III Pythonによるプログラミングの基礎

本講義ではサンプルプログラムの仕組みを理解できる程度の Python の知識を身につけることを目的として、Python の基本の構文について説明します。

1. Python の基本

1) コメントと文字列

Spyder でファイルを新規作成すると最初に以下のような行が記述されたファイルができます (USER NAME の部分には実行環境のユーザー名が入ります)。

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan  5 13:45:49 2018

@author: USER NAME
"""
```

コード 1 行目の # からはじまる行はファイルのエンコーディングを指定しています。Python ではスクリプトファイル内に指定がなければ基本的に UTF-8 形式で記述されたものとして実行しますが、ここでは明示的に UTF-8 形式を指定しています。3 行目はファイルを作成した日付、5 行目はコード作成者の名前（PC のユーザー名）です。Spyder ではこれらの行が自動的に追加されるようになっています。

コード 1 行目以外に # で始まる行が出てきた場合、 # から行末まではコメントとなり、実行時には無視されます。Spyder ではコメント行は灰色で表示されます。

```
# から始まる文はコメント行でコードとして認識されません
2  # 文の途中からコメントにすることもできます
```

2 行目と 6 行目にそれぞれダブルクォーテーションが 3 つ (""") ずつ記述してあります。Python では文字列を複数行に分けて書く場合、このようにクォーテーションを文頭と文末に 3 つずつ配します。クォーテーションはダブル・シングルのいずれでも使えますが、文頭と文末で種類をそろえる必要があります。

```
# ダブルクォーテーション３つで文字列を囲む
"""
複数行
に
```

```
分割
"""

# シングルクォーテーション３つで文字列を囲む
'''
メッシュ
気象
データ
'''
```

2) 数値、文字列と四則演算

数値・文字列を表す基本的なデータ型は以下の表の通りです。

表１　Python の代表的なデータ型

型	型名	内容
int	整数型	0 か、または 0 以外の数字で始まる整数
float	浮動小数点型	小数形式(123.45)もしくは指数形式(12345e-2)
str	文字列型	Unicode 文字の配列。前後を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(")で囲む

以下に記述例を示します。

```
1000  # 整数
1000.0  # 浮動小数点数
10e-2  # 0.01; 浮動小数点数
"AMeDAS"  # 文字列
'メッシュ'  # 文字列
'Say "Hello".'  # 文字列(クォーテーションマーク自体を表示するときはこうします)
```

3) 四則演算と代入文

四則演算の演算子は表に示す通りです。

表 2　四則演算の演算子

演算子	内容
+	加算
-	減算
*	乗算
/	除算
//	除算（切り捨て）
%	剰余
**	指数

四則演算は以下のように記述します。

```
1 + 2     # 3
1.0 + 2.0 # 3.0
2 - 1     # 1
2.0 - 1.0 # 1.0
2 * 2     # 4
2 * 2.0   # 4.0
6 / 3     # 2
7 / 3     # 2.3333333333333335
7 // 3    # 2
7 % 3     # 1
```

【注】除算については Python2 と Python3 で挙動が異なるので注意が必要です。今回用いている Pyton3 では整数（int 型）同士の割り算でも浮動小数点型（float 型）で戻りますが、Python2 では整数値が戻ります。

加算(+)や乗算(*)は文字列にも使えます。

```
'メッシュ' + '農業' + '気象'   # 'メッシュ農業気象'
'農業' * 3  # '農業農業農業'
```

等号（=）を用いて数値や文字列に名前をつけることができます。名前をつける文を代入文と呼びます。

```
lat = 36.0270
lon = 140.1104
site_name = 'noukanken'
```

名前を使った四則演算もできます。

```
site_name * 2 # 'noukankennoukanken'
lat + 5       # 41.027
```

代入文と演算子を組み合わせた代入文を累算代入文といいます。右辺と左辺の値に演算子を適用し、その結果を左辺に代入します。

```
a = 100
a += 10  # a に a + 10 の結果を代入、a = a + 10 と同じ
a      # 110

b = 1000
b -= 100  # b に b - 100 の結果を代入、b = b - 100 と同じ
b       # 900

c = 20
c *= 5  # c に c * 5 の結果を代入、 c = c * 5 と同じ
c     # 100

d = 100
d /= 20   # d に d / 20 の結果を代入、 d = d / 20 と同じ
d       # 5.0
```

4) 比較演算子

比較演算子は以下の表に示す通りです。条件が成立すれば True、成立しなければ False が返ります。

表 3 比較演算子

演算子	比較条件
x < y	x は y より小さい
x <= y	x は y より小さい、もしくは x と y は等しい
x > y	x は y より大きい
x >= y	x は y より大きい、もしくは x と y は等しい
x == y	x と y は等しい(等価である)
x != y	x と y は等しくない（等価でない）
x is y	x と y は同じオブジェクトである
x is not y	x と y は同じオブジェクトではない
x in y	x は y に含まれる
x not in y	x は y に含まれない

以下に比較演算子の例の利用例を示します。

```
a = 100
b = 200

a < b               # True
a <= b              # True
a > b               # False
a >= b              # False
a == 100            # True
a != 200            # True
a is 100            # True
a is not 100        # False
'2' in '12345'      # True
'2' not in '12345' # False
```

Python では比較演算子を連結して記述できます。

```
0 < a < 50    # False
0 < b <= 200  # True
```

5) リスト

リストは `[要素, 要素, ... ]` のように角括弧の中に要素を半角カンマで区切って格納したものです。リストでは要素の追加と削除、要素の入れ替えが可能です。また、異なる型の要素を格納したり、要素のないリストを作成したりできます。

```
# 要素が整数値のリスト
values = [1, 2, 3]
values  # [1, 2, 3]

# 要素が文字列のリスト
fruits = ['apple', 'orange', 'banana']
fruits  # ['apple', 'orange', 'banana']

# 要素の種類が混在するリスト
mixed = [1, 2.0, 'banana', values]
mixed  # [1, 2.0, 'banana', [1, 2, 3]]

# 要素がないリスト
blank_list = []
```

リストは結合できます。

```
fruits + values # ['apple', 'orange', 'banana', 1, 2, 3]
```

要素の追加は 組み込み関数 append()を使います。

```
fruits.append('mango')
fruits # ['apple', 'orange', 'banana', 'mango']
```

乗算の記号「*」を用いることにより要素を繰り返したリストを得られます。

```
fruits * 2  # ['apple', 'orange', 'banana', 'mango', 'apple', 'orange',
'banana', 'mango']
```

リストの要素は抽出できます。数値を入れると要素が 1 つだけ取り出せます。

```
fruits[0]  # 'apple'
```

リスト[m:n] のように数値をコロンでつないだ場合、m 番目から n-1 番目の要素を取り出します。

```
fruits[0:2]  # ['apple', 'orange']
```

変数 mixed のように入れ子のリストの中の要素を取り出す場合は「リスト[m][n]」のように、外側の入れ子の位置(m)、内側の入れ子の位置(n)を指定します。

```
mixed[3][2]  # 3
```

リストに特定の要素が入っているかを確認するときは in を使います

```
'orange' in fruits  # True
```

```
5 in values  # False
```

リストの長さ（要素数）を知りたいときには組み込み関数 len を使います

```
len(mixed)  # 4
```

6) タプル

タプルは (要素, 要素, ...) のように丸括弧の中に要素を半角カンマで区切って格納したものです。タプルは、リストと異なり更新不能です。

```
# 要素が整数値のタプル
values = (10, 20, 30)
values  # (10, 20, 30)

# 要素が文字列のタプル
vegetables = ('spinatch', 'carrot', 'onion')
vegetables  # ('spinatch', 'carrot', 'onion')

# 要素の種類は混合可能
mixed_tuple = ('spinatch', 100, 5.0)
mixed_tuple  # ('spinatch', 100, 5.0)

# 要素のないタプル
blank_tuple = ()
blank_tuple # ()
```

タプルはリストと同様、「+」による結合や「*」による繰り返し、要素の抽出、要素の確認、要素数の確認などが可能です

```
values + vegetables  # (10, 20, 30, 'spinatch', 'carrot', 'onion')
values * 3         # (10, 20, 30, 10, 20, 30, 10, 20, 30)
vegetables[2]      # 'onion'
10 in values       # True
```

ただし、タプルに要素を組み込み関数 append()で要素を追加することはできません。以下のコードを実行すると以下のエラーメッセージが戻ってきます。

```
vegetables.append('tomato')
```

7) 辞書

辞書は要素のキーとキーに対応する値を「:」でつないで 1 セットとして表し、{キー:値, キー:値, ...} の中に「キー:値」のセットを半角カンマで区切って格納したものです。

```
fruits_price = {'apple':100, 'banana':50, 'orange':80}
fruits_price  # {'apple': 100, 'banana': 50, 'orange': 80}
```

空の辞書を作ることもできます。

```
blank_dict = {}  # 空の辞書
```

辞書はリストやタプルと違い、各要素に順番を持ちません。値はキーを指定して検索します。

```
fruits_price['apple']  # 100
```

値は「辞書名[キー] = 値」という形式で指定することによって追加できます

```
fruits_price['avocado'] = 140
fruits_price  # {'apple': 100, 'avocado': 140, 'banana': 50, 'orange': 80}
```

8) インデントとブロック

Python ではインデントは見た目の問題だけではなく構文として意味があります。9）で説明する制御構文や 10）出説明する関数など、ひとまとまりで扱うべき文(＝ブロック、コードブロック)を記述する際、インデント（字下げ、Python では通常半角スペース 4 つ）の位置で表現します。図 1 は if 文（条件分岐）の記述例です（Visual Studio Code というエディタの表示を例として示しています）。図 1A の例では条件式 A が成り立つ場合は処理 1 以下が実行されますが、成り立たない場合は一切実行されません。一方、図 1B では条件式 A が成り立たなくても条件式 B は評価されます。

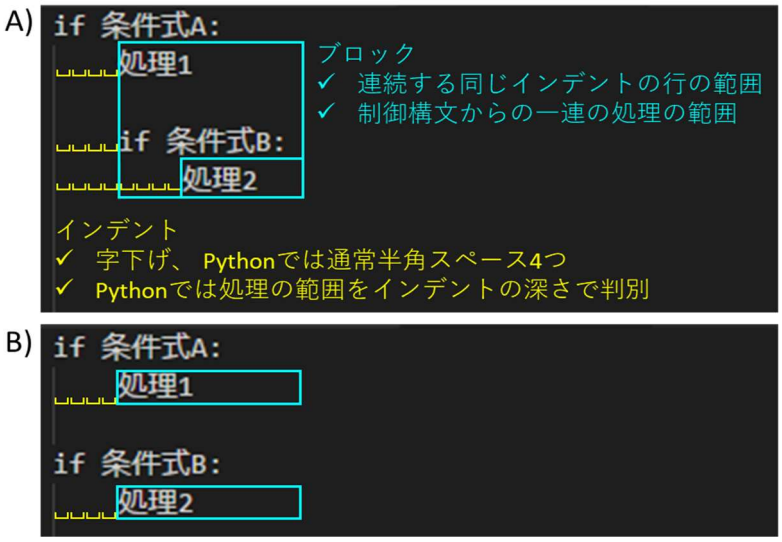

図 1.インデントとブロック

9) 制御構文

コードの実行順を条件によって分けたり繰り返したりするコードの構造のこと制御構文と呼びます。ここでは条件分岐と繰り返し（ループ）、について説明します。

for 文による繰り返し（ループ）

繰り返しは for 文という構文を使って以下のように書きます。

```
for 変数名 in イテラブルオブジェクト:
    処理
```

変数名	イテラブルオブジェクトから選択された値を格納する変数の名前
イテラブルオブジェクト	リスト、タプル、文字列など要素を順番に取り出せるもの
処理	変数（＝変数名）を使っておこないたい処理

以下に例を示します。

```
fruits = ['apple', 'banana', 'orange']

for fruit in fruits:
```

```
    print(fruit)  # 変数 fruit に格納された値を出力

# apple
# banana
# orange
```

3 行目の「print(fruit)」は for 文の条件の内側にあるため、ここではリスト fruits から選ばれた 1 つめ値を fruit に格納して出力、2 つめの値を fruit に格納して出力、3 つめの値を fruit に格納して出力、というように、リストから 1 つずつ値を選択して変数に格納して出力します。タプルでも同じことができます。

```
fruits = ('apple', 'banana', 'orange')

for fruit in fruits:
    print(fruit)  # 変数 fruit に格納された値を出力

# apple
# banana
# orange
```

for ループと range()という組み込み関数（特別な手続きなしに Python を起動すれば呼び出せる関数）を組み合わせて処理を特定の回数繰り返すことがよくおこなわれます。range()関数に引数として整数 x を与えると、ゼロから x - 1 までの x 個の整数のリストと同様に扱われます。

```
for i in range(3): # range(3)は [0, 1, 2] と同じ
    print(i)

# 0
# 1
# 2
```

range()関数には range(x, y, z)のように 3 つの引数を与えることもできます。x が初期値、y が上限 or 下限値（それぞれ正 or 負の場合）、z が数値の増分です。3 つめは省略可能で、省略した場合は増分が 1 となります。

```
for i in range(1, 10, 2):  # 2 は省略可能（省略すると増分は 1 となる）
    print(i)

# 1
```

```
# 3
# 5
# 7
# 9
```

なお、while 文という繰り返しの構文もありますが、メッシュ農業気象データの解析には使わないためここでは説明しません。

リスト内包表記

リスト内包表記は簡潔なコードでリストを作成する仕組みです。リスト内包は以下の形式で書きます。

```
[式 for 変数名 in イテラブルオブジェクト]
```

式	変数を含む式
変数名	イテラブルオブジェクトから選択された値を格納する変数の名前
イテラブルオブジェクト	リスト、タプル、文字列など要素を順番に取り出せるもの

リスト内包表記を使うと使わない場合に比べて少ないコードで記述できます。

```
# イテラブルオブジェクト（リスト）の作成
cities = ['Tokyo', 'New York', 'Paris']

# リスト内包表記を使った場合
[len(city) for city in cities]  # [5, 8, 5]

# リスト内包表記を使わない場合
city_len = []
for city in cities:
    city_len.append(len(city))

city_len  # [5, 8, 5]
```

条件分岐 if, elif, else

条件分岐は if, elif, else という構文を使って以下のように書きます。条件式のうしろには半角コロン 「：」がつきます。

```
if 条件式 1:  # 条件式 1 が真(True)場合
```

```
    処理 1    # 条件式 1 が真(True)の場合の処理

elif 条件式 2:  # 条件式 1 が偽(False)、条件式 2 が真(True)の場合
    処理 2     # 条件式 1 が偽(False)、条件式 2 が真(True)の場合の処理

else:  # 条件式 1・条件式 2 が偽(False)の場合
    処理 3  # 条件式 1・条件式 2 が偽(False)の場合の処理
```

以下に例を示します。

```
value = 50

if value <= 0:
    print('value = ', value, '<=0')
elif 0 < value < 100:
    print('0<= value =', value, '<100')
else:
    print('100 <= value = ', value)

# 0<= value = 50 <100
```

10) 関数

関数とはプログラムのいくつかの処理をひとまとめにしたものです。以下のように書きます。

```
def 関数名(引数, キーワード引数 = 値):
    処理

    return リターン
```

関数名	関数の名前（既存の関数の名前と重複しないように注意）
引数	関数の中で利用する変数のうち、呼び出し時に値（数値、文字列、リスト等）を指定したいもの（省略可）
キーワード引数	引数のうち初期値（数値、文字列、リスト等）を与えておきたいもの（省略可）
処理	関数の中でしたい処理
リターン	戻り値。関数を実行し終えた時に出力したい内容(省略可)

以下の例では 引数 a とキーワード引数 b をもつ calc_add という関数を作成しています。

```
def calc_add(a, b=3):
```

```
    c = a + b
    return c
```

作成した関数に引数と同じ数の値を与えて実行すると、戻り値が得られます。

```
calc_add(10, 20)  # 30 (a = 10, b = 20)
```

b には初期値として 3 が与えられていますが、引数として値を与えると上書きされます。
関数の戻り値を変数に格納することもできます。

```
return_value = calc_add(10, 20)
return_value  # 30 (a = 10, b = 20)
```

a には初期値が与えられていませんが、b には初期値として 3 が設定してあるため、引数を 1 つだけ与えるとその値は a に格納されて処理が実施されます。

```
calc_add(1)  # 4 (a = 1, b = 3)
```

11) ライブラリのインポート

Python の関数などを拡張子が「py」のテキストファイルにまとめて記述したものをモジュール、複数のモジュールをディレクトリにまとめたものをパッケージと呼びます。Python には特別な準備なしにそのまま利用できる「組み込み関数」や「組み込み型」とよばれるものがありますが（たとえば `print()` や `range()` は組み込み関数）、組み込みでないモジュール・パッケージは import 文で読み込んでから利用する必要があります。import 文で読み込めるモジュール・パッケージを Python ではライブラリと呼びます。

import 文には複数の書き方があります。たとえば、`AMD_Tools3.py` では以下のようなインポート文が書かれています。

```
from os import unlink
from os.path import join,exists,isdir
from datetime import datetime as dt
import numpy as np
```

ライブラリをインポートする方法

「`import` ライブラリ名」でライブラリを呼び出せます。インポートしたライブラリの中の関数は以下のようにピリオドでつないで呼び出せます。

```
# 「numpy」をインポート
import numpy
```

```
# Numpy というライブラリの中の zeros 関数を呼び出し（zeros については後述）
numpy.zeros((2, 3))

#  array([[ 0.,  0.,  0.],
#         [ 0.,  0.,  0.]])
```

略称を指定してライブラリをインポートする方法

ライブラリをインポートする際は略称を指定することができます。

```
# numpy というライブラリを略称「np」を指定してインポート
import numpy as np

# numpy のかわりに np を利用して関数を呼び出し
np.zeros((2, 3))

#   array([[ 0.,  0.,  0.],
#          [ 0.,  0.,  0.]])
```

ライブラリ内の関数を明示的にインポートする方法

ライブラリ内での関数を指定してインポートすることもできます。

```
# math というライブラリ内の floor という関数をインポート
from math import floor

x = 3.6
floor(x)  # 3 （floor は x 以下の最大の整数値を返す関数）
```

以下の方法でも同じ結果が得られますが、関数名 floor の前にライブラリを示す math をピリオドでつないでつける必要があります。

```
import math

x = 3.6
math.floor(x)  # 3
```

import 以下に複数の関数をカンマ区切りで列記することができます。

```
from math import floor, ceil
```

```
x = 4.7
floor(x), ceil(x)  # (4, 5) （ceil は x 以上の最小の整数値を返す関数）
```

略称を指定してライブラリ内の関数を明示的にインポートするして方法

関数に略称をつけて利用することもできます。

```
from math import floor as fr

x = 5.2
fr(x)  # 5
```

メッシュ農業気象データシステムで利用するライブラリ

メッシュ農業気象データシステムでは以下のライブラリを用いています。ライブラリは最初から Python に含まれている標準ライブラリと、別途入手が必要なサードパーティ製ライブラリとに分かれます。いずれのライブラリを使う際は import 文を書く必要があります。

標準ライブラリ

os

https://docs.python.jp/3/library/os.html

os 依存の機能を利用するために利用するモジュール

os.path

https://docs.python.jp/3/library/os.path.html

パス名を操作するのに便利な関数が含まれるモジュール

datetime

https://docs.python.jp/3/library/datetime.html

日付や時間データを操作するためのモジュール

math

https://docs.python.jp/3/library/math.html

各種数学関連関数を扱うモジュール

サードパーティ製ライブラリ

Matplotlib

https://matplotlib.org/

データ可視化（グラフ作成）に用いるライブラリ

netCDF4

http://unidata.github.io/netcdf4-python/
NetCDF(Network Common Data Form) のライブラリにアクセスするための Python インターフェイスライブラリ

NumPy
http://www.numpy.org/
数値計算を効率的に行うためのライブラリ

palettable
https://jiffyclub.github.io/palettable/
カラーパレット用ライブラリ

pyproj
https://pypi.python.org/pypi/pyproj
座標変換ライブラリ

2. NumPy

NumPy は Numerical Python の略語で、数値計算を効率的に行うためのライブラリです。NumPy により型が均一（たとえば浮動小数点数のみ、整数のみなど）な多次元配列（行列など）の操作が容易になります。

1) NumPy のインポート

NumPy を使う際は、まず NumPy のインポートが必要です。公式ページでも使われている以下の記法が広く使われています。

```
import numpy as np
```

2) ndarray

NumPy の配列は ndarray と呼ばれ、Python のリストとは異なります。ndarray は以下のように生成することができます。

```
list_data = [1, 2, 3]  # リスト
ndarray_data = np.array(list_data)  # リストを ndarray に変換
```

中身を確認すると ndarray は aray()で囲まれていることから、リストとは異なることがわかります。

```
list_data  # [1, 2, 3]
ndarray_data  # array([1, 2, 3])
```

NumPy は値がすべてゼロの ndarray を作ることもできます。np.zeros((行, 列)) として作成します。

```
# 値がすべてゼロだけの行列の ndarray
np.zeros((2, 3))  # 丸括弧が二重になっていることに注意

# array([[ 0.,  0.,  0.],
#        [ 0.,  0.,  0.]])
```

3) ndarray の計算

次にリストと ndarray の計算方法の違いを見るために、list_data と ndarray_data を使って同じような計算をおこない、結果を比較します。

```
# 加算
```

```
print('加算')

# リストの加算結果を出力
print('リスト', list_data + list_data)

# ndaray の加算結果を出力
print('ndarray', ndarray_data + ndarray_data)

# 加算
# リスト [1, 2, 3, 1, 2, 3]
# ndarray [2 4 6]
```

リストでは 2 つのリストが結合されているのに対し、ndarray では要素毎の足し算がおこなわれています。また、リストの減算、乗算、除算はできませんが、ndarray はできます。加算と同じく、要素毎に計算がおこなわれます。

```
# 減算
print('減算', ndarray_data - ndarray_data)

# 乗算
print('乗算', ndarray_data * ndarray_data)

# 除算
print('除算', ndarray_data / ndarray_data)

#  減算 [0 0 0]
#  乗算 [1 4 9]
#  除算 [ 1.  1.  1.]
```

4)　多次元配列と要素の計算

多次元配列は角括弧 []を入れ子にすることによって表現します。以下に 2 次元配列の例を示します。

```
array_2d = np.array([[1, 2, 3], [10, 20, 30], [100, 200, 300]])
array_2d



# array([[  1,   2,   3],
#        [ 10,  20,  30],
```

```
#       [100, 200, 300]])
```

特定の行を取り出す場合は、ndarray[行番号, :] のように指定します。

```
array_2d[0, :]  # array([1, 2, 3])
```

ここでコロンはすべての列を選択することを意味します。「, :」 は省略可能です。

```
array_2d[0]  # array([1, 2, 3])
```

特定の列を取り出す場合は ndarray[:, 列番号] のように指定します。

```
array_2d[:, 0]  # array([  1,  10, 100])
```

ここでコロンはすべての行を選択することを意味します。「: ,」は省略できません。
コロンを整数に置き換えると特定の位置の数値を選択することができます。以下の例では 2 行 3 列の数値を取り出しています。

```
array_2d[1, 2]  # 30
```

3. 参考文献

「Python 文法詳解」(2014) 石本敦夫著（オライリージャパン）
「いちばんやさしい Python の教本 人気講師が教える基礎からサーバサイド開発まで（「いちばんやさしい教本」シリーズ) (2017) 鈴木たかのり、杉谷弥月、株式会社ビープラウド著（インプレス）
「Python によるデータ分析入門 —NumPy、pandas を使ったデータ処理」(2013) Wes McKinney 著、 小林 儀匡・鈴木 宏尚・瀬戸山 雅人・滝口 開資・野上 大介訳（オライリージャパン）

4. 参考 URL

Python 3.6.3 ドキュメント https://docs.python.jp/3/index.html
Python 3.6.3 ライブラリーリファレンス https://docs.python.jp/3/library/index.html
Python Boot Camp テキスト http://bootcamp-text.readthedocs.io/textbook/index.html
NumPy http://www.numpy.org/