Сопоставление Шаблону

Сопоставление шаблону уже упоминалось у нас в конспекте об условных выражениях Как я говорил, это, на самом деле, отдельная система конструкций языка со своей логикой интерпретирования выражений. Будем разбираться

В ***match*** мы получаем объект, а в ***case*** явно описываем вид объекта, который представлен в match. Срабатывает первое *почти наверное точное* (об этом после примеров) совпадение

Например, в примерах ниже мы подаём на ввод строку, она методом ***split*** разбивается по словам в список. И совпадение со списком мы и проверяем

```
>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...
...
Help

>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...
...
help
HELP

>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...
...
help me
Sure
```

Любой объект в ***case*** (как составную часть всего объекта, так и сам объект) можно проименовать. Отсюда очень просто выводится логика обработки поумолчанию: обозначаем весь объект переменной (принято использовать имя переменной _), поймается вообще всё, что угодно. Собственно, этим и объясняется почти наверное точное совпадение: некоторые части могут быть описаны с помощью переменных

```
>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...     case _:
...         print('WAT?')
...
qwdfgyujkjnbfd
WAT?
```

А вот шаблоны с переменными

```
>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...     case['help', command]:
...         print(f'The {command} is important!')
...     case _:
...         print('WAT?')
...
help Bulbazavr
The Bulbazavr is important!

>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...     case['help', command]:
...         print(f'The {command} is important!')
...     case['help', *cmds]:
...         print("comands are", cmds)
...     case[]:
...         print('What do u want?')
...     case _:
...         print('WAT?')
...
help a s dfg
comands are ['a', 's', 'dfg']
```

Из приятного, эти связанные переменные мы тоже можем сопоставлять по шаблону (в case запихнуть ещё один match итд)

```
match  input().split():
    case ['move', *step]:
        match step:
            case ['s']:
                print(‘go to south’)
            case ['n']:
                print(‘go to north’)
            case ['w']:
                print(‘go to west’)
            case ['e']:
                print(‘go to east’)
            case _:
                print('Cannot move to', *step)
```

Следующая фишка - Альтернативы: можно внутренние составляющие объекта указывать через | и прописывать одну логику на разные варианты шаблона

```
>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...     case['help', command]:
...         print(f'The {command} is important!')
...     case[('help' | 'usage'), *cmds]:
...         print("comands are", cmds)
...     case[]:
...         print('What do u want?')
...     case _:
...         print('WAT?')
...
usage
comands are []
```

```
>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...     case['help', command]:
...         print(f'The {command} is important!')
...     case[('help' | 'usage'), *cmds]:
...         print("comands are", cmds)
...     case[]:
...         print('What do u want?')
...     case _:
...         print('WAT?')
...
usage qwerty
comands are ['qwerty']
```

Для альтернатив можно также явно указывать связанные переменные

```
>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...     case['help', command]:
...         print(f'The {command} is important!')
...     case['help' | 'usage' as hlp, *cmds]:
...         print(f'{hlp.upper()}', "comands are", cmds)
...     case[]:
...         print('What do u want?')
...     case _:
...         print('WAT?')
...
usage qwer asdfv cvgr
USAGE comands are ['qwer', 'asdfv', 'cvgr']
```

Следующая фишка - Фильтры. В ***case*** можно использовать клаузу ***if*** для проверки связанных переменных

```
>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...     case['help', command]:
...         print(f'The {command} is important!')
...     case['help' | 'usage' as hlp, *cmds]:
...         print(f'{hlp.upper()}', "comands are", cmds)
...     case[command, *_] if len(command) == 2:
...         print('ERRRR', command)
...     case[]:
...         print('What do u want?')
...     case _:
...         print('WAT?')
...
qwe
WAT?

>>> match input().split():
...     case['help']:
...         print('HELP')
...     case["help", 'me']:
...         print('Sure')
...     case['help', command]:
...         print(f'The {command} is important!')
...     case['help' | 'usage' as hlp, *cmds]:
...         print(f'{hlp.upper()}', "comands are", cmds)
...     case[command, *_] if len(command) == 2:
...         print('ERRRR', command)
...     case[]:
...         print('What do u want?')
...     case _:
...         print('WAT?')
...
qw qwerth
ERRRR qw
```

А вот и особый синтаксис match-case конструкций: обработка типов объектов

```
>>> match eval(input()):
...     case float():
...         print("Float")
...     case int(x):        # x - связанная переменная
...         print('Int', x)
...     case complex(real=1, imag=2):   # Поймает Только комплексные с такими значениями
...         print('1+2j!')
...     case complex(real=0) as y:
...         print("Zero", y)
...
1+2j
1+2j!

>>> match eval(input()):
...     case float():
...         print("Float")
...     case int(x):        # x - связанная переменная
...         print('Int', x)
...     case complex(real=1, imag=2):   # Поймает Только комплексные с такими значениями
...         print('1+2j!')
...     case complex(real=0) as y:
...         print("Zero", y)
...
5j
Zero 5j
```

Получается, что match-case ложится на структуру объекта, а не на синтаксис

Более того, после обработки шаблона связанные переменные не умирают

```
>>> y
5j
```

Экземпляр класса определяется перечислением полей поимённо или (если задано) позиционно

```
>>> from collections import namedtuple
>>> E = namedtuple('E', 'a b')
>>> e = E(1, 2)
>>> e = E(a=1, b=2)
>>>
>>> match e:
...     case E(1, 2):
...         print('QQ')
...
QQ
```

```
>>> C = namedtuple("C", "a b")
>>> for c in C(2, 3), C(1, 2), C(2, 1), C(42, 100500), C(-1, -1):
...     match c:
...         case C(2, 3):                   # Позиционное перечисление
...             print(C, "with 2 and 3")
...         case C(a=1, b=V) | C(a=V, b=1):       # Поимённое перечисление, одна переменная связана...
…             print(C, "with 1 and", V)
...         case C(42):                     # Позиционное задание только одного поля
...             print("Special", C)
...         case C(A, b=B):                   # Одна переменная связана позиционно, другая — именем
...             print("Any", C, "with", A, "and", B)
...
<class '__main__.C'> with 2 and 3
<class '__main__.C'> with 1 and 2
<class '__main__.C'> with 1 and 2
Special <class '__main__.C'>
Any <class '__main__.C'> with -1 and -1
```

Позиционное перечисление полей можно определить вручную спецполем __match_args__:

В шаблоне можно сравнивать всё, даже словарики

```
>>> match dict(enumerate(input().split(), 1)):
...     case {1: "one", 2: "two"}:
...         print("exact")
...     case {1: "one", 2: other}:
...         print("bounded", other)
...     case {2: "two", **others}:
...         print("inexact", others)
...
qwertyjmnbvfd two qsdvg sdfg werf
inexact {1: 'qwertyjmnbvfd', 3: 'qsdvg', 4: 'sdfg', 5: 'werf'}
```