コンピュータリテラシ発展 ～Pythonを学ぶ～

第4回：Pythonを始めよう

情報学部 情報学科 情報メディア専攻
清水 哲也 (shimizu@info.shonan-it.ac.jp)

今回の授業内容

今回の授業内容

- 前回の課題解説
- 定義した処理を実行する
- ファイルを機能ごとに分けて再利用する
- 例外処理
- 課題

前回の課題解説

前回の課題解説

- 前回の課題の解答例を示します
- 解答例について質問があればご連絡ください

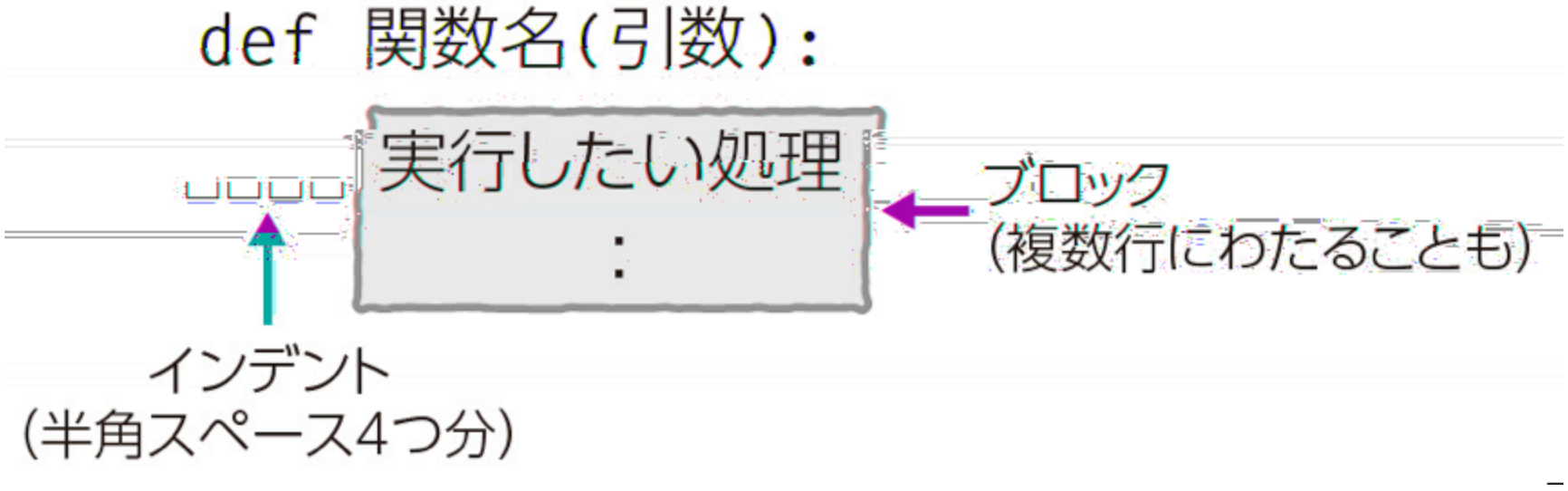
解答例

https://colab.research.google.com/drive/1N5mN8JCrI6VmaIpDBXVx392WOn1gUUDn?usp=sharing

定義した処理を実行する

定義した処理を実行する

- Pythonの関数は「`def`」を使って定義することができる
- `def` は，definition(「定義」という意味の英単語)の省略形です
- `if` 文や `while` 文と同じようにインデントが必要です
- 処理が複数行にわたる場合は2行目以降もインデントが必要になります



関数にわたす情報・関数から戻ってくる情報

- **引数**：関数`f()`の`()`内にあるオブジェクトを引数と呼びます
- 引数はその関数への入力値（関数にわたす情報）です
- 「`,`」で区切ることで複数の引数を関数にわたすことができます
- 引数をわたす方法は「**位置引数**」と「**キーワード引数**」の2種類があります
- **位置引数**：引数の順番によって引数を関数にわたす
- **キーワード引数**：引数名を指定した形式でわたす

関数にわたす情報・関数から戻ってくる情報

```
def f(x, y):
  return 2 * x + y

# 位置引数の場合
i = f(10, 20)        # f(20, 10)は結果が異なる
print(i)

# キーワード引数
j = f(x=10, y=20)    # f(y=20, x=10)でも同じ結果になる
print(j)
```

関数にわたす情報・関数から戻ってくる情報（オプション引数）

- 引数をあらかじめ指定することで関数を呼び出す時に引数を省略することができます
- こうした引数を **オプション引数** と呼びます
- オプション引数を指定していても関数を呼び出す時に引数を指定することで通常の引数と同じ処理ができます

```
def double(x = 2):
  return x * 2

# 関数に引数をわたさないとオプション引数が使用される
i = double()
print(i)

# 関数に引数をわたすとその値が使用される
j = double(3)
print(j)
```

関数にわたす情報・関数から戻って来る情報（戻り値）

- **戻り値** ：関数が何を出力するかを定義するものです
- `return` の後に書きます
- `return` を定義しない関数は，戻り値がないということになり「**None**」を返します
- Noneは，値が存在しないことを意味します

変数が使える範囲

- **変数** を関数のブロック内で定義するか，関数のブロック外で定義するかによって，**変数が使える範囲（スコープ）が異なります**
- **グローバル変数** ：関数のブロック外で定義される変数，プログラム内のどこからでも呼び出すことができます
- **ローカル変数** ：関数のブロック内で定義される変数，その関数内でのみ呼び出すことができます

変数が使える範囲（グローバル変数）

```
global_number = 10  # グローバル変数の定義

def show():
  print('グローバル変数の値：', global_number)

# グローバル変数の表示
show()

# グローバル変数の値を変更
global_number = 5
# グローバル変数の表示
show()
```

変数が使える範囲（ローカル変数）

```
def show():
  local_number = 5
  print('ローカル変数の値：', local_number)

# ローカル変数の表示
show()

# local_numberを自体を呼び出して表示
print(local_number)
```

この場合エラーが表示されると思います

```
NameError: name 'local_number' is not defined
```

ローカル変数を範囲外で参照したのでエラー（定義されていない）が発生します

あらかじめ用意されている関数

- `print()` や `str()` などの関数はあらかじめ用意されている関数です
- これらを **組み込み関数** と呼びます

関数名	概要
`print()`	引数の値を画面に出力する
`input()`	プロンプトを表示してユーザからの入力を受け付け，結果を文字列として返す
`len()`	渡したオブジェクトの持つ要素の個数を返す
`type()`	引数のデータ型を返す

組み込み関数：https://docs.python.org/ja/3/library/functions.html#builtinfunctions

ファイルを機能ごとに分けて再利用

ファイルを機能ごとに分けて再利用する

- Pythonではプログラムを便利に書くために **モジュール** と呼ばれる仕組みがあります
- モジュールは複数の関数を再利用できるようにしたものです
- 多くの先人が作った便利な関数を利用することができます
- 使いたいモジュールをPythonに教えてあげることを **インポート** と呼びます
- インポートすると，モジュール内の変数や関数を利用することができます
- インポート：「**import モジュール名**」
- モジュール内の関数：「**モジュール名.関数名**」
- インポート対象を指定：「**from モジュール名 import 関数名**」
- モジュール名を変更してインポート：「**import モジュール名 as 別名**」

使いたいファイルを読み込ませる

- 自分や他人が作った有用なプログラムを再利用します
- ここでは簡単な例を示します
- 自作モジュール（hello.py）

```
def many_times(count):
  s = count * 'hello '
  print(s)
```

- Moodleから「hello.py」ファイルをダウンロードしてください
- ColabでGoogle Driveをマウントしてください
- 今開いているノートと同じ場所（フォルダ）に「hello.py」をアップロードしてください

使いたいファイルを読み込ませる

- 自作モジュールを使用する側

```
import sys
ROOTPATH = '/content/drive/MyDrive/(開いているノートがあるフォルダ)'
sys.path.append(ROOTPATH)
import hello

hello.many_times(3)
```

- 先ほどアップロードした「hello.py」を読み込むためにファイルがある場所（パス）を指定します
- 「hello.py」で定義されている関数 `many_times` を利用することができます

使いたいファイルを読み込ませる

- 自作モジュールを使用する側
- インポート対象を指定することもできます
- 今回は `many_times()` 関数のみをインポートします

```
import sys
ROOTPATH = '/content/drive/MyDrive/(開いているノートがあるフォルダ)'
sys.path.append(ROOTPATH)
from hello import many_times

many_times(3)
```

使いたいファイルを読み込ませる

- 自作モジュールを使用する側
- 名前を変更してインポートします
- helloモジュールをhiとしてインポートします

```
import sys
ROOTPATH = '/content/drive/MyDrive//(開いているノートがあるフォルダ)'
sys.path.append(ROOTPATH)
import hello as hi

hi.many_times(3)
```

あらかじめ用意されているライブラリを使う

- 標準ライブラリ：Pythonのインストール時に同梱されているライブラリ
- https://docs.python.org/ja/3/library/index.html
- Colabでは標準ライブラリ以外にもたくさんのライブラリが用意されています
- `!pip list` で確認できます
- 例：標準ライブラリ内の日付や時刻を扱うdatetimeモジュールを使って，実行日の日付を出力するプログラム

```
import datetime

date = datetime.date.today()
print(date)
```

例外処理

例外処理

- 例外とは実行中に検出されたエラーを指します
- Webスクレイピングなどのプログラムは将来変更される可能性が高いため、壊れずに動くことは保証されません
- プログラムがエラーに遭遇した場合，事前に指示を与えることが重要です
- 例外処理を導入することでプログラムがエラーに対処する方法を指定できます

`try, except` 文

- 例外処理の基本は `try` と `except` 文で構成されます
- `try` ブロックには例外が発生する可能性のあるコードを記述
- 例外が発生しない場合：`try` ブロックの処理が正常に実行
- 例外が発生する場合：`except` ブロックに指定した例外の種類に応じた処理が実行
- 例外処理を使わない場合，例外が発生するとプログラムは中断される可能性があります

`try, except` 文

例外処理がない場合のプログラム

```
fruits = ['apple', 'orange', 'banana']
input_value = input('取り出したいフルーツの番号を教えてください：')

print(fruits[int(input_value)])
```

- このプログラムでは「-3〜2」の値のみ正常になります
- それ以外の数値や文字を入力するとエラーになります

`try, except` 文

例外処理を行うプログラム

```
fruits = ['apple', 'orange', 'banana']
input_value = input('取り出したいフルーツの番号を教えてください：')

try:
  print(fruits[int(input_value)])
except IndexError as e:
  print('catch IndexError:', e)
except ValueError as e:
  print('catch ValueError:', e)
```

`try, except` 文

▼正常処理

```
取り出したいフルーツの番号を教えてください: 1
orange
```

▼異常処理パターン1：配列のインデックス外の値（100）を指定する

```
取り出したいフルーツの番号を教えてください: 100
catch IndexError: list index out of range
```

▼異常処理パターン2：整数以外の値（sample）を入力する

```
取り出したいフルーツの番号を教えてください: sample
catch ValueError: invalid literal for int() with base 10: 'sample'
```

- 「except 例外型 as 変数名:」変数に例外オブジェクトを格納することができます
- 「except 例外型:」でも可能です

課題

課題

- Moodleにある「SCfCL-4th-prac.ipynb」ファイルをダウンロードしてColabにアップロードしてください
- 課題が完了したら「File」>「Download」>「Download .ipynb」で「.ipynb」形式でダウンロードしてください
- ダウンロードした **.ipynbファイル** をMoodleに提出してください
- 提出期限は **5月16日(木) 20時まで** です