


The background is a vibrant, abstract composition of light trails in shades of blue, yellow, and orange, creating a sense of motion and energy. A large, white, rounded rectangular box is centered over this background, serving as a container for the title text.

Python その2

この資料について



オンライン学習プラットフォーム『Udemy』
で公開している、
『Python x FastAPI初心者向け講座』の
説明用資料です

[https://www.udemy.com/course/
python_fastapi](https://www.udemy.com/course/python_fastapi)



モジュール

モジュール



関数、変数、クラスなどを含むPythonファイル
共通する処理をファイルにまとめて、
他のファイルからimportして使う事ができる

`import` ファイル名

ファイル名.関数名

モジュール作成・インポート

section3/module_sample.py #別ファイルで関数や変数を作る

```
def add(x, y):  
    return x + y
```

```
def multiple(x, y):  
    return x * y
```

```
sample_value = "モジュールのテスト"
```

section3/module_import.py

```
import module_sample # ファイル名を指定してインポート(拡張子は不要)
```

```
add_result = module_sample.add(3, 5) # ファイル名.関数名
```

```
multiple_result = module_sample.multiple(3, 5)
```

```
print(add_result, multiple_result, module_sample.sample_value) # 8, 15, モジュールのテスト
```

モジュール from句 関数や変数を指定できる

使う機能を限定できる

section3/module_from.py

from module_sample **import** add # 関数名を指定

from module_sample **import** sample_value # 変数名を指定

from module_sample import * # ファイル内全てインポート

add_result = add(3, 5)

multiple_result = multiple(3, 5) # これはエラー

print(add_result, sample_value)

モジュール asで名称変更

section3/module_as.py

```
import module_sample as sample #別名指定  
# from module_sample import add as sample_add  
  
add_result = sample.add(3, 5) # ファイル名.関数名  
multiple_result = sample.multiple(3, 5)  
print(add_result, multiple_result, sample_value)  
# 8, 15, モジュールのテスト
```




標準モジュール

標準モジュール (抜粋)

Pythonに含まれるモジュール 追加インストール不要

モジュール名	できること	メソッド(関数)
os	OSとのやりとり、ファイルフォルダの操作、環境変数の取得など	getcwd(), environ, listdir() path.join(), remove(), mkdir()
datetime	日付、時刻の操作	now(), strftime()
math	数学関数へのアクセス	sqrt(), pi, sin(), radians(), degrees()
sys	Pythonインタプリタとのやりとり システム固有のパラメータへのアクセス	argv, exit(), path, platform, version
json	JSONデータのエンコード、デコード	dumps(), loads(), load()
urllib	URLを扱ういくつかのモジュール	request.urlopen(), parse.urlencode() parse.quote()
random	乱数の生成	randint(), choice(), shuffle() uniform()
re	正規表現	search(), findall(), sub(), compile()

標準モジュール 例

section3/standard_module.py

```
import os
```

```
from datetime import datetime
```

```
import math
```

```
import random
```

```
current_directory = os.getcwd() # 現在のフォルダ
```

```
print(current_directory)
```

```
print(datetime.now()) # 現在の時刻
```

```
print(math.pi) # 円周率
```

```
print(random.randint(0, 9)) # ランダムな整数
```




パッケージ

複数のモジュールをまとめたフォルダ

フォルダ内に空のファイル

`__init__.py` を作成する事で

パッケージ化できる

パッケージ 例1



mypackage/__init__.py 空のファイル

mypackage/module1.py

```
def hello(name):  
    print(f"Hello, {name}!")
```

mypackage/module2.py

```
def good_bye(name):  
    print(f"Goodbye, {name}!")
```


パッケージ 例2



`package_import.py`

`from パッケージ名 import モジュール名`

`from mypackage import module1, module2`

`module1.hello("Tanaka")`

`module2.good_bye("Yamada")`



pip

pip

Pythonのパッケージ管理ツール

用途	コマンド	例
パッケージのインストール	pip install xxx	pip install requests pip install requests==2.25.1 pip install -r requirements.txt
アンインストール	pip uninstall xxx	pip uninstall requests
アップグレード	pip install --upgrade xxx	pip install --upgrade requests pip install --upgrade pip
リスト表示	pip list	
特定パッケージの情報表示	pip show xxx	pip show requests
現環境のファイル書き出し	pip freeze > requirements.txt	
一括インストール	pip install -r requirements.txt	



外部パッケージ

外部パッケージ (抜粋)

追加インストール必要

パッケージ名	できること	メソッド(関数)
requests	HTTPリクエストの送受信	get(), post(), put(), delete()
numpy	数値計算・配列操作	array(), arange(), reshape()
pandas	データ分析と操作	DataFrame(), Series(), read_csv()
matplotlib	データの可視化(グラフ作成)	plot(), show(), scatter()
scipy	科学技術計算	integrate.quad(), optimize.minimize()
pytest	テストコードの作成・実行	assert, fixture(), mark.parametrize()
Beautiful Soup	HTMLとXMLの解析	BeautifulSoup(), find(), find_all()
fastapi	Webアプリ構築	FastAPI(), get(), post(), run()

外部パッケージ 例



```
pip install numpy # インストール
```

```
section3/numpy_test.py
```

```
import numpy as np
```

```
# 2次元配列（行列）の作成
```

```
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
```

```
print(arr_2d)
```




クラス

クラス

Pythonはオブジェクト指向言語 (オブジェクトは物という意味)

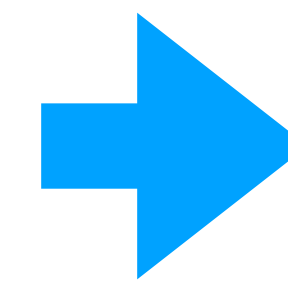
似たような機能、役割をまとめる

(他言語 Java, C#, PHP, Ruby, ...)

プログラミングで表現するにはクラスを使う

変数/定数
関数

変数/定数
関数



属性
メソッド

名称が変わる

クラス



パッケージやモジュールの中に
含める事もできる

パッケージ > モジュール > クラス > 変数、
関数

クラス独自機能



クラスには複数の独自機能がある

インスタンス(実体化)

コンストラクタ `__init__`

クラス変数とインスタンス変数

インスタンスメソッド/静的メソッド/クラスメソッド

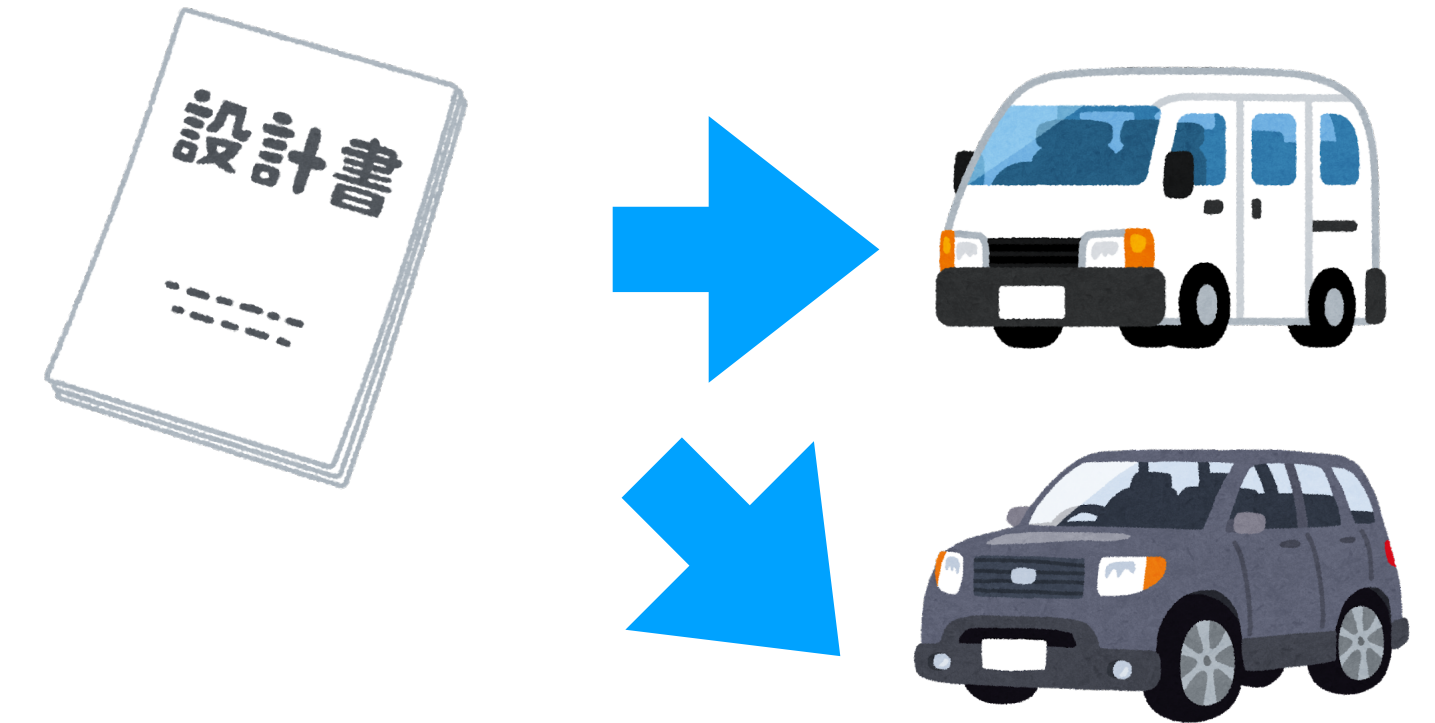
専用デコレータ (`@property`, `@classmethod`, `@staticmethod`)

継承

インスタンス

クラス(設計図)

-> 実際に使える状態(インスタンス(実例、実態))



定義

```
class TestClass:
```

```
    def メソッド名(self, 引数, ...):  
        # 処理
```

クラス名は大文字スタートのパスカルケース
selfはクラス自身を示す

使う時(インスタンス化)

```
変数名 = クラス名()
```

```
変数名.メソッド名()
```


インスタンス 例



`section4/instance.py`

```
class SimpleClass:
```

```
    def hello(self):
```


```
        print("hello")
```

```
simple = SimpleClass() # インスタンス化
```

```
print(type(simple)) # <class '__main__.SimpleClass'>
```

```
simple.hello() # メソッドを実行
```


コンストラクタ



クラスの初期化(イニシャライザ)

クラスがインスタンス化される時に1度実行される
属性の指定によく使われる

```
def __init__(self, 引数1, ...):  
    self.xxx = 引数1
```

(前後にアンダーバー2つのメソッドは
マジックメソッドやデューンダーメソッドと呼ばれる)

コンストラクタ 例

section4/init.py

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
    def greet(self):
```

```
        print(f"名前は{self.name}。年齢は{self.age}です。")
```

```
person1 = Person("三苦", 25) # インスタンス化実行時に変数に代入
```

```
person2 = Person("道安", 28)
```

```
person1.greet()
```

```
person2.greet()
```


クラス変数とインスタンス変数

クラス変数・・・インスタンス全てで共通

height = 10 (selfを使わず直接書く 定数など)

インスタンス変数(属性)・・・インスタンス毎に保持
(コンストラクタ内で定義する)

```
def __init__(self, name, age):  
    self.name = name  
    self.age = age
```


クラス変数

section4/init.py

```
class Person:
    height = 10 # 追加
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"名前は{self.name}。年齢は{self.age}です。")

person1 = Person("三苦", 25) # インスタンス化実行時に変数に代入
person2 = Person("道安", 28)

person1.height # どのインスタンスでも同じ値になる
person2.height #
```


静的(スタティック)メソッド

インスタンス化しないでメソッドを使える方法
(コンストラクタは実行されない)

デコレータを使う(関数やメソッドの装飾)

@staticmethod

selfは不要



静的(スタティック)メソッド 例

section4/static.py

```
class MathClass:
```

```
    @staticmethod
```

```
    def add(a, b):
```

```
        return a + b
```

```
    @staticmethod
```

```
    def multiple(a, b):
```

```
        return a * b
```

```
result_add = MathClass.add(5, 3) # クラス名.メソッド名 を直接指定
```

```
result_multiple = MathClass.multiple(5, 3)
```

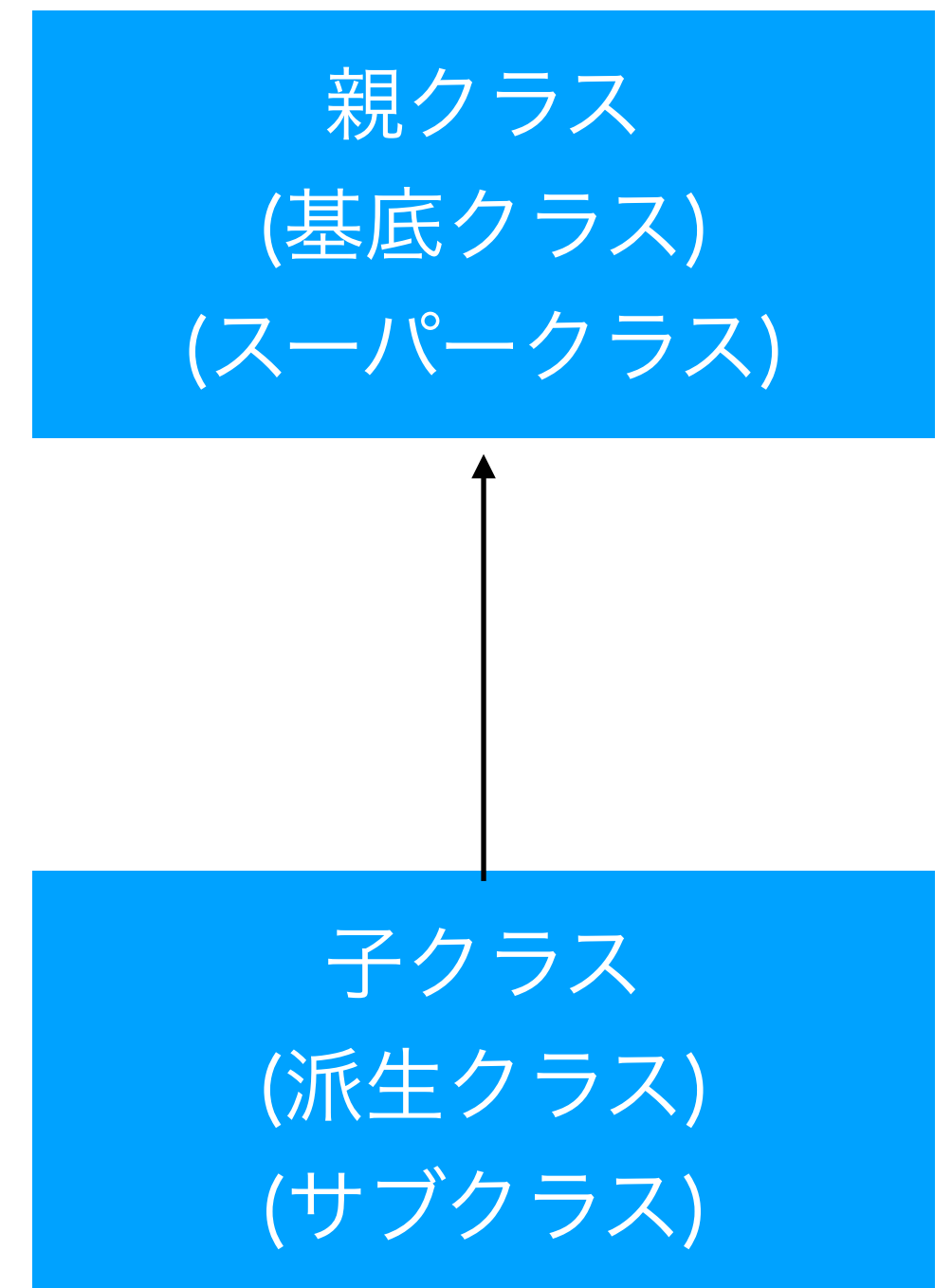
```
print(result_add, result_multiply) # 8 15
```


継承

親クラスの情報(属性・メソッドなど)を受け継いで使える

クラスが増えた時に、
共通した処理を親クラスにまとめて、
子クラスには必要な機能だけ追加する

class クラス名(親クラス名) :



継承 例

section4/extend.py

```
class Vehicle: # 親クラス
```

```
    def __init__(self, name, speed):
```

```
        self.name = name
```

```
        self.speed = speed
```

```
    def get_information(self):
```

```
        return f"{self.name}はスピード{self.speed}km/hです。"
```

```
class Bicycle(Vehicle): # 子クラス1
```

```
    pass
```

```
class Car(Vehicle): # 子クラス2
```

```
    def get_information(self): # 親のメソッドと同名なので上書きする(オーバーライド)
```

```
        return f"{self.name}の車はスピード{self.speed}km/hです。"
```

```
    def original(self):
```

```
        return "子のメソッド"
```


継承 実行例



```
vehicle = Vehicle("テスラ", 30) # 親クラス  
print(vehicle.get_information())
```

```
bicycle = Bicycle("自転車", 10) # 子クラス1  
print(bicycle.get_information()) # 親のメソッド
```

```
car = Car("車", 50) # 子クラス2  
print(car.get_information()) # オーバーライド  
print(car.original()) # 子のメソッド
```




デコレータ

デコレータ

装飾者・インテリアデザイナー

関数やメソッドの前後に追加の処理を加える

(ログ出力、実行時間の計測、アクセス制御、キャッシュ、引数チェック etc...)

```
def デコレータ名(func): #引数に関数をとる
    def wrapper():
        # 処理
        func() # 引数にとった関数の実行
    return wrapper
```

@デコレータ名 #使う時は @デコレータ名

```
def 関数名():
```

デコレータ 例



section4/decorator.py

```
def simple_decorator(func):  
    def wrapper():  
        print("デコレータ内の関数")  
        func()  
    return wrapper
```

@simple_decorator

```
def say_hello():  
    print("Hello!")
```

say_hello()



引数の デフォルト値

引数のデフォルト値



関数(メソッド)に

あらかじめデフォルトの引数を設定する仕組み

(デフォルト値・・初期設定値)

引数が渡ってくればその引数を使う

引数がないければデフォルト値を使う

```
def 関数名(引数名=デフォルト値)
```


引数のデフォルト値

section5/default.py

関数定義、引数にデフォルト値を設定

```
def greet(name="World"):
    print(f"Hello, {name}!")
```

引数を指定

```
greet("John") # Hello, John!
```

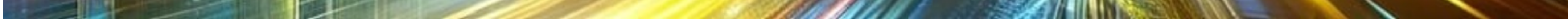
引数なし デフォルト値で実行

```
greet() # Hello, World!
```




位置引数と キーワード引数

位置引数とキーワード引数



位置引数: 関数に引数を順番に渡す方法

キーワード引数: 引数に対応するパラメータ
名を指定 (順番が違ってても処理できる)

キーワード引数なら順番に影響しない

section5/kw_args.py

```
def introduce(name, age):  
    print(f"私は{name}です、年齢は{age}です")
```

順番を間違う

```
introduce(30, "山田")
```

キーワード引数を使って関数を呼び出す

```
introduce(age=20, name="田中")
```

引数内のコードが少し増えるので、必要に応じて



可變長引數

可変長引数



関数で扱う引数の数が可変の場合

argumentsの略(引数)

*args タプルとして扱われる

**kwargs 辞書として扱われる

可変長引数 例

section5/variable_args.py

```
def print_args(*args):
```

```
    for arg in args:
```

```
        print(arg)
```

print_args('apple', 'banana', 'cherry') # 引数を追加しても処理される

```
def print_kwargs(**kwargs):
```

```
    for key, value in kwargs.items(): # 辞書型.items()
```

```
        print(f"{key}: {value}")
```

print_kwargs(apple='green', banana='yellow', cherry='red')



```
if __name__ ==  
    "__main__"
```


if __name__ == "__main__":



__name__ ・ ・ Pythonスクリプト実行時に
自動的に設定されるグローバル変数

__main__ ・ ・ Pythonスクリプト実行時に
自動的に設定される値

Pythonスクリプト実行かどうか、
という判定に使える

__name__ と __main__

section5/name_main.py

```
def main_function():
```

```
    print("メイン関数")
```

```
def utility_function():
```

```
    print("ユーティリティ関数")
```

```
# スクリプト実行していたら実施
```

```
if __name__ == "__main__":
```

```
    print(__name__) # __main__ が設定されている
```

```
    main_function()
```

```
$ python name_main.py
```


別モジュールから読み込む



```
section5/import_main.py
```

```
import name_main
```

```
name_main.utility_function()
```

```
# name_main.main_function() 関数指定すれば使える
```

```
$ python import_main.py # mainは実行されない
```




例外処理

例外処理

エラーが出そうな時にあらかじめ仕込んでおく

1. 外部からの入力进行处理する場合(ユーザー入力、ファイル読込)
 2. 外部接続 (データベース接続、外部API接続 etc...)
 3. 計算エラー
- etc...

try: # try以下の処理でエラーが発生する場合を想定して 処理を書いておく

 #処理

except 例外:

 # 例外発生時の処理

finally:

 # 例外の種類に関わらず実行される処理

例外処理 例



section5/try_except.py

```
def divide(a, b):
```

```
    try:
```

```
        result = a / b
```

```
    except Exception as e: # Exceptionは全ての例外を捕捉(原因不明の時に使う)
```

```
        print(f"エラー発生: {e}")
```

```
    finally:
```

```
        print("処理終了")
```

divide(10, 0) # 例外発生 エラー発生: division by zero

divide(10, 2) # 例外発生しない

例外の種類



AttributeError(属性エラー)、TypeError(型エラー)、
FileNotFoundError(ファイルがないエラー)、
PermissionError(権限エラー)、TimeoutError(タイムアウト)
など様々

独自の例外を定義し、発生させる事ができる
(Exceptionクラスを継承して作成)

raise 独自例外



API

API



Application Programming Interfaceの略
外部から情報にアクセスできる仕組み

楽天市場API、じゃらんAPI、Amazon
API, X(Twitter) API, 天気予報 API, 郵便番
号API、Qiita API, GitHub API etc...

HTTPリクエストとレスポンス

HTTPリクエスト


- HTTPリクエスト行 (メソッド)
- HTTPヘッダー
- データ本体



HTTPレスポンス

- レスポンス状態行 (状態コード)
- HTTPヘッダー
- データ本体

HTTPメソッド



get ・ ・ URLに表示される(検索条件など)

クエリースtring(パラメータ)をつかって情報送信

?以降 key=value形式

<https://www.google.com/search?>

[q=python&oq=python&sourceid=chrome&ie=UTF-8](https://www.google.com/search?q=python&oq=python&sourceid=chrome&ie=UTF-8)

post ・ ・ URLに表示されない

リクエストボディを使って情報送信

他に、put, patch, deleteなどがある

JSON



JavaScript Object Notation

XMLに比べ軽量 シンプルでわかりやすい

(キーバリュー形式 Pythonだと辞書型に近い)

```
{ "key1" : "value1" }
```

```
{ "key1" : [  
  {"key2" : "value2" },  
  {"key3" : "value3" },  
]}
```


エンコードとデコード

エンコード

`json.dumps()`

`requests post`送信時に変換可能



デコード

`json.loads()`

`res.json()` ・ ・ 内部的に`json.loads()`を呼び出す

シリアライズ
ともいう

API通信をやってみる

```
$ pip install requests # 外部ライブラリ
```

```
section5/api_test.py
```

```
import requests # インポート
```

```
url = "https://dog.ceo/api/breeds/image/random" # URL指定
```

```
# get通信し、レスポンスを変数に代入
```

```
res = requests.get(url)
```

```
print(res.status_code) # 200 なら OK
```

```
# 取得したJSONを辞書オブジェクトに変換して表示
```

```
print(res.json())
```

```
print(res.json()["message"])
```




型ヒント

型ヒント

Pythonは動的型付け言語（自動で型が設定される）

- ・ ・ どのデータ型を扱っているかわからない場合がある

関数の引数、戻り値に期待されるデータ型を指定

バグ早期発見、ドキュメント化する役割

(FastAPIではバリデーションにも使われる)

def 関数名(引数名: 型) -> 戻り値の型:

コレクション型(リスト、辞書、タプルなど)はインポートする場合もある

型ヒント 例

section5/type_hint.py

from typing import List # リストや辞書などはインポートする場合もある

```
def add_numbers(a: int, b: int) -> int:  
    return a + b
```

```
def greet(name: str) -> str:  
    return f"Hello, {name}!"
```

```
def list_sum(numbers: List[int]) -> int: # リスト内の型も指定できる  
    return sum(numbers)
```

```
result = add_numbers(5, 3)  
print(result) # 8  
print(greet("Tanaka")) # Hello, tanaka!  
print(list_sum([1,2,3])) # 6
```

型ヒント 種類

型	意味
Union	複数の型のいずれか Python10.0以降はパイプも使える ex) a b
Annotated	型ヒントに追加情報を提供 Python3.9以降 ex) Annotated[User, Depends(get_current_user)]
オブジェクト名(クラス名)	そのオブジェクト(クラス)を型指定
その他	datetime, dict, timedelta,



非同期处理

非同期処理



多数のユーザーからアクセスがあり、
APIサーバー通信やデータベース接続など
外部との通信が必要な場合

同期処理・・・多数の同時接続などがあると処理が重くなりやすい

非同期処理(asynchronous)・・・
大量の同時接続やリクエストを効率的に処理できる
(コードが複雑になりがち)

Pythonでは `asyncio` 標準ライブラリ を使う事で実現

非同期処理 例

section5/async.py

```
import asyncio
```

```
# 関数定義の前に asyncをつける
```

```
async def async_function():
```

```
    print('Hello')
```

```
    await asyncio.sleep(2) # 処理の完了を待つ
```

```
    print('World')
```

```
# 非同期関数の実行
```

```
asyncio.run(async_function())
```