公式問題集によるこの章(2つの章を合わせた)のインデックス (徹底攻略Python 3 エンジニア認定［基礎試験］問題集 – 2023/3/9)

- OSの操作
- ワイルドカード
- コマンドライン引数
- 数学.統計
- 日付と時刻
- 単体テスト
- ネットアクセス
- 出力データの整形

OSモジュール

- ! `from os import *` は避けること！**必ず** `import os` **を使う**こと！
  - 理由は、`osモジュールのopen()` と `ビルトイン関数のopen()` は違うのである
    - `from os import *` だと、os.open()がビルトインのopen()を隠してしまい、意図しない動作になる
- **os.getcwd()** ⇦ get current working directory
  - current working directory ⇦ 今作業しているフォルダのこと
  - つまり「今いるフォルダはどこ？」を教えてくれる
- **os.chdir()** ⇦ change directory
  - 作業フォルダの移動
    - "." : 現在のフォルダ
      - 現在のフォルダへ移動する意味はないが、この書き方は、他の所で役に立ちます
    - ".." : 一つ上のフォルダ
    - "aru_folder" : aru_folderへ移動（あるフォルダをイメージ）
      - この書き方だと、一つ下の「aru_folder」へ移動する
  - `os.getcwd()` で、確かに移動しているか確認できる
  - `os.listdir(".")`
    - 現在のフォルダに含まれる「フォルダ・ファイル」を表示する
      - これで、「道に迷った時(どのパスにいるか分かりづらい時)」便利です
- **os.system()**
  - コンピュータのシステムコマンドを実行する
    - Windowsの場合、コマンドプロンプトのコマンドを文字列で渡す
    - Windows以外は、分かりませんが、類推できると思います
    - 例(Windows)
      - os.system('mkdir today') # フォルダ「today」を作成

```
"""
＊ このコメントは、Markdownで書き加えたものです！

Cドライブ
  py_os_module_test          # 最初のカレントディレクトリ
     1st                     # todayフォルダが、下のコードで作成される
         2nd, text2.txt
"""

# 以下は、コマンドプロンプトで実行した例です
# 最後の`0`はコマンド成功と言う意味です

>>> import os
>>> os.getcwd()
'C:\\py_os_module_test'
>>> os.listdir(".")
['1st']
>>> os.chdir("1st")
>>> os.getcwd()
'C:\\py_os_module_test\\1st'
>>> os.listdir(".")
['2nd', 'text2.txt']
>>> os.chdir("..")
>>> os.getcwd()
'C:\\py_os_module_test'
>>> os.system("mkdir today")
0
>>>
```

大きなモジュールの扱い方

- ビルトインの、dir()関数、help()関数を使おう
  - dir()関数
    - 下の実行例では、osモジュールの関数がすべて入ったリストを返す
  - help()関数
    - 下の実行例では、モジュールであるファイルに書かれている関数の説明("""～""")が、多数返される
      - 関数の説明： `docstring` と言う
  - 下の実行例での `...` は、省略を意味している

```
# コマンドプロンプトで実行
# import os を忘れないこと！
# dir(os)は、2,30行ある感じ（文字の大きさや、折り返しによって違ってくる）
# help(os)は、下の`--- More ---`で、Enterキーを押すと下から残りがどんどん出てくる
```

```
>>> import os
>>> dir(os)
['DirEntry', 'EX_OK', 'F_OK', 'GenericAlias', 'Mapping', ... 'write']
>>>
>>> help(os)
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

MODULE REFERENCE
    https://docs.python.org/3.13/library/os.html

    The following documentation is automatically generated from the

...

-- More  --
```

shutilモジュール

- shutil ⇦ shell utilities（シェルの便利道具の意味）
- osモジュールは、低レベル(コンピュータを直接扱う感じ)
- `shutilモジュール`は、高レベル(使いやすい・日常的なファイル操作な感じ)

shutilモジュールのメソッド例

- `shutil.copyfile()` 公式本に載っていたメソッド

```
# shutil.copyfile()メソッド

import shutil

# two.txtは新たに作成され、内容もコピーされた
shutil.copyfile("one.txt", "two.txt")
```

- その他： Windowsで、よくおこなう`コピー・移動・削除`をPythonから行うことが出来るようになる

globモジュール

glob.glob()メソッド

- 公式本にあるのは、これだけである
- `*`を、ワイルドカードとして使う

概要

- やっていることのイメージは、`Windowsなどの、エディタで、検索すること` です

```
import glob
import os  # osモジュールでは、必ずこの形（from～は使わない）

# 2025年5月にコレクションした写真を探す
#   .jpg 等を探すべきですが、コード例ということで、.txt を使います

# まずは、すべてのファイルを探す
#   これは本来必要ないと思われるが、サンプルコードなので
print(os.listdir("."))
"""
['glob_test.py', '家族旅行_25-0505.txt', '珍しい蝶_25-0510.txt',
 '町内清掃_25-0425.txt', '花見_25-0315.txt', '飲み会_25-0515.txt']
"""

# 5月の写真(実はテキストファイル、25-05～.txt)を探す
#   5月の写真だけ見たかった場合
print(glob.glob("*05*.txt"))
"""
['家族旅行_25-0505.txt', '珍しい蝶_25-0510.txt', '飲み会_25-0515.txt']
"""
```

コマンドライン引数

sys.argv

- sysモジュールのargv(データ属性)
    - オブジェクトには、データ属性と関数属性がある（言い方は2,3通りある）
    - python(Windowsではpyでも可) 以降の、`文字列の並び` (sys.argvは数値でも文字列)
    - リストで扱う
- コマンドラインは、コマンドプロンプト・PowerShell で、Python実行時コマンド(命令文)の並び
    - 例：（Windowsでは、`py` でもOKだが、pythonを使う）
        - python `demo.py hello world 123` （hello、world、123 に意味はない）
- コードで説明します

```
# demo.py

import sys

# python ファイル名 コマンドライン引数(0以上)
```

```
#   ↑：sys.argv は、python 以降すべての文字列リスト
#   この例では `demo.py hello world 123`
print("コマンドライン引数:", sys.argv)

# [0] は、python の次なので、ファイル名になる
print("ファイル名:", sys.argv[0])

# ファイル名以降が、一つでもあれば、それらをリストにする
#   つまり、ファイル名の後にある、引き数はすべてリスト化 ← [1:]
#       [1:]は、リストのスライスを意味している
if len(sys.argv) > 1:
    print("追加の引数:", sys.argv[1:])

"""
argv_test> python demo.py hello world 123 ← コマンドプロンプトで、実行します
コマンドライン引数: ['demo.py', 'hello', 'world', '123']
ファイル名: demo.py
追加の引数: ['hello', 'world', '123']
"""
```

argparse

- 一言で言えば、「sys.argv」よりも、多機能な上、チェック機能も合わせ持つ

コード例（説明なし）

```
# Pythonチュートリアル第4版：p120より抜粋
# demo2.py

import argparse

parser = argparse.ArgumentParser(
    prog='top', description='Show top lines from each file'
)

parser.add_argument('filenames', nargs='+')

parser.add_argument('-l', '--lines', type=int, default=10)

args = parser.parse_args()

print(args)

"""
実行例（PyCharmのPowerShell）
ap> python demo2.py x.txt y.txt -l 8
Namespace(filenames=['x.txt', 'y.txt'], lines=8)
"""
```

コード例（コメントで説明）

```
# Pythonチュートリアル第4版：p120より抜粋
# demo2_copy.py

import argparse  # argparseモジュールのインポート

# parser変数名は文法的には任意である
#   Pythonプログラムで使うコマンドライン引数に関するオブジェクトになる
# prog：プログラム名、ファイル名とは違うことに注意
#   何をするのか分かりやすい名前が良い
#   この例の、top は、おそらくファイルの先頭行を意味してるが、
#   このプログラムは、設計段階で、実装はされていないようである
# description：プログラムの説明を書く
#   'Show top lines from each file'
#   「各ファイルの先頭数行を表示する」という意味
#    各ファイルの先頭行をみて、目的のファイルを探す時に使えそうな感じである
parser = argparse.ArgumentParser(
    prog='top', description='Show top lines from each file'
)

# 引き数の設定１：
# "filename"と紐づける（位置引数に対してなので、必須）
#  nargs="+"：1つ以上のファイル（+は正規表現の使い方に近い、数値でもOK）
# nargsを指定した場合引き数が一つでも実際の引数はリストとして紐づけられる
parser.add_argument('filenames', nargs='+')
# parser.add_argument('filenames')

# 引数の設定２：
# "-なになに"、"--なになに"：`-`か`--`が付くとオプションの引数になる
# この場合、default=10 が設定されているので、書かないとそれが採用される
# また、オプションなので、コマンドライン上のどこにあってもよい
#   ただし今回の場合、設定１で複数のファイルが使われる時は、
#   その間にいれると、[usage](使用法)が出てきて、注意される
# type=int なので、`-l=5` や `-l 8`(半角空けて直後) のように書く
# `-l` は、`--lines` の、代替方法でどちらを使ってもよい
#   `-l`は、簡単に書けるように用意されている
parser.add_argument('-l', '--lines', type=int, default=10)

# args変数は文法的には任意である
# parser.parse_args()の役割：
#       ・ 引数を解析し、定義された引数の形式に従っているかチェックと実際の引数との紐づけ
#   1. この場合、設定１・設定２が、正しく設定しているが調べる
#   2. 1.がパスすれば「args.設定名」の変数に引数の値が格納される
#       ↓print(args.filenames)
args = parser.parse_args()

print(args.filenames)
print(args)

# このプログラムを使う時は、当然PowerShellなどを使うべきである
"""
```

```
実行例（PyCharmのPowerShell）
ap> python demo2_copy.py -l=3 a.txt b.txt
['a.txt', 'b.txt']
Namespace(filenames=['a.txt', 'b.txt'], lines=3)
"""
```

【sysモジュール】 エラーの出力　リダイレクト　プログラムの修了

- ここでの重要用語：
  - `stdin`：standard input：標準入力
  - `stdout`：standard output：標準出力
  - `stderr`：standard error：標準エラー出力
    - 参考：
      - input()関数・print()関数も、デバイス入力も、「標準入力・出力」を使う
      - 標準でない例は、ファイルの入出力や、ネットワークからの入力、データベースの読み取り
- `sys.stderr` の特徴と用途
  - **目的**：警告やエラーメッセージの出力専用
  - **利点**：`stdout` がリダイレクトされていても、エラーメッセージが確実に表示される
  - **使用場面**：ログファイルへの出力時などで重要
- プログラムの終了
  - **方法**：`sys.exit()` を使用
  - **特徴**：最も直接的なスクリプト終了方法
  - **用途**：エラー発生時やプログラムの強制終了時

使用例

```python
# demo_stderr.py

import sys

# PowerShellで、` py demo_stderr.py > output.txt` とすれば、
# output.txtにもprint文は、書き込まれる

# これは標準出力（stdout）へ出力されます
print("このメッセージは stdout に送られます。")
print("リダイレクトするとファイルに書き込まれます。")

# これは標準エラー出力（stderr）へ出力されます
```

```
sys.stderr.write("このメッセージは stderr に送られます。\n")
sys.stderr.write("リダイレクトしても通常は画面に表示されます。\n")

print("stdout へのメッセージ、再び表示します。")
sys.stderr.write("stderr へのメッセージ、再び表示します。\n")
```

ポイント

- `stdout`と`stderr`を使い分けることで、出力の管理が容易になる
- `stderr`は通常の出力とは別扱いされるため、
  ログファイルへのリダイレクト時も画面にエラーが表示される
- `sys.exit()`は例外を発生させてプログラムを終了する

リダイレクトについて

リダイレクトの具体例（リダイレクトは、ここでは、ファイルに保存すること）

```
"""
日常的な例で考えると
- stdout：レポートの内容（ファイルに保存したい）
- stderr：「プリンターの紙が詰まりました」（すぐに知りたい）

プログラムでも同じ考え方：
- 処理結果 → ファイルに保存（stdout）
- エラー通知 → すぐ画面で確認（stderr）
"""
```

文字列検索　文字列置換

文字列検索

- `re`モジュール：正規表現を使って、文字列の処理を行う
- r"文字列"：raw文字列
  - エスケープシーケンスなどを気にすることなく、正規表現そのままを書くことが出来る
- `re.findall()`メソッド
  - マッチする部分文字列を、リストで返す
  - 見つからなかったら、空のリストを返す
- 以下のコードで使われている正規表現
  - `\b`
    - 単語の区切りを意味します（`\W`のように、1文字に一致するわけではありません）
    - 単語の前後どちらの境界もチェックします（先頭だけでなく末尾の境界も含む）

    - `w`
        - 「w」の文字そのもの
    - [a-z]*
        - 「abcdefghijklmnopqrstuvwxyz」の文字が、0以上
    - \bw[a-z]*
        - 区切られた単語 かつ wで始まる かつ wの後が「a~z」の0文字以上
            - 以下のコードでは「how」以外がリストで返されました

```
import re

result = re.findall(
    r"\bw[a-z]*", "w what why where how when who"
)

print(result)  # ['w', 'what', 'why', 'where', 'when', 'who']
```

文字列置換

- Pythonのstr型とそのメソッド
    - 標準ライブラリとして組み込まれているため、別途importせずにそのまま使用できます。
- `"文字列".replace()` メソッド
    - 使い方の例：
        - "あいうえお".replace("い", "い ") ➡ 「あい うえお」

```
string = "あいうえお"

new_string = string.replace("い", "い ")
print(new_string)
print(f"私の名前は、{new_string} です")

"""
あい うえお
私の名前は、あい うえお です
"""
```

数学

mathモジュール

- 浮動小数点数の数学用の関数にアクセスできる
    - 一言で言えば、数学のためのモジュール

コサインとラジアン

```python
# mathモジュール
import math

# math.pi → π
print(math.pi)  # 3.141592653589793


# ラジアン・コサイン関係

# 角度を度で定義
angle_degrees = 60

# math.cos()の引数は、ラジアンなので、変換する
angle_radians = (math.radians(angle_degrees))

# 角度60度に対するコサイン値を求める
cos_value = math.cos(angle_radians)

# コサイン値を出力
print(cos_value)  # 0.5000000000000001 (2進数の計算での誤差)

# フォーマット前
print(f"斜辺の長さが10mの時、隣辺の長さは{10 * cos_value}mである")
"""斜辺の長さが10mの時、隣辺の長さは5.000000000000001mである"""

# フォーマット後
#   フォーマットは、最後にすべきである
#       途中でフォーマットすると、数値と文字列の計算になってしまうので
# f"{〜:.2f}"：小数点2位まで
print(f"斜辺の長さが10mの時、隣辺の長さは{10 * cos_value:.2f}mである")
"""
斜辺の長さが10mの時、隣辺の長さは5.00mである
"""
```

コサインについて

直角三角形を思い浮かべてください。ある角度に注目したとき、
その角度を挟む2つの辺のうち、直角の向かい側にある一番長い辺を「斜辺（しゃへん）」
もう一方の辺を「隣辺（りんぺん）」と呼びます。

コサインとは、この「隣辺の長さ」を「斜辺の長さ」で割った値のことです。

つまり、角度が決まれば、その角度に対するコサインの値（隣辺 ÷ 斜辺の比率）も決まります。
この値は、三角形の大きさに関わらず、角度が同じなら常に一定になります。
例えば、ある角度のコサインが0.5だとしたら、
それは「隣辺の長さが斜辺の長さの半分ですよ」という意味になります。

このように、コサインは角度と直角三角形の辺の長さの関係を表す便利な道具の一つです。

追記：
コサインは特定の角度における辺の比率であり、それを使って辺の長さを計算できる
隣辺の長さ = 斜辺の長さ × コサイン のようにして、
斜辺の長さから、隣辺の長さを求めることが出来るのである

「特定の角度に対するコサインの値は定数である」と言えます。
例えば、
角度が0度（または0ラジアン）の場合、コサインの値は常に1です。
角度が60度（またはπ/3ラジアン）の場合、コサインの値は常に0.5です。
角度が90度（またはπ/2ラジアン）の場合、コサインの値は常に0です。

ラジアンについて

「度」と「ラジアン」の違いは、「メートルとフィートの違い」のように、
同じ「角度」という量を測るための異なる単位と考えることができます。

log (対数関数)

- 対数関数の考え方の基本は、コメントを見てください

```
import math

# log₂(1024) ➡ 10  ：  2を10乗すると1024
#   logx(y): xを何乗したらyになるかということ
# Pythonの場合：math.log(x, base) ➡ base を何乗したら x になるかを求める
print(math.log(1024, 2))  # 10.0

# 参考
print(f"2 ** 10 = {2 ** 10}")  # 2 ** 10 = 1024
```

randomモジュール

random.choice()メソッド

- シーケンスである引数から、無作為に一つの値を返す

```
import random

# chosen：choose(選ぶの受け身(be chosen)の略)
chosen = random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(chosen)
if chosen == 7:
    print("あなたはラッキーですね！")
```

```
# シーケンス：順序を持つデータ構造
#    タプル・文字列・rangeオブジェクトも、OKである
print(random.choice(("タ", "プ", "ル")))
print(random.choice("文字列"))
print(random.choice(range(4, 7)))

"""
# ラッキーな実行例

7
あなたはラッキーですね！
タ
列
4
"""
```

random.sample()メソッド

- random.sample(シーケンス, 個数) -> リスト
  - シーケンスの中から、指定された個数を選択します
  - 元のシーケンスは変更されません
  - リストを返します

```
import random

# 私の環境で絵文字が使えたので使いました
chosen = random.sample(["🍎", "🍒", "🍇"], 2)
print(chosen)  # ['🍎', '🍇']

# rangeオブジェクトを使う
chosen = (random.sample(range(1, 10), 3))
print(chosen)  # [2, 3, 6]
```

random.random()メソッド

- random.random() -> float
  - `0.0 <= 返される値 <1.0` である
    - つまり「0以上で 1 より小さい小数(float)」であるランダムな値が返される

```
import random

for num in range(1, 11):
    print(f"{num}回目：{random.random()}")
```

```
"""実行結果：
1回目：0.36797767283596283
2回目：0.37540999120656515
3回目：0.6408120671911439
4回目：0.011679022384529891
5回目：0.6512795547390425
6回目：0.8113890027958266
7回目：0.6364044106829057
8回目：0.29517788175442994
9回目：0.5656727335321161
10回目：0.18351560890681784
"""
```

random.randrange()メソッド

- 細かい点を除いて、`random.choice(range(start, stop, step))`と同じである
  - 試験対策としては、これで十分だと思われます

```
import random

# 0～9の中で値を返す
ran = random.randrange(10)
print(ran)
print()

for num in range(1, 11):
    ran = random.randrange(10)
    print(f"{num}回目：{ran}")

"""
実行結果：

6

1回目：3
2回目：8
3回目：8
4回目：1
5回目：0
6回目：4
7回目：8
8回目：5
9回目：1
10回目：9
"""
```

statisticsモジュール

- 数値データの基本統計量（平均、中央値、分散など）を求める
  - statistics.mean(データ)：平均値を求める
  - statistics.median(中央値)：中央値を求める
  - statistics.variance(分散)：分散を求める
- 中央値、分散
  - 中央値：昇順や降順に並べた中央の値、偶数個の時は中央の2つの値の平均である
  - 分散：データの散らばり具合のこと
    - なので、全て同じ値の時は`0`で、値が大きいほど個々のデータのばらつきがあることになる

```
import statistics

data = [8, 7, 4, 9, 3, 15]

# average: 平均
average = statistics.mean(data)
print(average)  # 7.666666666666667

# median: 中央値
median = statistics.median(data)
print(median)  # 7.5

# variance：分散（散らばりの度合いを示す指標）
variance = statistics.variance(data)
print(variance)  # 18.266666666666666

# すべて同じデータであると、分散値は0である
variance2 = statistics.variance((5, 5, 5, 5, 5))
print(variance2)  # 0
```

インターネットへのアクセス

urllib.requestモジュール

- URLにあるデータを取得するモジュール
  - もっと平たく言えば、「Webページ、画像、ファイルなど」を取得するモジュール
    - 以下のコードでは、HTMLコードを一括で取り込み、200文字だけ表示している
- `urllib.request.urlopen()`メソッド
  - ここでの引数は、アドレスの文字列 ("https://example.com/")
  - イメージとしては、ビルトインであるopen()関数を開いて、ファイルオブジェクトを取得する感じである

コード例 1

- 大ざっぱな流れ
  i. URLアドレスの文字列 (url = "https://example.com/") から、オブジェクトを取得する
    - ここでは、HTMLコードのオブジェクトといえる
  ii. オブジェクトを、read()メソッドで読み込む（一括でまとめて取得：html_bytes）
    - この状態では、人には読めない
      - バイト列：コンピュータがデータを扱うための基本的な形式（生のデータ）
  iii. decode()メソッドで、utf-8形式に変換(デコード)して、`文字列` にする
  iv. 全てを表示するのは長すぎるので、`文字列[:200]` と、スライスしている
    - 実行結果が尻切れトンボになっているのは、200文字で終わっているからである
  v. コードが成功したかを確認：
    - `responseオブジェクト.getcode()` が、`200` を返している
      - 200は、成功したことを意味している

```
import urllib.request

# アクセスしたいWebページのURL
url = "https://example.com/"

try:
    # 指定したURLを開いて、レスポンスオブジェクトを取得
    response = urllib.request.urlopen(url)

    # レスポンスからHTMLの内容を読み込む
    html_bytes = response.read()

    # バイト列をUTF-8で、デコードして文字列にする
    html_string = html_bytes.decode("utf-8")

    print(f"'{url}'から取得したHTMLの最初の200文字：")
    print(html_string[:200])  # 長すぎるので最初の200文字だけ表示する

    # ステータスコードの確認（200なら成功）
    print(f"\nステータスコード: {response.getcode()}")

except Exception as e:
    print(f"エラーが発生しました: {e}")

"""
実行結果（尻切れトンボになっているのは、最初の200文字なので）


'https://example.com/'から取得したHTMLの最初の200文字：
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
```

```
    <meta name="viewport" conten

ステータスコード: 200

Process finished with exit code 0
"""
```

コード例２

- 大ざっぱな流れ
  - コード例１と同じように、responseオブジェクトを取得している（with文を使っている）
    - このイメージは、open()関数で、ファイルオブジェクトを取得するのと似ている
    - そして、ビルトインのopen関数から取得するファイルオブジェクトにも似ており、イテラブルで、for文を使って1行ずつ取得できる
    - utf-8の文字列にするのは、「例１」と同様である
    - `文字列.startswith("<")`の意味は、文字列が、"<"で始まっていますか？ということである（bool値を返す）
    - `line.rstrip()`で、削除している文字は`改行文字`です
      - これで、print文の改行文字だけになり、空白行が出来てしまうのを防いでいます

```
from urllib.request import urlopen

with urlopen("https://example.com/") as response:
    for line in response:
        line = line.decode("utf-8")  # バイト列をutf-8の文字列にする
        # 行の始まりが、"<"であれば：
        if line.startswith("<"):
            print(line.rstrip())

"""
実行結果：
<!doctype html>
<html>
<head>
</head>
<body>
<div>
</div>
</body>
</html>
"""
```

smtplibモジュール

- 以下のコードの大まかな流れ
    i. import smtplib
    ii. SMTPクラスのオブジェクト生成 ➡ server変数
    iii. server.sendmail("メールの内容")

```
import smtplib

# これを、実際に動かすには、その他設定が必要である
# また、試験に対応することが、目的なので、実際には、もっとコードが膨らみます

# --- 設定値（実際の環境に合わせて変更してください）---
# SMTPサーバーのアドレス（例: "smtp.gmail.com", "localhost" など）
smtp_host = "ご自身のSMTPサーバーアドレス"
# SMTPサーバーのポート番号（例: 587, 465, 25, 1025 など）
smtp_port = 587  # 例: TLS接続の場合

# 指定されたSMTPサーバーとポートに接続するための
# SMTPクラスオブジェクトを作成します。
#       ここでは、server変数
server = smtplib.SMTP(smtp_host, smtp_port)

# 通信を暗号化します（必須ではありませんが、推奨されます）。
# 公式テキストにはなかったが、近年重要だと思われます
server.starttls()

# 作成したメールメッセージを送信します。
server.sendmail("from@example.com", "to@example.com",
                """To: to@example.com
                From: from@example.com

                Subject: テストメール

                本文：
                How are you doing these days?
                """)

# 接続を閉じます（リソースを適切に開放するために重要です！）
server.quit()
```

日付と時間

- 概要
    - 日付、時間を扱う
    - 日付や時間に対して、計算が出来る
        - 例えば、試験日まで何日あるかなど
    - 扱う目的に合わせて、柔軟な表示ができる

- タイムゾーンにも対応

```
import datetime

# datetime.dateクラスが、クラスメソッド`today()`を呼び出している
# nowは、datetime.dateオブジェクト
"""
>>> import datetime
>>> now = datetime.date.today()
>>> now
datetime.date(2025, 6, 2)
>>> type(now)
<class 'datetime.date'>
"""
now = datetime.date.today()

# ターミナルでの実行と表示が違うのは、__str__のオーバーライド
#   Python3基礎認定試験で、ここまで問うとは思わないが、
#   気にしておくと役に立つときが来ると思う
#   ちなみに、ターミナルでは、__repr__(representation)が使われている
print(now)  # 2025-06-01

# date.datetimeオブジェクトのstrftime()メソッド呼び出し
#   ニーズに合わせて整形できる
print(now.strftime(
    "%m-%d-%y. %d %b %Y is a %A on the %d day of %B."
))
"""
06-01-25. 01 Jun 2025 is a Sunday on the 01 day of June.
"""

# date.datetimeオブジェクトは、カレンダー計算をサポートしている
#   datetime.date(year, month, day)：コンストラクタ
#   ここでの計算は「試験日 - 本日」
the_day_I_will_take_the_exam = datetime.date(2025, 7, 31)

# remaining_days: 残りの日にち
remaining_days = the_day_I_will_take_the_exam - now
print(f"試験まであと{remaining_days.days}日です")
"""試験まであと60日です"""
```

- 参考（strftime()のフォーマット方法：暗記は無理だと思われるが雰囲気をつかもう）

```
%m: 06（月）
%d: 01（日）
%y: 25（年、下2桁）
.  : 文字列リテラル
%d: 01（日）
 : 文字列リテラル
%b: Jun（月の省略名）
 : 文字列リテラル
```

```
%Y: 2025 (年、4桁)
is a: 文字列リテラル
%A: Sunday (曜日のフルネーム)
on the: 文字列リテラル
%d: 01 (日)
day of: 文字列リテラル
%B: June (月のフルネーム)
.: 文字列リテラル
```

UTC と JST を扱う

```
import datetime

# 以下の3つが必要
# 公式テキストには、無かったので、詳細は調べませんでした
# 雰囲気だけ直前に見ればOKだと思う
# datetime.datetime
# datetime.timezone
# datetime.timedelta

# UTC（世界中の時計の基準）
utc_dt = datetime.datetime.now(datetime.timezone.utc)
print(f"UTC: {utc_dt}")  # 今のUTCを表示する

# JST (UTC+9)：（UTC + 9時間：日本時間）
jst_tz = datetime.timezone(datetime.timedelta(hours=9))
jst_dt = datetime.datetime.now(jst_tz)
print(f"JST: {jst_dt}")  # 今の日本時間を表示する
```

アーカイブ　データ圧縮

- アーカイブ
  - 複数のファイルを一つにまとめること
- 圧縮
  - ファイルのサイズを小さくすること
- アーカイブして圧縮
  - 複数のファイルを一つにまとめてから、サイズを小さくする
- 公式テキスト(pythonチュートリアル第4版)に表示されている、モジュール名：
  - zlib、gzip、bz2、lzma、zipfile、tarfile

```
# zlibモジュール

import zlib
```

```
original = b"Hello, World!"  # バイト列で扱う（b"～"）
print(type(original))  # <class 'bytes'>
length_original = len(original)
print(length_original)  # 13

processed = zlib.compress(original)  # processed: 処理が行われた
print(type(processed))  # <class 'bytes'>
length_processed = len(processed)
print(length_processed)  # 21

"""
上記の例では、サイズが膨らんでいる：
これは、必要な付加情報のためである。
たとえれば、
「扱う量(売上)が少なくて、経費を回収することが出来ない」イメージ
ですね。

"""

# もっと長い文字列で試す
longer = (
    b"""abcdefghijklmnopqrstuvwxyz0123456789
    abcdefghijklmnopqrstuvwxyz0123456789
    abcdefghijklmnopqrstuvwxyz0123456789
    abcdefghijklmnopqrstuvwxyz0123456789
    """)
length_longer = len(longer)
print(length_longer)  # 164

processed_longer = zlib.compress(longer)
length_processed_longer = len(processed_longer)
print(length_processed_longer)  # 52
```

パフォーマンス計測（performance：性能、業績、実績）

- パフォーマンス計測のためのモジュール
    - timeit
        - 小さめのコードが適している
            - 特定の処理の速さを測るのに適している
    - profile、pstats
        - 大きめのコードが適している
            - プログラムの塊の中で、どこが遅いのかを、見つけるのに適している
- timeitモジュールのポイント
    - timeit.Timerクラス（クラスは大文字にする慣習）

        - コンストラクタ
            - Timer(stmt(statement([コンピュータ]命令文), setup(定義・段取り))
        - 下のコードの例
            - "temp = x; x = y; y = temp", "x=10; y=20"
                - `;`で、1行で書くことが出来る文法がある
                    - 文字列形式になっているが、区切りを分かりやすくするためだと私は思う
    - timeit.Timerオブジェクト.timeit(number=1_000_000)
        - number：実行する回数：1000000はデフォルトの回数（`_`は数値に使える）

- 下のコードの説明：
    - x と y の値を交換するコード
        - 上は、temp変数を介して、交換する
        - 下は、タプルの機能を使って、簡単に交換している
    - 結果を見れば、タプルの機能を使って、簡単に書いた方がパフォーマンスが良いことが分かる
        - result2と、result3で、多少違うのは、PCの環境などが影響しているからである

```
import timeit

step1 = timeit.Timer("temp = x; x = y; y = temp", "x=10; y=20")
result = step1.timeit(number=1000000)  # 0.016584599972702563
print(result)

# a, b = b, a  ➡  値の交換が出来る（タプルの機能を使っている）
step2 = timeit.Timer("x, y = y, x", "x=10; y=20")
result2 = step2.timeit(number=1000000)
print(result2)  # 0.012994899996556342
result3 = step2.timeit()  # デフォルトの回数（1000000：上のコードと同じ）
print(result3)  # 0.01297879999037832
```

品質管理

-

【docstring の具体的な書き方】

docstringとは、関数やクラスの定義の後には、直後の行に、インデントを1回分行い、
モジュールの場合は、インデント無しで書く

""" と """ の間に書く。
書式は、以下のようである

---書き方---
>>> コード # IDLEのように書く（>>>の後に半角空ける）
結果 # 期待される出力
---書き方、ここまで

doctestモジュール

- docstringに埋め込まれたテストを検証するツールを提供する
    - 以下に、普通に書いた時と、doctestモジュールを使った場合を示します
    - docstringとは、関数やクラスの定義の直後の行に、インデントを1回分行い、
      `"""`と`"""`の間に書けば問題ありません。
      下のコード(doctestモジュールを使った例)を参考にしてください。

普通のコード

```
# 型ヒント付き
#   `: list[int]`：numbersは、整数のリストを期待（強制はできない）
#   ` -> int`：整数値を返すことを期待（強制はできない）
def sum_positive_even_numbers(numbers: list[int]) -> int:
    """
    数値のリストを受け取り、その中の正の偶数のみを合計して返します。
    （このdocstringは関数の説明として残りますが、テストコードは含みません）
    """
    total = 0
    for num in numbers:
        if num > 0 and num % 2 == 0:
            total += num
    return total


# このファイルを直接実行した時だけ、実行される書き方
if __name__ == '__main__':
    # 関数を実際に使ってみる例
    my_list1 = [1, 2, 3, 4, 5, 6]

    # 出力: [1, 2, 3, 4, 5, 6] の中の正の偶数の合計: 12
    result1 = sum_positive_even_numbers(my_list1)
    print(
        f"{my_list1} の中の正の偶数の合計: {result1}")

    # 出力: [10, 21, 30, 43, 50] の中の正の偶数の合計: 90
    my_list2 = [10, 21, 30, 43, 50]
    result2 = sum_positive_even_numbers(my_list2)
    print(
        f"{my_list2} の中の正の偶数の合計: {result2}")

    # 出力: [] の中の正の偶数の合計: 0
    empty_list = []
    result_empty = sum_positive_even_numbers(empty_list)
    print(
        f"{empty_list} の中の正の偶数の合計: {result_empty}")

"""実行結果：
[1, 2, 3, 4, 5, 6] の中の正の偶数の合計: 12
[10, 21, 30, 43, 50] の中の正の偶数の合計: 90
[] の中の正の偶数の合計: 0
"""
```

doctestモジュールを使ったコード

```
import doctest


# 型ヒント付き
#   `: list[int]`：numbersは、整数のリストを期待（強制はできない）
```

```
#   ` -> int`：整数値を返すことを期待（強制はできない）
def sum_positive_even_numbers(numbers: list[int]) -> int:
    """
    数値のリストを受け取り、その中の正の偶数のみを合計して返します。

    >>> sum_positive_even_numbers([1, 2, 3, 4, 5, 6])
    12
    >>> sum_positive_even_numbers([10, 21, 30, 43, 50])
    90
    >>> # 奇数のみのリストの場合
    >>> sum_positive_even_numbers([1, 3, 5, 7])
    0
    >>> # 空のリストの場合
    >>> sum_positive_even_numbers([])
    0
    >>> # 負の数や0を含む場合（0は偶数ですが正ではないので対象外）
    >>> sum_positive_even_numbers([-2, 0, 2, 4, -5, 6])
    12
    >>> # 全て負の数の場合
    >>> sum_positive_even_numbers([-2, -4, -6])
    0
    """
    total = 0
    for num in numbers:
        if num > 0 and num % 2 == 0:
            total += num
    return total


# このファイルを直接実行した時だけ、実行される書き方
if __name__ == '__main__':
    # doctestを実行します。
    # verbose=True を指定すると、成功したテストも含め、
    # 各テストケースの詳細な実行結果が表示されます。
    # 試験対策の学習としては、詳細を確認できる verbose=True がおすすめです。
    doctest.testmod(verbose=True)                    ↑ ： PowerShell から実行した時には、詳細も、出力されました。

"""実行結果：
6 tests passed
"""
```

unittest

- ポイント
  - テストクラスの継承 (unittest.TestCase)
  - テストメソッドの命名規則 (test_)
  - アサーションメソッドの使用 (self.assertEqual, self.assertRaises)
  - テストの実行 (unittest.main())

テストされるコード

```
# my_module.py
def myfunc(x: int, y: int) -> int:  # 型ヒントには強制力はない

    # これは、作者の確認のためなので、型ヒントには含めないことする
    # ちなみにprint関数が返すのは、Noneです
    print("私はmy_moduleのmy_func()関数です")

    # これは、例外が発生した場合なので、
    # TypeErrorも型ヒントには含めないことにする
    if type(x) != int or type(y) != int:
        raise TypeError
    else:
        return x + y
```

テストするコード

```
# u_test_1_copy2.py

import unittest  # unittestモジュールをインポートする
from my_module import myfunc  # テストしたい関数をインポートした

# unittest.TestCaseを、継承することは必須である
# クラス名に関しては、Test～とするのは、慣習であり、必須ではない
#   例えば、`TestFunc`としなくてもよい
class CheckFunc(unittest.TestCase):
    # 関数名の先頭に、`test_`を付けるのは必須である
    def test_myfunc(self):
        # self.assertEqual(関数, 期待される値):
        self.assertEqual(myfunc(3, 4), 7)  # テスト1
        self.assertEqual(myfunc(-1, 4), 3)  # テスト2

        # withを付ける理由の一つは、より簡単に書けることです
        # with self.assertRaises(例外クラス):
        # 2つのテストをする場合、2つ書く必要がある
        with self.assertRaises(TypeError):
            myfunc("a", 1)  # テスト３
        with self.assertRaises(TypeError):
            myfunc(1, True)  # テスト４


# このコードは、必須である
# これで、テストが実行できるようになっている
# （というより、トリガー的だとAIは言っていました）
# 名前が`test_start()`などだと覚えやすいがこのコードの内部はやや複雑なので、
# その中で、main()と言う名前になったのだと思われます
# 私たちユーザーにとって大事なのは、慣れることでしょう
```

```
# もっとも内部の仕組みを完全に理解すればそれに越したことはないと思いますが、
# 最初にやるべきことでもないと思います
unittest.main()

"""実行結果（最後のメッセージが、`OK`なので、テストは正しく実行されました）
～unit_test> py u_test_1_copy2.py
私はmy_moduleのmy_func()関数です
私はmy_moduleのmy_func()関数です
私はmy_moduleのmy_func()関数です
私はmy_moduleのmy_func()関数です
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
"""
```

Pythonには電池が付いている

- ポイント
  - 結局「少ない手間で、目的を達成できる」ことが言いたい
    - Pythonの作者的には、　＜作者：Guido van Rossum（グイド・ヴァン・ロッサム）＞
      - 簡単に実行できることを、体に叩き込んで覚える人には、落とし穴も、問題にならない
      - 細かいことが、気になる人には、嫌われるかもしれない
        - 公式本(Pythonチュートリアル第4版)で、分かりずらい所を説明してくれている
          - 付録E 初心者だった頃、みんながひっかかる Pythonのヘンなとこ
    - 恐れながら私の意見
      - 最初は慣れが大事で、とにかくルールとして覚える
      - 少しずつドキュメントなどが理解できるように頑張る
        - それは、理屈が分かると覚えるのがずいぶん楽になるから

出力整形

str関数　repr関数　reprlibモジュール　pprintモジュール

- str関数
  - ビルトイン関数
  - 人にやさしい文字列を返す
  - ただし、eval関数での復元は通常困難（有効なPython式にならないため）
- repr関数
  - ビルトイン関数
  - 人には優しくない

    - 可能な限りPythonコードで実行可能な形式で出力する
    - eval関数で復元できることを目指すが、常に可能とは限らない
- reprlibモジュール
    - 目的：データを見やすくする場面でこれが適している場合に使用する
    - repr機能を拡張したモジュール
    - 標準ライブラリのモジュールである（`import reprlib`）
    - 大量のデータがある辞書やリストをざっと確認したい時
    - デバッグ時に巨大なオブジェクトの概要だけ知りたい時
- pprintモジュール
    - 目的：データを見やすくする場面でこれが適している場合に使用する
    - JSONデータやAPI応答の構造を確認したい時
    - 複雑な設定ファイルの内容を読みやすく表示したい時
    - デバッグ時に入れ子になったデータ構造を理解したい時
    - `width=80`がデフォルト：1行の最大文字数を制限できる
    - 出力結果はPythonの有効な構文を維持しているため、そのままコードとして使用可能

```
# IDLEを使っています

# repr関数
import datetime
crnt = datetime.datetime.now()
repr(crnt)  # repr関数
'datetime.datetime(2025, 5, 27, 10, 48, 18, 71414)'
eval(repr(crnt))  # 復元できた
datetime.datetime(2025, 5, 27, 10, 48, 18, 71414)

str(crnt)  # str関数
'2025-05-27 10:48:18.071414'
eval(str(crnt))  # SyntaxError
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    eval(str(crnt))  # 復元できない
  File "<string>", line 1
    2025-05-27 10:48:18.071414
         ^
SyntaxError: leading zeros in decimal integer literals are not permitted;
 use an 0o prefix for octal integers
```

```
# IDLEを使っています

# replibモジュール
# 文字列(イテラブルオブジェクト)が、セットで返された結果を、どんな感じになるのかを見せている
# `...`は、まだデータがあることを示している
import reprlib
```

```
reprlib.repr(set("abcdabcdefghefghhijkhijk12341234"))
"{'1', '2', '3', '4', 'a', 'b', ...}"
```

```
# IDLEを使っています

# pprintモジュール
# 結果をコードとして使えると、エディタ上で読みやすい
# また、構造の理解を助ける
import pprint
lst = [[[1,2,3], 4,5, [[6], [7,8],9]]]

pprint.pprint(lst)  # width=80 がデフォルト
[[[1, 2, 3], 4, 5, [[6], [7, 8], 9]]]

pprint.pprint(lst, width=30)
[[[1, 2, 3],
  4,
  5,
  [[6], [7, 8], 9]]]

pprint.pprint(lst, width=10)
[[[1,
   2,
   3],
  4,
  5,
  [[6],
   [7, 8],
   9]]]
```

textwrapモジュールのfill()メソッド

- 第1引数：文字列
- 第2引数：width=70がデフォルトのキーワード引数
- 文字列を、(デフォルトで)70文字以内で納めて改行し、かつワードラップも行う
    - ワードラップ機能：単語の途中では改行しない

```
# textwrap

import textwrap

# `\`は、PDFに収まらないので、改行のために使っています。
# 本来は、1行の文字列です
# PDFにする目的がなければ、長い文字列のままでも一気に読みやすくなる
doc = """Lorem Ipsum is simply dummy text of the printing and typesetting \
industry. Lorem Ipsum has been the industry's standard dummy text ever \
```

```
since the 1500s, when an unknown printer took a galley of type and \
scrambled it to make a type specimen book."""

print(textwrap.fill(doc, width=50))

"""実行結果
Lorem Ipsum is simply dummy text of the printing
and typesetting industry. Lorem Ipsum has been the
industry's standard dummy text ever since the
1500s, when an unknown printer took a galley of
type and scrambled it to make a type specimen
book.
"""
```

localeモジュール

locale.setlocale()メソッド

- 目的：プログラムのロケールを設定すること
  - ロケール：言語別または地域別に分けられた設定情報
- 第1引数
  - locale.LC_ALL
  - これは、通貨、日付、数値の書式など、
    ロケール（地域や言語に依存する設定）に関する**全ての設定を一度に**行うための定数です。
  - 実際には数値なので、引用符は付けない
- 第2引数
  - 地域と言語、文字コードを指定する文字列
  - 例：
  - "ja_JP.UTF-8" (日本、日本語、UTF-8エンコーディング)
  - "English_United States.1252" (アメリカ、英語、Windows-1252エンコーディング)

試験対策のポイント

- `locale.setlocale()` で事前にロケールを設定する必要があること。
- `format_string()` が、その設定されたロケールに基づいて文字列を整形すること。
- 主要な引数である「書式文字列」「整形する値」「（桁区切り）」の役割を理解していれば、
  基礎レベルとしては十分対応できる可能性が高いです。

(再まとめ：第4引数は念のため)

- locale.format_string(★)の説明：
  - ★の部分（大雑把な説明：このレベルで試験にはOKだと思われる）
    - 第1引数：書式の設定の文字列

        - 小数点以下の桁など
    - 第2引数：その書式設定に従ってフォーマットされる値
        - 数値や文字列、タプルも使えます
    - 第3引数：grouping(キーワード引数)、省略でFalseのデフォルト
        - 数値に桁区切りを適用する
    - 第4引数：monetary(キーワード引数)、省略でFalseのデフォルト
        - 通貨に関係する設定
        - キーワード引数は、キーワードを使わない時は、順序を守ること

```
# ！ ファイル名とimportするモジュール名を同じにすると、Errorが出る！
#    それは、初めて知ったが、覚えておくべきこと！（これは内容と関係ありません）

# locale.setlocale()

# localeモジュールのインポート
import locale

# 日本のロケールを設定する場合
# Shift_Jisの場合：'Japanese_Japan.932'
# locale.setlocale(locale.LC_ALL, 'Japanese_Japan.932')
# utf-8に設定
locale.setlocale(locale.LC_ALL, 'ja_JP.UTF-8')

# 数値を地域の慣習に合わせて表示（ここでは、日本に設定されている）
price = 1500
formatted_price = locale.format_string("%s%.*f",
                                       (locale.localeconv()[
                                            'currency_symbol'],
                                        locale.localeconv()['frac_digits'],
                                        price),
                                       grouping=True)
print(formatted_price)  # 日本の設定なら '¥1,500'

# アメリカのロケールを設定する場合
# 「.1252」は、エンコーディングしてい部分
# locale.setlocale(locale.LC_ALL, 'English_United States.1252')
# utf-8に設定
locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')

# 数値を地域の慣習に合わせて表示
price = 1000
formatted_price = locale.format_string("%s%.*f",
                                       (locale.localeconv()[
                                            'currency_symbol'],
                                        locale.localeconv()['frac_digits'],
                                        price),
                                       grouping=True)
print(formatted_price)  # アメリカ設定なら '$1,000.00'
```

END