でtest.txtをコピーしてtest2.txtを作成します。
shutil.copyfile('test.txt', 'test2.txt')
# test2.txtをfolder2に移動します。
shutil.move("test2.txt",'../folder2')
```

ファイルの検索

- 標準ライブラリglobを使用すると、ファイルのワイルドカード検索が実施できます。

```
# globをインポートします。
import glob
import shutil

# ファイルをコピーします。
shutil.copyfile('test.txt', 'test3.txt')

# ファイルをワイルドカード検索します。
glob.glob("*.txt")
```

```
['test3.txt', 'test.txt']
```

演習1：ファイルパスの取得

- 標準ライブラリosを使用して、本jupyter notebookの位置（ファイルのパス）を取得してください。

演習2：フォルダの作成

- 標準ライブラリosを使用して、本jupyter notebookが格納されているフォルダに新しいフォルダを作成してください。

演習3：現在位置の移動

- 標準ライブラリosを使用して、新しく作成したフォルダに移動してください。

演習4：ファイルコピー、ファイル移動

- 標準ライブラリshutilを使用して、ファイルのコピー、移動を実施してください。

演習5：ファイル検索

- 標準ライブラリglobを使用して、ファイルのワイルドカード検索を実施してください。

標準ライブラリによる数値計算

数論、数表現

- 標準ライブラリmathを使用して、数論および数表現の関数を実行することができます。

```
import math

# 切り上げ
print(math.ceil(1.01))
# 切り捨て
print(math.floor(1.01))
# 四捨五入:mathの関数ではないが参考として
print(round(1.01, 1))
# 絶対値
print(math.fabs(-1.01))
# 階乗
print(math.factorial(5))
# 割り算の余り
print(math.fmod(5, 2))
# 整数部
print(math.trunc(1.01))
```

```
2
1
1.0
1.01
120
1.0
1
```

指数関数、対数関数

- 標準ライブラリmathを使用して、指数関数と対数関数を実行することができます。

```
import math

# e = 2.718281... を自然対数の底としたeのx乗
print(math.exp(2))
# eのx乗から1を引いた値
print(math.expm1(2))
# xの(eを底とする)自然対数
print(math.log(2))
# xの(eを底とする)自然対数
print(math.log2(2))
# xの10を底とした対数(常用対数)
print(math.log10(100))
# xのy乗
print(math.pow(2, 3))
# xの平方根
print(math.sqrt(4))
```

```
7.38905609893065
6.38905609893065
0.6931471805599453
1.0
2.0
8.0
2.0
```

三角関数、角度変換

- 標準ライブラリmathを使用して、三角関数や角度変換を実行することができます。

```
import math

# 角xをラジアンから度に変換します。
print(math.degrees(math.pi / 2))
# 角xを度からラジアンに変換します。
print(math.radians(90))

# xラジアンの正弦
print(math.sin(math.pi / 2))
# xラジアンの余弦
print(math.cos(math.pi))
# xラジアンの正接
print(math.tan(math.pi / 4))
```

```
90.0
1.5707963267948966
1.0
-1.0
0.9999999999999999
```

演習6：数論、数表現

- 標準ライブラリmathを使用した数論および数表現の関数を実行してください。

演習7：指数関数、対数関数

- 標準ライブラリmathを使用した指数関数と対数関数を実行してください。

演習8：三角関数、角度変換

- 標準ライブラリmathを使用した三角関数、角度変換を実行してください。

文字列の取り扱い

Pythonにおける文字列

- 文字列は、リストのように各文字を要素とする文字の集合として扱うことができます。
- リストは要素数をカウントすることができますが、同様に文字列でも文字数をカウントすることができます。

```
message = "今日は良い天気です"
# 文字数を表示します。
print(len(message))
```

```
9
```

任意の文字の抽出

- 文字列のインデックスを使用することにより、任意の文字列を抽出することができます。
- また、開始/終了文字のインデックスを使用することにより、任意の範囲の文字列を抽出することもできます。

```
# 左から2つ目(インデックスは0から始まります)の文字を抽出します。
print(message[1])
```

```
日
```

```
# 右から1つ目(左端のインデックスは-1から始まり、左に向かって-1、-2とマイナス値がインクリメントされます)の文字を抽出します。
print(message[-1])
```

```
す
```

- 文字列から任意の文字をスライスしてください。

```
# 開始文字のインデックス、終了文字のインデックス+1を指定します。
print(message[1:5])
```

```
日は良い
```

演習9：文字列の文字数

- 文字列の文字数を確認してください。

演習10：任意の文字の抽出

- インデックスを指定して、文字列の任意の場所の文字のみ抽出してください。
- 文字列から任意の文字をスライスしてください。

文字列の結合

- 文字列リテラルを並べて書くと、文字列を結合することができます。
- 変数と文字列リテラルを結合するときは、「+」を使用する必要があります。

```
# 文字列リテラルを並べて書くと、文字列を結合することができます。
message = "今日は""雨です。"
print(message)
```

```
今日は雨です。
```

```
message1 = "今日は"
# 変数に格納した文字列と文字列リテラルを並べて書くと、エラーが発生します。
message = message1"曇りです。"
```

```
  File "<ipython-input-59-8f648b6c4236>", line 3
    message = message1"曇りです。"
                      ^
SyntaxError: invalid syntax
```

```
message1 = "今日は"
# 変数に格納した文字列と文字列リテラルを結合するときは、「+」でつなぎます。
message = message1 + "曇りです。"
print(message)
```

```
今日は曇りです。
```

文字列の乗算

- 「* 整数」を使うことによって、文字列を整数回繰り返した文字列を生成することができます。

```
message = "abc"
# messageを3回繰り返します。
print(message * 3)
```

```
abcabcabc
```

演習11：文字列の結合

- 文字列を結合してください。

演習12：文字列の乗算

- 文字列の乗算を実施してください。

第13回：標準ライブラリ2

アジェンダ

- 標準ライブラリによる日付操作
- 標準ライブラリによる正規表現
- 標準ライブラリによるファイル入出力

標準ライブラリによる日付操作

日付操作

- 「datetime」モジュールは、日付や時刻を操作するためのクラスを提供しています。
- 「calendar」モジュールは、汎用のカレンダー関連関数を提供しています。
- 「time」モジュールは、時刻へのアクセスと変換機能を提供しています。
- 「dateutil」パッケージは、拡張タイムゾーンと構文解析サポートのあるサードパーティーライブラリです。

日付操作：datetime

- 「datetime」を使用して日付を取り扱うことができます。

```
# datetimeモジュールをインポートします。
import datetime

# 現在時刻を取得します。
dt_now = datetime.datetime.now()
print(dt_now)
```

```
2019-12-07 11:34:56.062119
```

- 現在時刻の年月日、時分秒を取得することができます。

```
# 現在時刻の年を取得します。
print(dt_now.year)
# 現在時刻の月を取得します。
print(dt_now.month)
# 現在時刻の日を取得します。
print(dt_now.day)
```

```
2019
12
7
```

```
# 現在時刻の時を取得します。
print(dt_now.hour)
# 現在時刻の分を取得します。
print(dt_now.minute)
# 現在時刻の秒を取得します。
print(dt_now.second)
```

```
11
34
56
```

日付操作：datetime

- コンストラクタを利用して、任意の日付、時刻のdatetimeオブジェクトを生成することも可能です。
  - [datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None)]
- 年year、月month、日dayは必須で、その他は省略可能です。省略した場合は初期値が設定されます。

演習1：datetimeによる現在時刻の取り扱い

- 標準ライブラリdatetimeを使用して、現在時刻を取得してください。
- 現在時刻の年月日を取得してください。
- 現在時刻の時分秒を取得してください。

演習2：任意のdatetimeオブジェクトの生成

- 任意の時刻のdatetimeオブジェクトを生成してください。

日付操作：calendar

- Pythonの標準ライブラリcalendarモジュールを使用すると、カレンダーをプレーンテキストやHTML、数値のリストなどで生成することができます。
- 列幅、行幅の指定が可能です。

```
import calendar

print(calendar.month(2019, 1))
```

```
    January 2019
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

```
# 列幅、行幅を指定します。
print(calendar.month(2019, 1, w=3, l=2))
```

```
       January 2019

Mon Tue Wed Thu Fri Sat Sun

     1   2   3   4   5   6

 7   8   9  10  11  12  13

14  15  16  17  18  19  20

21  22  23  24  25  26  27

28  29  30  31
```

日付操作：calendar

- 表示するカレンダーの週の始めの曜日を指定することができます。

```
# 週の始めを日曜日にします。
calendar.setfirstweekday(calendar.SUNDAY)
# 任意の年月のカレンダーを取得します。
print(calendar.month(2019, 1, w=3, l=2))
```

```
       January 2019

Sun Mon Tue Wed Thu Fri Sat

         1   2   3   4   5

 6   7   8   9  10  11  12

13  14  15  16  17  18  19

20  21  22  23  24  25  26

27  28  29  30  31
```

演習3：カレンダーオブジェクトの生成

- 標準ライブラリcalendarを使用して、任意の年月のカレンダーを取得してください。
- 列幅、行幅を指定してカレンダーを取得してください。

演習4：曜日の設定

- 表示するカレンダーの週の始めの曜日を指定してください。

日付操作 : time

- timeオブジェクトは時刻(時、分、秒、マイクロ秒)の情報を持つオブジェクトです。
- これらの情報に、属性hour、minute、second、microsecondでアクセスできます。

- コンストラクタを利用して、任意の時刻のtimeオブジェクトを生成できます。
  - [time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None)]

日付操作 : time

- 生成したtimeオブジェクトの属性(時・分・秒・ミリ秒)を参照することができます。

```
# timeオブジェクトを生成します。
t = datetime.time(12, 15, 30, 2000)
print(t)
```

```
12:15:30.002000
```

```
# 時
print(t.hour)
# 分
print(t.minute)
# 秒
print(t.second)
# ミリ秒
print(t.microsecond)
```

```
12
15
30
2000
```

演習5：timeオブジェクトの生成と参照

- 任意の時刻のtimeオブジェクトを生成してください。
- timeオブジェクトの属性にアクセスしてください。

日付操作：timedelta

- 二つの日時の時間差、経過時間を表すオブジェクトです。日数、秒数、マイクロ秒数の情報を持ち、属性days、seconds、microsecondsでアクセスできます。
- 2つのdatetimeオブジェクトの引き算を実行すると、timedeltaオブジェクトが得られます。

```
# datetimeモジュールをインポートします。
import datetime

# 現在時刻を取得します。
dt_now = datetime.datetime.now()
# 任意の時刻情報を設定します。
dt = datetime.datetime(2019, 12, 1, 12, 15, 30, 2000)
# datetime同士の引き算を実行します。
td = dt_now - dt
```

```
# 引き算の結果を表示します。
print(td)
# 型を確認します。
print(type(td))
```

```
6 days, 1:10:48.838166
<class 'datetime.timedelta'>
```

演習6：timedeltaオブジェクトの取り扱い

- datetimeオブジェクト同士の引き算を実行し、timedeltaオブジェクトを生成してください。
- timedeltaオブジェクトの属性にアクセスしてください。

日付操作：サードパーティーのライブラリ

- 日付操作ライブラリのdatetimeの拡張版で、dateutilライブラリがあります。興味のある方は[https://dateutil.readthedocs.io/en/stable/]を参照ください。

標準ライブラリによる正規表現

正規表現とは

- 正規表現とは、文字列を1つのパターン化した文字列で表現する表記法のことです。
- パターン化した文字列のことをメタ文字と言います。

メタ文字	説明	指定例	合致する
.	任意の一文字	a.c	abc, acc, aac
^	行の先頭	^abc	abcdef
$	行の末尾	abc$	defabc
*	0回以上の繰り返し	ab*	a, ab, abb, abbb
+	1回以上の繰り返し	ab+	ab, abb, abbb
?	0回または1回	ab?	a, ab
{m}	m回の繰り返し	a{3}	aaa
{m,n}	m〜n回の繰り返し	a{2, 4}	aa, aaa, aaaa
[★]	★のどれか1文字	[a-c]	a, b, c
★|★	★のどれか	a|b	a, b

メタ文字の例
参照：https://techacademy.jp/magazine/15635

正規表現：match

- 「match」を使用すると、検索文字列と一致した文字列を抽出できます。

```
import re

# 検索したい文字を指定します。
pattern = r'ky'
# compile
repatter = re.compile(pattern)

# 検索対象文字列の頭に検索文字列kyがある場合
content = r'kyoto station.'
result = repatter.match(content)

print(result.group())
```

```
ky
```

正規表現：match

- 検索文字列「yo」が検索対象文字列「i went to kyoto station.」の先頭にない場合、matchは検索できません。

```
# 検索対象文字列の頭に検索文字列kyがない場合
content = r'i went to kyoto station.'
result = repatter.match(content)
# 一致文字列がないためエラーが発生します。
print(result.group())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-12-a87e51abf2f6> in <module>
      3 result = repatter.match(content)
      4 # 一致文字列がないためエラーが発生します。
----> 5 print(result.group())

AttributeError: 'NoneType' object has no attribute 'group'
```

正規表現：match

- 正規表現「.*」を使って、検索文字列「yo」の直前までの表現「i went to k」を表現します。このように正規表現を使用すると、matchで検索することができます。

```
import re

# 検索したい文字を指定します。
pattern = r'.*yo'
# compile
repatter = re.compile(pattern)

# 検索対象文字列の頭に検索文字列kyがない場合
content = r'i went to kyoto station.'
result = repatter.match(content)

print(result.group())
```

```
i went to kyo
```

演習7：matchを使用した検索

- matchを使用して文字列マッチングを実行してください。
- 検索対象文字列の頭に検索文字列kyがない場合、matchでは検索できないことを確認してください。

演習8：正規表現とmatchを使用した検索

- matchを使用して、検索したい文字列が検索対象文字列の頭にない場合に、正規表現を利用して文字列マッチングを実施してください。

正規表現：search

- searchを使用すると、検索文字列が検索対象文字列に存在するだけで、検索することができます。

```
import re

# 検索したい文字を指定します。
pattern = r'ky'
# compile
repatter = re.compile(pattern)

# 検索対象文字列の頭に検索文字列kyがある場合
content = r'i went to kyoto station.'
result = repatter.search(content)

print(result.group())
```

```
ky
```

正規表現：findall

- searchは最初の結果のみを返しますが、findallを使用すると検索文字列と一致した箇所全てを返します。

```
import re

# 検索したい文字を指定します。
pattern = r'ky'
# 検索対象文字列の頭に検索文字列kyがある場合
content = r'i went to kyoto station and tokyo station.'

results = re.findall(pattern, content)

print(results)
['ky', 'ky']
```

演習9：searchによる検索

- searchを使用して、検索したい文字列が検索対象文字列の頭にない場合に、文字列マッチングを実施してください。

演習10：findallによる検索

- findallを使用して、複数箇所の検索結果を取得してください。

標準ライブラリによるファイル入出力

ファイルの読み込み

- 「open」関数を使用することにより、ファイルを読み込むことができます。
- ファイルに複数行記載されている場合、参照結果に改行コードが含まれます。

```
# 読み込むファイルのパス
path_input = "chapter13_test_folder/input1.txt"
```

```
with open(path_input) as f:
    # for文でループさせると、各行を参照することができます。
    for s_line in f:
        print(s_line)
```

```
line1

line2

line3
```

- 不要な改行を除いてください。

```
with open(path_input) as f:
    # for文でループさせると、各行を参照することができます。
    for s_line in f:
        print(s_line.strip())
```

```
line1
line2
line3
```

リスト形式でのファイル読み込み

- 「readlines」を使用することにより、ファイルに記載されている複数行をリストで取得することができます。
- 一行がリストの1要素に対応します。※改行コードが含まれます。

```
# 読み込むファイルのパス
path_input = "chapter13_test_folder/input1.txt"
```

```
with open(path_input) as f:
    # リストとして取得する。※改行を含む
    l = f.readlines()
    print(l)
```

```
['line1\n', 'line2\n', 'line3\n']
```

演習11：ファイルの読み込み

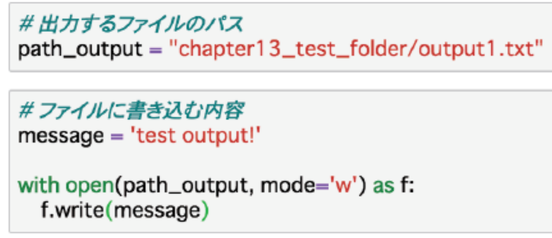
- テキストファイルを読み込み、記載されている内容を確認してください。
- 不要な改行を除いてください。

演習12：リストとしてのファイル読み込み

- 各行をリストの要素としてファイルを読み込んでください。

ファイルの出力

- open()の引数modeに‘w‘を設定することで、書き込み用としてファイルが開かれます。
- 指定したファイルが存在しなければ新規作成、存在していれば上書き（既存の内容は削除）で保存されます。



```
# 出力するファイルのパス
path_output = "chapter13_test_folder/output1.txt"
```

```
# ファイルに書き込む内容
message = 'test output!'

with open(path_output, mode='w') as f:
    f.write(message)
```

演習13：ファイルの出力

- ファイルを出力してください。出力するファイルには、任意の文字列を記載してください。

第14回：NumPyを使用したプログラミング

アジェンダ

- NumPyの基本
- NumPyによる配列の操作
- NumPyの数学関数
- NumPyによるデータの前処理

NumPyとは

- [WikipediaよりNumPyは、プログラミング言語Pythonにおいて数値計算を効率的に行うための拡張モジュールである。効率的な数値計算を行うための型付きの多次元配列（例えばベクトルや行列などを表現できる）のサポートをPythonに加えるとともに、それらを操作するための大規模な高水準の数学関数ライブラリを提供する。

スカラー、ベクトル、行列、テンソル

- 実数をスカラーといいます。
- スカラーを1次元の配列にしたものをベクトルといいます。
- スカラーを縦横に長方形に並べたものを行列といいます。
- 行列を更に入れ子にした3次元配列をテンソルといいます。

```
scalar_1 = -1.2
scalar_2 = 2
```

```
# 要素が1つのベクトル
vector_1 = [2]
# 要素が2つのベクトル
vector_2 = [2, 4]
```

```
# 2行3列の行列
matrix = [
    [1, 2, 3],
    [4, 5, 6]
]
```

```
# 2×3×3のテンソル
tensor = [
    [
        [1, 2, 3],
        [4, 5, 6]
    ],
    [
        [7, 8, 9],
        [10, 11, 12]
    ],
    [
        [13, 14, 15],
        [16, 17, 18]
    ]
]
```

スカラー（0次元の配列）	0階のテンソル
ベクトル（1次元の配列）	1階のテンソル
行列（2次元の配列）	2階のテンソル
（3次元の配列）	3階のテンソル
（4次元の配列）	4階のテンソル
:	:
（X次元の配列）	X階のテンソル

演習1：スカラー、ベクトル、行列、テンソルの表現

- スカラーを表現してください。
- ベクトルを表現してください。
- 行列を表現してください。
- テンソルを表現してください。

NumPyにおけるデータ構造

- NumPyは多次元配列を基本的なデータ構造として操作するライブラリです。
- 効率よく計算するため、NumPyではリストを使った演算ではなくndarrayという独自のデータ構造を使います。

- ndarrayというのはN-dimensional array（N次元配列）の略です。
- 要素は同じ属性や大きさを持つ必要があります。
  - リストのように要素ごとにデータ型を柔軟に変えることはできません。
  - 各次元ごとの要素数(2次元なら列ごとや行ごと) は必ず一定にする必要があります。

ndarrayの属性：T（転置）

- 行列の転置を返します。次元<2の場合は元の配列が返されます。
- 転置を含め、属性を参照した際にndarrayの値は一切変わりません。例えば、[.T]で転置させたndarrayを表示させても、元のデータが変更になることはありません。

演習2：テンソルの転置

- NumPyで1次元配列を作成し、作成した1次元配列の転置行列を取得してください。
- 2次元配列を作成し、作成した2次元配列の転置行列を取得してください。
- 転置属性を参照した後に再度行列を参照し、参照前後で値が変わっていないことを確認してください。

ndarrayの属性：flat（1次元配列への変換）

- 多次元配列を1次元配列に変換します。
- [.flat[n]]とすることで、変換後の1次元配列の要素を確認できます。

```
# NumPyで行列(2次元配列)を作成します。
array_2d = np.array([[1, 2, 3],[4, 5, 6]])
```

```
# 1次元配列に変換します。
print(array_2d.flat)
```

```
<numpy.flatiter object at 0x7f9d188b1e00>
```

```
# 要素を確認します。
for v in array_2d.flat:
    print(v)
```

```
1
2
3
4
5
6
```

演習3：多次元配列の1次元配列への変換

- 多次元配列を1次元配列に変換してください。

ndarrayの属性：real / imag（複素数の取り扱い）

- 要素が複素数の配列に対して、実部と虚部を分けて表示することができます。
- [real]属性を参照すると実部を表示することができます。
- [imag]属性を参照すると虚部を表示することができます。

```
# 複素数を含む1次元配列を作成します。
array_complex_1d = np.array([1.2-2.3j, 2.1+3.6J, 4.-3.2j])
print(array_complex_1d)
```

```
[1.2-2.3j 2.1+3.6j 4. -3.2j]
```

```
# 実部を表示します。
print(array_complex_1d.real)
# 虚部を表示します。
print(array_complex_1d.imag)
```

```
[1.2 2.1 4. ]
[-2.3  3.6 -3.2]
```

演習4：複素数で構成される配列の取り扱い

- 複素数で構成される1次元配列の実部、虚部を分けて表示してください。
- 複素数で構成される2次元配列の実部、虚部を分けて表示してください。

ndarrayの属性：size / itemsize / nbytes

- 「size」を参照すると、配列に含まれる要素の数を参照することができます。
- 下の例では、2×3の2次元配列の要素数6が返ってきます。

```
# NumPyで行列(2次元配列)を作成します。
array_2d = np.array([[1, 2, 3],[4, 5, 6]])
```

```
# 要素数を表示します。
print(array_2d.size)
```

```
6
```

ndarrayの属性：size / itemsize / nbytes

- 「itemsize」を参照すると、要素1つ当たりのメモリ容量を参照することができます。
- 下の例では、要素1つのメモリ容量は8バイトであることが確認できます。

```
# 要素のメモリ容量を表示します。
print(array_2d.itemsize)
```

```
8
```

- 「nbytes」を参照すると、配列全体のメモリ容量を参照することができます。
- 「array_2d」配列は要素1つあたり8バイト、要素が6つあるので、8×6で48バイトであることが確認できます。

```
# 2次元配列全体のメモリ容量を表示します。
print(array_2d.nbytes)
```

```
48
```

演習5：配列のサイズ

- 2次元配列を作成し、配列に含まれる要素数、要素のメモリ容量、配列全体のメモリ容量を表示してください。
- 複素数を含む2次元配列を作成し、配列に含まれる要素数、要素のメモリ容量、配列全体のメモリ容量を表示してください。

ndarrayの属性： ndim / shape

- 「ndim」を参照すると、配列の次元を参照することができます。
- 「shape」を参照すると、配列の形状（m×n行列などの情報）を参照することができます。

```
# NumPyで行列(2次元配列)を作成します。
array_2d = np.array([[1, 2, 3],[4, 5, 6]])
```

```
# 次元を確認します。
print(array_2d.ndim)
```

```
2
```

```
# 形状を確認します。
print(array_2d.shape)
```

```
(2, 3)
```

演習6：配列の次元、形状

- 1次元配列を作成し、次元と形状を確認してください。
- 2次元配列を作成し、次元と形状を確認してください。
- 3次元配列を作成し、次元と形状を確認してください。

Numpyによる配列の操作

スライシング：開始位置、終了位置の指定

- NumPyのndarrayは多次元データ構造をより操作しやすくするために、スライシングという機能を備えています。
- [start:stop:step]を指定し、startからstopまでstepごとの要素を参照します。例えばk番目の要素からl番目の要素まで2個おきに参照したいときは、[k:l+1:2]とすれば目的の要素を抜き出すことができます（stopは目的の要素+1されていることに注意してください。）

```
# NumPyでベクトル(1次元配列)を作成します。
array_1d = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8, 9])
```

```
"""
array_1dの1, 2, 3が取得したい場合、開始位置は1のインデックスである1を設定します。
終了位置は3のインデックスである3に1を足した4を設定します。
"""
print(array_1d[1:4])
```

スライシング：開始位置、終了位置の省略

- 先頭から要素を取得したい場合、startは省略することができます。

```
"""
array_1dのインデックスが0~4の要素を取得したいので、開始位置は0を設定します。
終了位置はインデックス4に1を足した5を設定します。
"""
print(array_1d[0:5])
```

```
[0 1 2 3 4]
```

```
"""
先頭から4つの要素を取得したいことと同義なので、開始位置の0は省略できます。
"""
print(array_1d[:5])
```

```
[0 1 2 3 4]
```

- 最後の位置まで要素を取得したい場合、stopは省略することができます。

```
"""
開始位置の4のみ指定し、終了位置は省略できます。
"""
print(array_1d[4:])
```

```
[4 5 6 7 8 9]
```

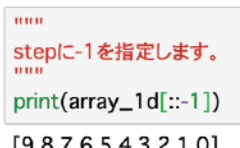
スライシング：その他の参照の仕方

- Stepを指定することで、要素を飛ばして参照することができます。

```
"""
開始/終了位置は省略し、stepのみを指定します。
"""
print(array_1d[::2])
```

```
[0 2 4 6 8]
```

- Stepに-1を指定することで、要素の並びを逆にして参照することができます。



```
"""
stepに-1を指定します。
"""
print(array_1d[::-1])
```

```
[9 8 7 6 5 4 3 2 1 0]
```

演習7：1次元配列のスライス

- 1次元配列をスライスし、インデックスが1から3の間にある要素を抽出してください。
- 1次元配列をスライスし、インデックスが0から4の間にある要素を抽出してください。
- 1次元配列をスライスし、インデックスが5から最後の要素までを抽出してください。
- 1次元配列をスライスし、要素を1つ飛ばしで抽出してください。
- 要素の並びを逆にして参照してください。

スライシング：2次元以上の配列への応用

- 次元を増やした場合でも、[:,:,:]とカンマ区切りで軸の数だけ指定することができます。
- 例えば、2次元配列に対しては以下のように指定します。

```
# NumPyでベクトル(2次元配列)を作成します。
array_2d = np.array(
    [
      [0, 1, 2, 3, 4, 5, 6 ,7, 8, 9],
      [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
      [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
    ]
)
```

```
"""
1つ目の軸と2つ目の軸のそれぞれに対してインデックスを指定します。
"""
print(array_2d[1:3, 2:5])
```

```
[[12 13 14]
 [22 23 24]]
```

スライシング：2次元以上の配列への応用

- 1次元の配列と同じように、start / stop / stepは省略することができます。
- 下記では1つ目の軸に対するstopを省略しています（2〜3行目と抽出）。

```
# NumPyでベクトル(2次元配列)を作成します。
array_2d = np.array(
  [
    [0, 1, 2, 3, 4, 5, 6 ,7, 8, 9],
    [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
  ]
)
```

```
"""
3行までしかないため、2行目のインデックスのみを指定します。
"""
print(array_2d[1:,])
```

```
[[10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]]
```

演習8：2次元配列のスライス

- 2次元配列をスライスしてください。2~3行目のインデックスが2〜4の範囲を抽出してください。
- 3行×10列の配列の2行目〜3行目をスライスしてください。終了位置を省略して実装してください。

配列の行数、列数の変換：reshape

- 配列の行数、列数を変換します。
- 下記の例では、12個の要素からなる1次元配列を3行×4列の2次元配列に変換しています。

```
# NumPyでベクトル(1次元配列)を作成します。
array_1d = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8, 9, 10, 11])
```

```
# 3×4配列に変換します。
array_3_4 = np.reshape(array_1d, (3, 4))
print(array_3_4)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

配列の行数、列数の変換：reshape

- 変換前後の配列の要素数が一致しない場合、エラーが発生します。

```
# NumPyでベクトル(1次元配列)を作成します。
array_1d = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8, 9, 10, 11])
```

```
# 3×3配列に変換を試みます。
array_3_3 = np.reshape(array_1d, (3, 3))
print(array_3_3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-7125fa658ea3> in <module>
      1 # 3×3配列に変換を試みます。
----> 2 array_3_3 = np.reshape(array_1d, (3, 3))
      3 print(array_3_3)
```

演習9：reshapeによる配列の形状変換

- 1次元配列を2次元配列に形状変換してください。
- 要素数 != M行×N列　となるM×N行列に変換してください（エラーが発生します）。
- 2次元配列は1次元配列に形状変換してください。

配列の行数、列数の変換：resize

- 変換前後の配列の要素数が一致しない場合、reshapeではエラーが発生しましたが、resizeではエラーは発生せず無理やり変換します。

```
# NumPyでベクトル(1次元配列)を作成します。
array_1d = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8, 9, 10, 11])
```

```
# 無理やり3×3配列に変換を試みます。
array_3_3 = np.resize(array_1d, (3, 3))
print(array_3_3)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

演習10：resizeによる配列の形状変換

- 要素数 != M行×N列　となるM×N行列に変換してください（reshapeと異なりエラーは発生しません）。

配列への要素の追加：append

- 配列の末尾に要素を追加します。
- 下記の例では、1次元配列の末尾に2つの要素を追加しています。

```
# NumPyでベクトル(1次元配列)を作成します。
array_1d = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8, 9, 10, 11])
```

```
# 末尾に要素を追加します。
print(np.append(array_1d, [12, 13]))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13]
```

配列への要素の追加：append

- 2次元配列に要素と追加することもできます。Axisで追加する軸を指定します。
- shapeが一致しないとエラーが発生します。

```
# NumPyでベクトル(2次元配列)を作成します。
array_2d = np.array(
    [
        [0, 1, 2, 3, 4, 5, 6 ,7, 8, 9],
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
    ]
)
```

```
"""
行方向に要素を追加します。
"""
print(np.append(array_2d, [[20, 21, 22, 23,24, 25, 26, 27, 28, 29]], axis=0))
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]]
```

```
"""
列方向に要素を追加します。
"""
print(np.append(array_2d, [[-1],[-2]], axis=1))
```

```
[[ 0  1  2  3  4  5  6  7  8  9 -1]
 [10 11 12 13 14 15 16 17 18 19 -2]]
```

演習11：配列への要素の追加

- 1次元配列の末尾に要素を追加してください。
- 2次元配列に要素を追加してください。

配列の軸の入れ替え：transpose

- 配列の軸を入れ替える場合、transposeを使用します。下記の例では、0軸と1軸を入れ替えています。

```
#NumPyでベクトル(2次元配列)を作成します。
array_2d = np.array(
    [
        [0, 1, 2, 3, 4, 5, 6 ,7, 8, 9],
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
    ]
)
```

```
#軸を入れ替えます。
print(array_2d.transpose(1, 0))
```

```
[[ 0 10]
 [ 1 11]
 [ 2 12]
 [ 3 13]
 [ 4 14]
 [ 5 15]
 [ 6 16]
 [ 7 17]
 [ 8 18]
 [ 9 19]]
```

演習12：配列の軸の入れ替え

- 2次元配列の軸を入れ替えてください。
- 3次元配列の軸を入れ替えてください。

Numpyの数学関数

要素の平均：average

- averageを使用すると、要素の平均を取得することができます。
- 1次元配列の場合は、スカラー値が返却されます。

```
# NumPyでベクトル(1次元配列)を作成します。
array_1d = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8, 9])
```

```
# 要素の平均値を求めます。
print(np.average(array_1d))
```

```
4.5
```

要素の平均：average

- 2次元配列の場合は、axisで平均を算出する軸を指定することができます。

```
# NumPyでベクトル(2次元配列)を作成します。
array_2d = np.array(
  [
    [0, 1, 2, 3, 4, 5, 6 ,7, 8, 9],
    [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
  ]
)
```

```
# 行方向に要素の平均値を求めます。
print(np.average(array_2d, axis = 0))
```

```
[ 5.  6.  7.  8.  9. 10. 11. 12. 13. 14.]
```

```
# 列方向に要素の平均値を求めます。
print(np.average(array_2d, axis = 1))
```

```
[ 4.5 14.5]
```

演習13：要素の平均

- 1次元配列の要素の平均を求めてください。
- 2次元配列の平均を求めてください。

中央値の取得：median

- データを降順(もしくは昇順)に並べた際に、ちょうど中央の順番に並ぶ値のことを中央値といいます。
- データの個数が偶数の場合、中央値は真ん中2つの値の平均を取ります。

$$(a_1, a_2, \cdots, a_n, a_{n+1}, \cdots, a_{2n}) \Rightarrow \frac{a_n + a_{n+1}}{2}$$

- データの個数が奇数の場合、ちょうど真ん中の値が中央値となります。

$$(a_1, a_2, \cdots, a_n, \cdots, a_{2n-1}) \Rightarrow a_n$$

```
# 要素が奇数個の1次元配列を作成します。
array_1d_odd = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8])
```

```
# 中央値を表示します。
print(np.median(array_1d_odd))
```

```
4.0
```

```
# 要素が偶数個の1次元配列を作成します。
array_1d_even = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8, 9])
```

```
# 中央値を表示します。
print(np.median(array_1d_even))
```

```
4.5
```

中央値の取得：median

- 2次元以上の配列に対しても、axis(軸方向)を指定することで中央値を求めることができます。

```
# NumPyでベクトル(2次元配列)を作成します。
array_2d = np.array(
    [
        [0, 1, 2, 3, 4, 5, 6 ,7, 8,],
        [10, 11, 12, 13, 14, 15, 16, 17, 18]
    ]
)
```

```
# 軸を指定しないで中央値を表示します。
# 全ての要素の中から中央値を算出します。
print(np.median(array_2d))
```

```
9.0
```

```
# 軸0方向(行)に対して中央値を算出します。
print(np.median(array_2d, axis=0))
```

```
[ 5.  6.  7.  8.  9. 10. 11. 12. 13.]
```

```
# 軸1方向(列)に対して中央値を算出します。
print(np.median(array_2d, axis=1))
```

```
[ 4. 14.]
```

演習14：中央値

- 1次元配列の要素の中央値を求めてください。要素が奇数個と偶数個の両方を試してください。
- 2次元配列の要素の中央値を求めてください。軸の方向も変えて試してみてください。

合計値の取得：sum

- 要素の和を求めます。
- 1次元配列の場合は、要素全ての和を計算します。

```
# 要素が奇数個の1次元配列を作成します。
array_1d = np.array([0, 1, 2, 3, 4, 5, 6 ,7, 8])
```

```
# 要素の和を表示します。
print(np.sum(array_1d))
```

```
36
```

合計値の取得：sum

- 2次元配列の場合も、和を求める軸を指定しなければ全ての要素の和を計算します。
- 軸を指定した場合、その方向に要素の和を計算します。

```
# NumPyでベクトル(2次元配列)を作成します。
array_2d = np.array(
    [
        [0, 1, 2, 3, 4, 5, 6 ,7, 8,],
        [10, 11, 12, 13, 14, 15, 16, 17, 18]
    ]
)
```

```
# 要素の和を表示します。
# 軸を指定しないので、全ての要素の和を計算します。
print(np.sum(array_2d))
```

```
162
```

```
# 要素の和を表示します。
# 軸0方向(行)に対して要素の和を計算します。
print(np.sum(array_2d, axis=0))
```

```
[10 12 14 16 18 20 22 24 26]
```

```
# 要素の和を表示します。
# 軸1方向(列)に対して要素の和を計算します。
print(np.sum(array_2d, axis=1))
```

```
[ 36 126]
```

演習15：要素の合計値

- 1次元配列の要素の和を求めてください。
- 2次元配列の要素の和を求めてください。軸の指定を変えて実行してください。

標準偏差の取得：std

- 標準偏差とは、データの平均との差を2乗した値の平均に対して平方根をとった数値です。
- データの散らばり具合を表す指標の1つとして用いられます。
- 値が[1, 2, 3, 4, 5]の要素の（標本）標準偏差は約1.41となります。

データ	1	1	1	1	1.1	1.01
	20	10	5	2	1.2	1.02
	300	100	10	3	1.3	1.03
	4000	1000	15	4	1.4	1.04
	50000	10000	20	5	1.5	1.05
平均値	10864.2	2222.2	10.2	3	1.3	1.03
（標本）標準偏差	19626.1876	3906.89741	6.7941151	1.41421356	0.14142136	0.01414214
変動係数 =標準偏差/平均値	1.80650095	1.75812142	0.66608972	0.47140452	0.10878566	0.01373023

標準偏差の取得：std

- 母集団の標準偏差（不偏分散の平方根、不偏標準偏差）を求めたいときは、引数ddofに1を設定します。

```
# 要素の標準偏差を表示します。
print(np.std(array_1d))
```

1.4142135623730951

```
# ddofに1を設定します。
print(np.std(array_1d, ddof=1))
```

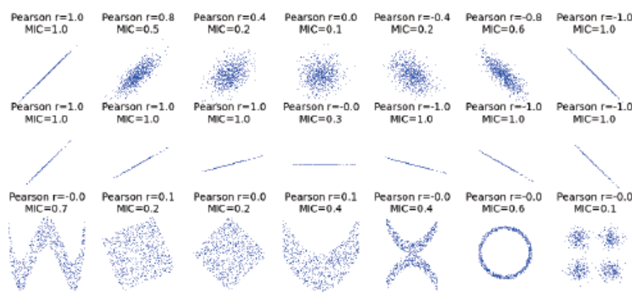
1.5811388300841898

演習16：標準偏差

- 1次元配列の要素の標準偏差を求めてください。
- 不偏標準偏差を求めてください。
- 2次元配列の要素の標準偏差を求めてください。

相関係数の取得：corrcoef

- 相関係数は、2組のデータ間にどれほどの関係性があるのかを示す1つの指標としてよく使用されます。
- 相関係数が大きくても、2つのデータ間に因果関係が成り立っているとは限らないので、注意が必要です。
- 「corrcoef」で求められる相関係数は、あくまで線形の関係性についての強さの指標です。非線形の関係性についてはMIC（Maximal information coefficient）などで評価します。



参照：https://minepy.readthedocs.io/en/latest/

演習17：相関係数

- 2組の数値群(2次元配列)の相関係数を求めてください
- 別の数値群(1次元配列)を作成し、上で作成した2組の数値群との相関係数を求めてください。

ベクトルの内積・行列の積：dot

- ベクトルの内積とは、各要素の積を全て足し合わせたスカラー量を求めることです。

$$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}, \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ a_n \end{pmatrix}$$

$$\vec{a} \cdot \vec{b} = a_1 * b_1 + a_2 * b_2 + \cdots + a_n * b_n$$

ベクトルの内積・行列の積：dot

- 行列の積では、行ベクトルと列ベクトルとの内積を要素とする行列が新たに作り出されます。
- 積を構成する左側の行列の列数と、右側の行列の行数が一致する必要があります。

$$(a_1, a_2, \cdots, a_n) \cdot \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ a_n \end{pmatrix} = (a_1 * b_1 + a_2 * b_2 + \cdots + a_n * b_n)$$

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$A \cdot B = \begin{pmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} + b_{22} \end{pmatrix}$$

演習18：ベクトルの内積・行列の積

- 2つのベクトルの内積を計算してください。
- 1×1行列と1×2行列の積を計算してください（エラーが発生します）。
- 2×3行列と3×2行列の積を計算してください。

Numpyによるデータの前処理

なぜ前処理（正規化）が必要か

- モデル作成に説明変数（パラメータ）を複数使用する場合、説明変数間でそれぞれスケールが違うと、計算するときにスケールの違いに値が引っ張られてしまうことがあります。それを防ぐために正規化を実施します。

- 例えば、身長と体重が挙げられます。身長はcmで表現すると「178cm」の様に3桁の数値となり、体重はKgで表現すると「68Kg」のように2桁の数値となります。身長と体重を説明変数としたモデルを作成すると、数値の大きな身長により重みをおいたモデルになってしまいます。
- また、身長を「178cm」と数値そのままで表現するのではなく、「平均身長170cmからの差分」など何かしらの加工を施したほうがデータ群の特徴をよく表現できるようになることがあります。

Min-Max normalization

- 最小値を0、最大値を1とする正規化のことです。
- (Min-Max normalizationに限らずどの正規化にも共通していますが)データ内に外れ値が存在する場合、外れ値を処理した後に当該正規化を実施した方がモデルの精度は向上しやすくなります。

$$x^i_{new} = \frac{x^i - x_{min}}{x_{max} - x_{min}}$$

演習19：Min-Max normalization

- Min-Max normalizationを実装してください。

z-score normalization（標準化）

- 元データを平均0、標準偏差が1のデータに変換する正規化のことです。

$$x^i_{new} = \frac{x^i - \mu}{\sigma}$$

演習20：z-score normalization（標準化）

- z-score normalization（標準化）を実装してください。

第15回：Matplotlibによるデータの可視化

アジェンダ

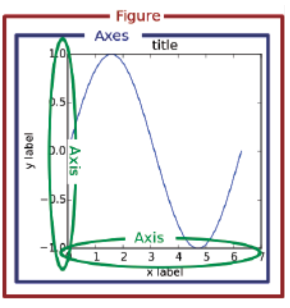
- Matplotlibとは
- オブジェクトの構成
- 様々なグラフの作成
- 複数グラフの表示

Matplotlibとは

- [Wikipediaより] Matplotlibは、プログラミング言語Pythonおよびその科学計算用ライブラリNumPyのためのグラフ描画ライブラリである。オブジェクト指向のAPIを提供しており、様々な種類のグラフを描画する能力を持つ。描画できるのは主に2次元のプロットだが、3次元プロットの機能も追加されてきている。

オブジェクトの構成

- Matplotlibを使用するには、まずFigureオブジェクトを生成します。
- Figureオブジェクトの中に、グラフを表示するAxesオブジェクトを生成し、さらに軸を表示するためのAxisオブジェクトを生成します。

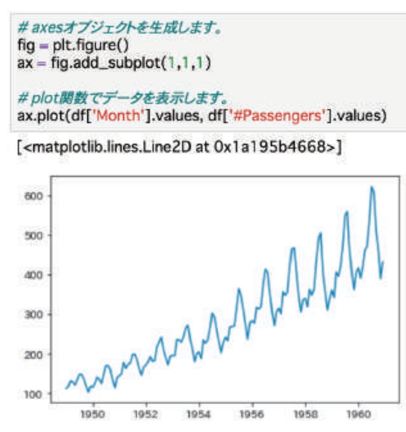


参照：
https://matplotlib.org/1.5.1/faq/usage_faq.html

様々なグラフの作成

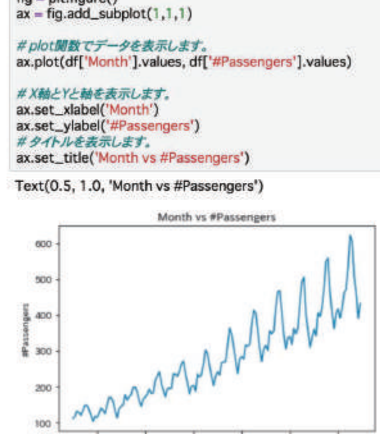
折れ線グラフ

- Axesオブジェクトのplot関数を使用すると、折れ線グラフを作成することができます。
- plot関数の引数として与えるデータの順番でX軸、Y軸を指定することができます。



折れ線グラフ

- Axesオブジェクトのset_xlabel / set_ylabel / set_title関数を使用するとX軸名、Y軸名、タイトルを表示させることができます。

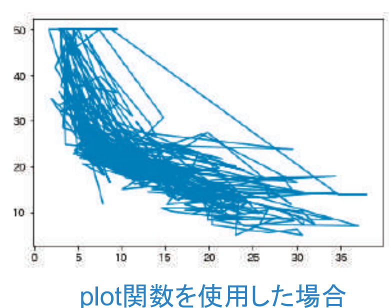


演習1：折れ線グラフの作成

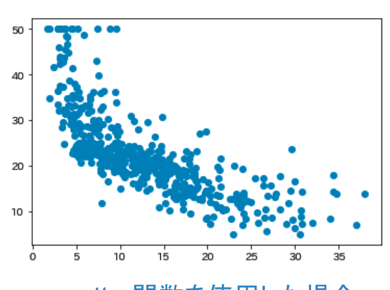
- 飛行機乗客数の時系列データを読み込み、Y軸に乗客数、X軸に年月日のグラフを表示してください。
- X軸とYと軸のラベルを表示してください。また、グラフのタイトルも表示してください。

散布図

- Axesオブジェクトのplot関数を使用すると、折れ線グラフが作成できました。
- Axesオブジェクトのscatter関数を使用すると、散布図を作成することができます。



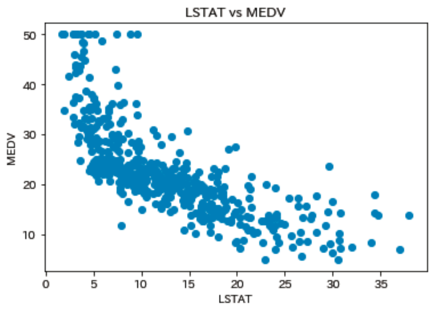
plot関数を使用した場合



scatter関数を使用した場合

散布図

- 折れ線グラフと同様に、Axesオブジェクトのset_xlabel / set_ylabel / set_title関数を使用するとX軸名、Y軸名、タイトルを表示させることができます。

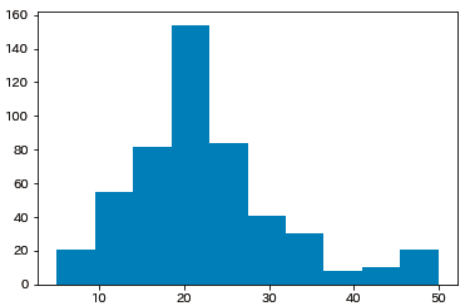


演習2：散布図の作成

- ボストン住宅価格データを読み込み、Y軸に住宅価格、X軸に低所得者の割合のグラフを表示してください。
- 折れ線ではなく、散布図を作成してください。
- X軸とYと軸のラベルを表示してください。また、グラフのタイトルも表示してください。

ヒストグラム

- Axesオブジェクトのhist関数を使用すると、ヒストグラムを作成することができます。



ヒストグラム

- ヒストグラムの度数を計算する際の範囲を決めるbin（棒の数）を、引数で指定することができます。

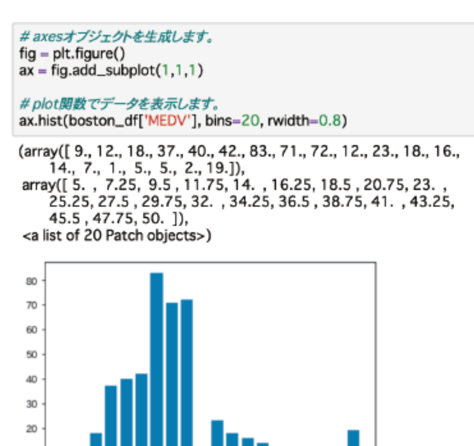
```
# axesオブジェクトを生成します。
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

# plot関数でデータを表示します。
ax.hist(boston_df['MEDV'], bins=20)
```

```
(array([ 9., 12., 18., 37., 40., 42., 83., 71., 72., 12., 23., 18., 16.,
       14.,  7.,  1.,  5.,  5.,  2., 19.]),
 array([ 5.  ,  7.25,  9.5 , 11.75, 14.  , 16.25, 18.5 , 20.75, 23.  ,
       25.25, 27.5 , 29.75, 32.  , 34.25, 36.5 , 38.75, 41.  , 43.25,
       45.5 , 47.75, 50.  ]),
 <a list of 20 Patch objects>)
```

ヒストグラム

- グラフを見やすくするため、binの幅を指定することもできます。



演習3：ヒストグラムの作成

- ボストン住宅価格データを読み込み、住宅価格のヒストグラムを表示してください。
- bin（棒の数）を20に増やし、より細かいヒストグラムを表示してください。
- binの幅を狭くしてください。
- X軸とYと軸のラベルを表示してください。また、グラフのタイトルも表示してください。

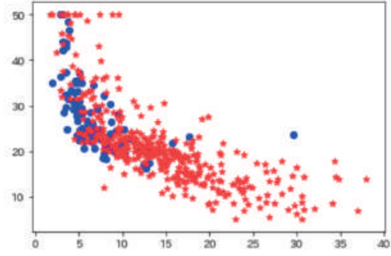
データの重ね合わせ

- Axesオブジェクトに表示するデータは、plot関数などを繰り返し実行することで重ねて表示することができます。
- データを区別しやすくするため、色やプロットの形を指定する事もできます。

```
# axesオブジェクトを生成します。
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

# plot関数で若い年代層にデータを表示します。
ax.scatter(df_younger['LSTAT'], df_younger['MEDV'], c='blue')
# plot関数で年配の年代層にデータを表示します。
ax.scatter(df_older['LSTAT'], df_older['MEDV'], c='red', marker="*")
```

```
<matplotlib.collections.PathCollection at 0x1a1a238198>
```



演習4：データの重ね合わせ

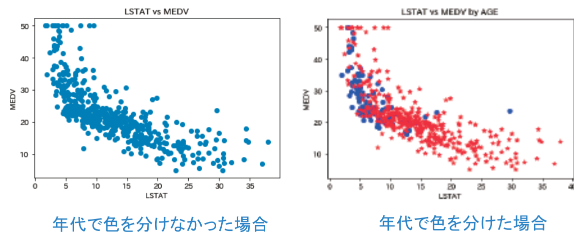
- ボストン住宅価格データを読み込み、Y軸に住宅価格、X軸に低所得者の割合のグラフを表示してください。
- 40歳以下のデータと40歳より上のデータを、色を変えて表示してください。
- X軸とYと軸のラベルを表示してください。また、グラフのタイトルも表示してください。

可視化の重要性

- ビジネスにおいて、何か判断を下す場面ではその理由がセットで議論されることがほとんどです。昨今、ディープラーニングなど表現力は高いが説明性が低いアルゴリズムが登場していますが、「精度さえ高ければ良い」というのは限られた場面だけなのが現状です。

- また、データ分析においては、どの説明変数が重要なのかを見極める作業が何度も繰り返されます。
- 分析者はドメイン知識（データが発生した背景、業務などの情報）に明るいクライアントと2人3脚で分析を進めることが多いですが、「視覚に訴えるわかりやすい可視化」は新たな気付きを得て、また効率よく分析を進める上でも非常に効果的です。

- 様々なBIツールや機械学習ライブラリが登場しており特徴選択、特徴加工の自動化が試みられていますが、まだまだ人間の知識や気付きがモデリングに与える影響は大きいのが現状です。



年代で色を分けなかった場合　　年代で色を分けた場合

複数グラフの表示

複数のAxesオブジェクトの作成

- Figureオブジェクトの中には複数のAxesオブジェクトを作成することができます。
- Figureオブジェクトのadd_subplot関数を使用して、Axesオブジェクトを縦横に何個並べるのかを指定できます。

fig.add_subplot(行,列,場所)を表します。

例1：fig.add_subplot(3,3,5)

1	2	3
4	⑤	6
7	8	9

例2：fig.add_subplot(3,4,7)

1	2	3	4
5	6	⑦	8
9	10	11	12

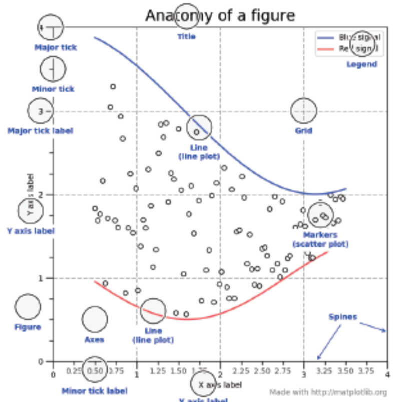
参照：https://teratail.com/questions/72586

演習5：複数グラフの表示

- 縦に2つ並んだグラフを作成してください。
- 上には、ボストン住宅価格データの、Y軸に住宅価格、X軸に低所得者の割合のグラフを表示してください。
- 下には、住宅価格のヒストグラムを表示してください。

Matplotlibのまとめ

- 本講義ではオブジェクトの構成、各種グラフの作成、複数グラフの作成などを学習してきました。
- オブジェクトや関数のマップは[https://matplotlib.org/tutorials/introductory/usage.html]においてより詳細に紹介されていますので、興味のある方は参考にしてください。



令和２年度「専修学校による地域産業中核的人材養成事業」
Society5.0 実現のための IT 技術者養成モデルカリキュラム開発と実証事業

■実施委員会

◎	船山　世界	日本電子専門学校　校長
	大川　晃一	日本電子専門学校　エンジニア教育部長／ケータイ・アプリケーション科科長
	種田　裕一	東北電子専門学校　第２教務部長　学生サポート室長
	勝田　雅人	トライデントコンピュータ専門学校　校長
	安田　圭織	学校法人上田学園　上田安子服飾専門学校
	平田　眞一	学校法人第一平田学園　理事長
	平井　利明	静岡福祉大学　特任教授
	木田　徳彦	株式会社インフォテックサーブ　代表取締役
	渡辺　登	合同会社ワタナベ技研　代表社員
	岡山　保美	株式会社ユニバーサル・サポート・システムズ　取締役
	冨田　慎一郎	株式会社ウチダ人材開発センタ　代表取締役社長

■人材育成委員会

◎	大川　晃一	日本電子専門学校　エンジニア教育部長／ケータイ・アプリケーション科科長
	福田　竜郎	日本電子専門学校　AI システム科
	阿保　隆徳	東北電子専門学校　学科主任
	小澤　慎太郎	中央情報大学校　高度情報システム学科
	神谷　裕之	名古屋工学院専門学校　メディア学部　情報学科
	北原　聡	麻生情報ビジネス専門学校　校長代行
	原田　賢一	有限会社ワイズマン　代表取締役
	柴原　健次	合同会社ヘルシーブレイン　代表 CEO
	菊嶋　正和	株式会社サンライズ・クリエイティブ　代表取締役

■評価委員会

平井　利明	静岡福祉大学　特任教授
冨田　慎一郎	株式会社ウチダ人材開発センタ　代表取締役社長
平田　眞一	学校法人第一平田学園　理事長

令和２年度「専修学校による地域産業中核的人材養成事業」
Society5.0 実現のための IT 技術者養成モデルカリキュラム開発と実証事業

AI プログラミングⅠ教材

令和３年２月

学校法人電子学園（日本電子専門学校）
〒169-8522　東京都新宿区百人町 1-25-4
TEL 03-3369-9333　FAX 03-3363-7685

●本書の内容を無断で転記、掲載することは禁じます。