اصول مقدماتی پایتون 6

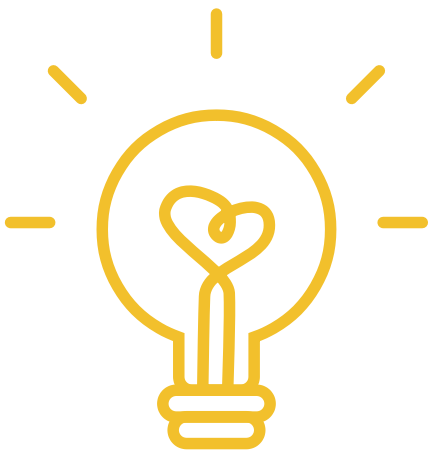
functional programming

هادی فرهادی

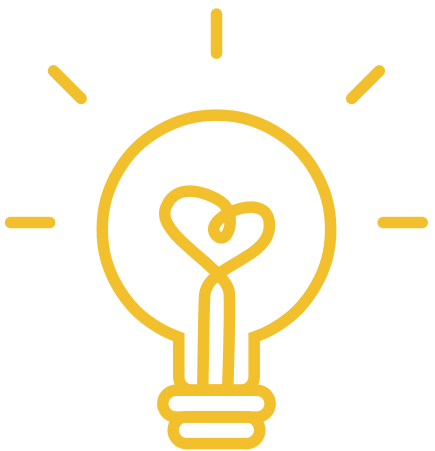
مهر ۱۴۰۴

h.farhadi.py@gmail.com

یادآوری



سؤال جواب



تابع function

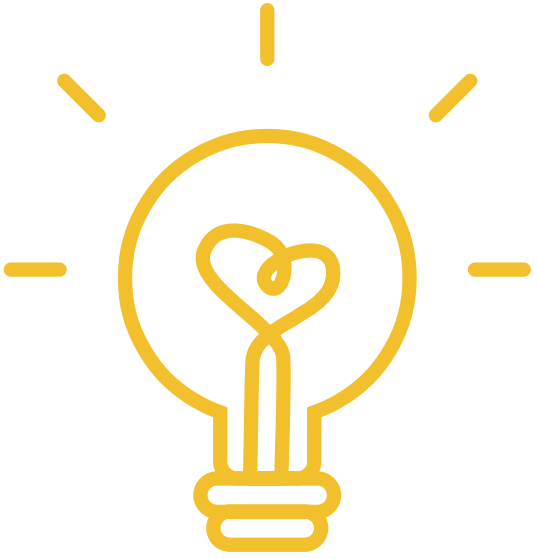
تابع یا function

توابع در پایتون بلوکهای کدی هستند که یک کار خاص را انجام میدهند و میتوانند بارها در برنامه فراخوانی - call شوند.

```
def function_name(parameters: type) → return_type:
    # codes
    return value  # Optional


def say_hello_to_world():
    print("Hello, World!")

# call
say_hello_to_world()
```

تابع یا function

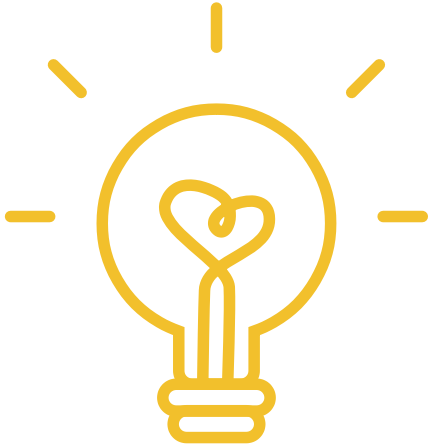
```python
def calculate(first_number: int, second_number: int) -> tuple:
    sum_result = first_number + second_number
    diff_result = first_number - second_number
    return sum_result, diff_result

result = calculate(10, 5)  # result = (15, 5)
sum_result, diff_result = calculate(10, 5)  # unpacking: sum=15, diff=5

print(f"calculation result: {sum_result}")
print(f"sum result: {sum_result}")
print(f"diff result: {diff_result}")
```



تابع یا function

```python
def check_age(age: int) -> str:
    if age < 0:
        return "Invalid age"

    if age >= 18:
        return "Adult"
    else:
        return "Minor"


result = check_age(25)  # "Adult"
print(f"age result: {result}")
```

Functions as Objects in Python



Functions as Objects in Python

در پایتون، توابع اشیاء درجه اول (first-class objects) هستند.

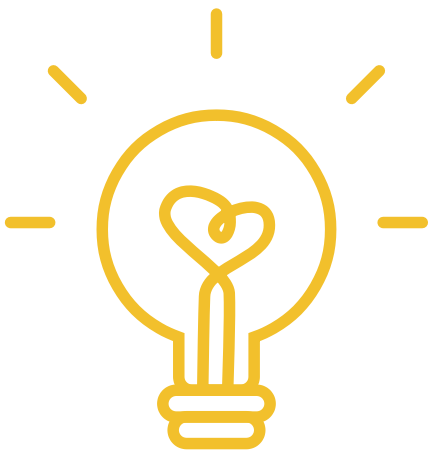
این به معنای آن است که توابع را می‌توان مانند هر شیء دیگری

در پایتون مدیریت کرد، به متغیرها اختصاص داد، به عنوان آرگومان ارسال کرد،

از توابع بازگشت داده شد و در ساختارهای داده ذخیره کرد.

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

def user_operation(name: str, func: any)-> any:
    return func(name)

# assign the method to a variable
my_function = greet

result = user_operation(name="Sara", func=my_function)

print(result)  #  Hello, Sara!
```

lambda



lambda

Lambda functions یا توابع بی‌نام در پایتون، توابع کوچک و یک‌خطی هستند که می‌توانند هر تعداد آرگومان بگیرند اما فقط یک عبارت دارند. این توابع با استفاده از کلمه کلیدی lambda ایجاد می‌شوند.

```
lambda arguments: expression
```

```
add = lambda x, y: x + y

print(add(5, 3))  # output: 8
```

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)
```

lambda

```
words = ["apple", "banana", "cherry", "date"]
sorted_words = sorted(words, key=lambda x: len(x))
print(sorted_words)  # output: ['date', 'apple', 'banana', 'cherry']


pairs = [(1, 5), (3, 2), (2, 8)]
sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs)
```

map

map

تابع map یکی از توابع built-in پایتون است که برای اعمال یک تابع

روی تمام اعضای یک iterable (مانند لیست، تاپل و...) استفاده می‌شود.

```
map(function, iterable, ...)
```

map

اعمال توان دو از طریق تابع map, lambda, function

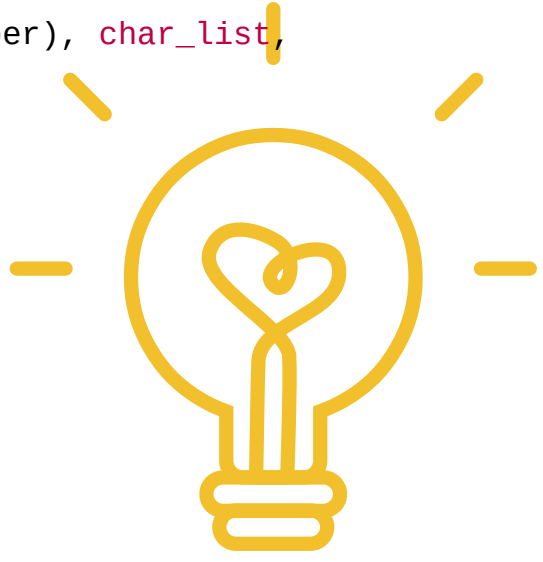
```
number_list: list[int] = [1,2,3,4,5]

squared_map = map(lambda number: number ** 2, number_list) # 1 map object

print("Squared list(lambda)".center(50, '-'))
squared_list = list(squared_map) # cast to list
print(squared_list)


# استفاده از توابع نرمال پایتون
def square_number(number):
    return number ** 2

squared_with_function_map = map(square_number, number_list) # 2

print("Squared list(function)".center(50, '-'))
squared_with_function_list = list(squared_with_function_map)
print(squared_with_function_list)
```

کدام یک از روش های فوق خوانایی بهتری دارد؟

map

به‌دست آوردن عدد متناظر با کاراکتر

```
char_list: list[str] = ["a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"]

ord_number_list: list[int] = list(map(ord, char_list))

print("Ordinal numbers".center(50, "-"))
print(char_list)
print(ord_number_list)

letter_ord_list: list[tuple] = list(map(lambda letter, number: (letter, number), char_list,
ord_number_list))

for char, number in letter_ord_list:
    print(f"{char}: {number}")
```



کدام یک از روش های فوق خوانایی بهتری دارد؟

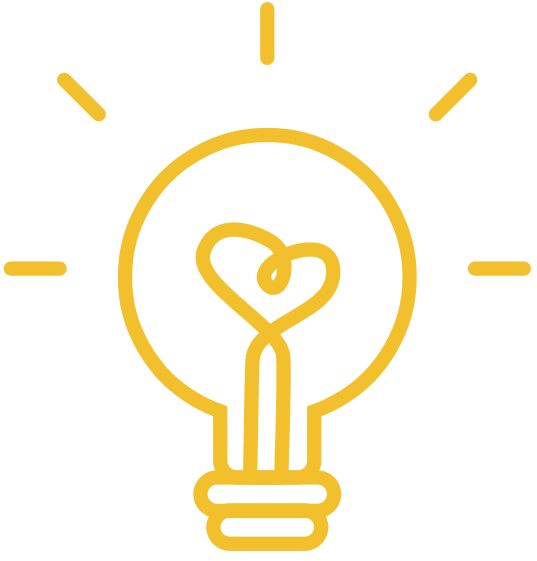
filter

filter

تابع filter یکی از توابع built-in پایتون است که برای فیلتر کردن

عناصر یک iterable بر اساس یک شرط استفاده می‌شود.

```
filter(function, iterable)
```



filter

جدا کردن(پیدا کردن) اعداد زوج از طریق filter

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def is_even(number):
    return number % 2 == 0

print("Filter even numbers(function)".center(50, "-"))

even_numbers = filter(is_even, numbers) # 1
print(list(even_numbers))  # [2, 4, 6, 8, 10]


print("Filter even numbers(lambda)".center(50, "-"))

even_numbers = filter(lambda number: not number % 2, numbers) # 2 the same
variable name
print(list(even_numbers))  # [2, 4, 6, 8, 10]
```

filter

فیلتر کردن اعداد بین 15 تا 35 که بر 5 بخش پذیر باشند filter

```
numbers = [10, 15, 20, 25, 30, 35, 40]

# اعداد بین ۱۵ و ۳۵ که بر ۵ بخش پذیرند
filtered = filter(lambda number: 15 <= number <= 35 and number % 5 == 0,
numbers)
print(list(filtered))  # output: [15, 20, 25, 30, 35]
```

filter

فیلتر اعداد زوج و سپس مربع آنها filter

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

result = map(lambda x: x**2, filter(lambda x: x % 2 == 0, numbers)) # for
odd numbers?
print(list(result))  # [4, 16, 36, 64, 100]
```

reduce

reduce

تابع reduce یکی از توابع مهم برنامه‌نویسی تابعی است که برای کاهش یک دنباله به یک مقدار واحد استفاده می‌شود. این تابع در پایتون ۳ به ماژول functools منتقل شده است.

```
from functools import reduce

reduce(function, iterable[, initializer])
```

function: تابعی که دو پارامتر گرفته و یک مقدار برمی‌گرداند

iterable: شیء قابل پیمایش (لیست، تاپل، رشته و...)

initializer: مقدار اولیه (اختیاری)

reduce دو عنصر اول را با تابع داده شده کال (فراخوانی) می‌کند سپس نتیجه را با عنصر سوم پردازش می‌کند، این روند را تا انتها ادامه می‌دهد و در نهایت یک مقدار واحد برمی‌گرداند

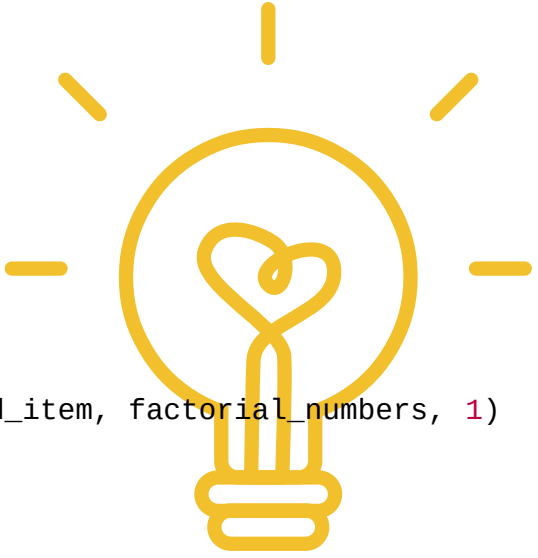
reduce

```
# محاسبه جمع اعداد یک لیست توسط for , reduce

from functools import reduce

numbers = [1, 2, 3, 4, 5]

print(f"numbers: {numbers}")

print("Sum of numbers(for)".center(50, '-'))
# روش سنتی
total = 0
for num in numbers:
    total += num

print(f"Total: {total}") # output: 15

print("Sum of numbers(reduce)".center(50, '-'))
# با reduce
# first step: result = 1 + 2
# second step: result = result + 3
# third step: result = result + 4
total = reduce(lambda x, y: x + y, numbers)
print(f"Total: {total}") # output: 15
```

reduce

محاسبه فاکتوریل یک عدد while, reduce

```
from functools import reduce

# 6! = 6 * 5 * 4 * 3 * 2 * 1
# number! = (number) * (number - 1) * (number - 2) * …

number: int = 6
number_factorial: int = 1

print(f"Factorial of {number} (while)".center(50, "-"))

while number > 0:
    number_factorial *= number
    number -= 1

print(number_factorial)


print(f"Factorial of {number} (reduce)".center(50, "-"))

number: int = 6

factorial_numbers = range(1, number + 1) # 1, 2, 3, 4, 5, 6

factorial_result: int = reduce(lambda first_item, second_item: first_item * second_item, factorial_numbers, 1)

print(factorial_result)
```



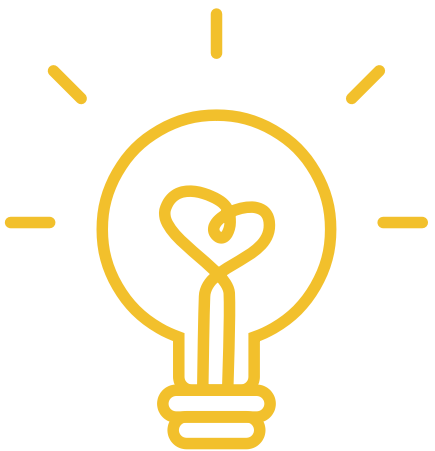
reduce

تابع reduce یک ابزار قدرتمند در برنامه‌نویسی تابعی است، اما بهتر است زمانی استفاده شود که جایگزین ساده‌تری وجود ندارد یا خوانایی کد بهبود می‌یابد.

پیچیدگی: برای مبتدیان ممکن است گیج‌کننده باشد

کارایی: در برخی موارد کندتر از جایگزین‌هاست max(), min(), sum

Generators



Generators

Generators (مولدها) یکی از قدرتمندترین ویژگی‌های پایتون هستند که به شما امکان می‌دهند iteratorهای سفارشی ایجاد کنید بدون اینکه نیاز به پیاده‌سازی کامل کلاس iterator داشته باشید.

Generator نوع خاصی از تابع است که به جای return از yield استفاده می‌کند و حالت (state) خود را بین فراخوانی‌ها حفظ می‌کند.

Generators

```
number_list = [1, 2, 3, 4, 5, 6]

def get_number():
    index = 0
    number = number_list[index]
    index += 1
    return number

print("Default function".center(50, '-'))
print(get_number()) # output: 1
print(get_number()) # output: 1
print(get_number()) # output: 1


def get_number_with_yield():
    index = 0
    number = number_list[index]
    index += 1
    yield number
    number = number_list[index]
    index += 1
    yield number
    number = number_list[index]
    index += 1
    yield number


print("Generator function".center(50, '-'))
number_generator = get_number_with_yield()
print(next(number_generator)) # output: 1
print(next(number_generator)) # output: 2
print(next(number_generator)) # output: 3
```

Generators

```
# مشابه list comprehension اما با پرانتز
generator = (x**2 for x in range(5)) # ساخت یک generator بدون استفاده از تابع

print(list(generator))  # [0, 1, 4, 9, 16] برای داده های بزرگ این عمل خطرناک است
```

Generator برای داده های خیلی بزرگ مناسب هستند چرا که مدیریت حافظه (رم) عالی دارند. چون در هر لحظه به یک عنصر دسترسی داریم next. از جابجایی بین رم و هارد استفاده می کند. پس احتمال پر شدن رم برای داده ای بزرگ به شدت پایین و در حد صفر است. با این اوصاف سرعت کمتری نسبت به لیست داشته باشد

Generators

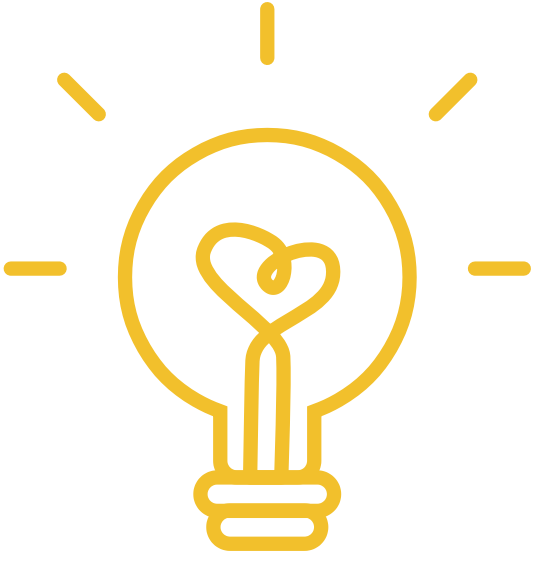
```
def number_generator(n): # به شدت تنبل است تا زمانی
که فراخوانی نشود عدد جدید ایجاد نمی کند
    i = 0
    while i < n:
        yield i
        i += 1

for number in number_generator(5):
    print(number)  # 0, 1, 2, 3, 4
```

داخل for به صورت اتوماتیک عملیات next را انجام می دهد

Generators

Lazy Evaluation (ارزیابی تنبل)

```
def infinite_sequence():
    number = 0
    while True:
        yield number
        number += 1

# استفاده بدون ایجاد لیست بی‌نهایت!
inf_generator = infinite_sequence()
for i in range(5):
    print(next(inf_generator)) # 0, 1, 2, 3, 4
```



Generators

قابلیت ترکیب (Pipelining)

```
def read_numbers():
    for index in range(10):
        yield index

def square(numbers):
    for number in numbers:
        yield number ** 2

def filter_even(squares):
    for square in squares:
        if square % 2 == 0:
            yield square

# ایجاد pipeline
pipeline = filter_even(square(read_numbers()))
print(list(pipeline))  # [0, 4, 16, 36, 64]
```

Generators

خواندن بهینه از فایل

```
def read_file_line_by_line(filename):
    try:
        with open(filename, 'r') as file:
            for line in file:
                yield line.strip()
    except FileNotFoundError:
        print(f'File {filename} not found.')

# استفاده - حافظه کارآمد
for line in read_file_line_by_line('data.txt'):
    print(line)
```

Generators

```
# مقایسه list, generator

import time

def get_numbers(size:int) -> list:
    number: int = 0

    while number < size:
        yield number
        number += 1

start_time = time.time()
for number in get_numbers(10):
    print(number)
end_time = time.time()

print("Generation time".center(50, '-'))
print(end_time - start_time)


start_time = time.time()
for number in range(10):
    print(number)
end_time = time.time()

print("List time".center(50, '-'))
print(end_time - start_time)
```

Decorators

Decorators

Decorators (دکوراتورها) یکی از قدرتمندترین و پیشرفته‌ترین ویژگی‌های پایتون هستند که به شما امکان می‌دهند رفتار توابع یا کلاس‌ها را بدون تغییر کد اصلی آن‌ها تغییر دهید.

Decorator یک تابع است که تابع دیگری را به عنوان ورودی می‌گیرد و عملکرد جدیدی به آن اضافه می‌کند.

```
@decorator
def function():
   Pass
```

```
def function():
    pass
function = decorator(function) # معادل @
```

Decorators

```
def simple_decorator(func):
    def wrapper():
        print("قبل از فراخوانی تابع")
        func()
        print("بعد از فراخوانی تابع")
    return wrapper


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


func = simple_decorator(say_hello)
func()
```

Decorators

```
def simple_decorator(func):
    def wrapper():
        print("قبل از فراخوانی تابع")
        func()
        print("بعد از فراخوانی تابع")
    return wrapper


@simple_decorator
def say_hello():
    print("Hello!")


say_hello()
```

Flask, FastAPI به وفور و django براساس نیاز از دکوریتور استفاده می کند

Decorators

#دکوریتوری که تابع آن ورودی و خروجی دارد

```
def decorator_with_args(func):
    def wrapper(*args, **kwargs):
        print(f"تابع {func.__name__} با آرگومان‌ها فراخوانی شد")
        print(f"آرگومان‌ها: {args}, {kwargs}")
        result = func(*args, **kwargs)
        print(f"نتیجه: {result}")
        return result
    return wrapper

@decorator_with_args
def add(a, b):
    return a + b

result = add(5, 3)
```

Decorators

#دکوریتوری که ورودی دارد

```
def repeat(number):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for index in range(number):
                print(f"execution {index + 1}:")
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Abbas Booazar")
```

Decorators

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start} seconds")
        return result
    return wrapper


def get_number(size: int):
    number: int = 0

    while number < size:
        yield number
        number += 1

@timer
def generate_number_with_generator(size: int):
    for number in get_number(size):
        print(number)

@timer
def generate_number_with_list(size: int):
    for number in range(size):
        print(number)


print("Generating numbers with generator".center(50, '-'))
generate_number_with_generator(10)

print("Generating numbers with list".center(50, '-'))
generate_number_with_list(10)
```

Decorators

کاربردها

لاگینگ و مانیتورینگ

احراز هویت و مجوزها

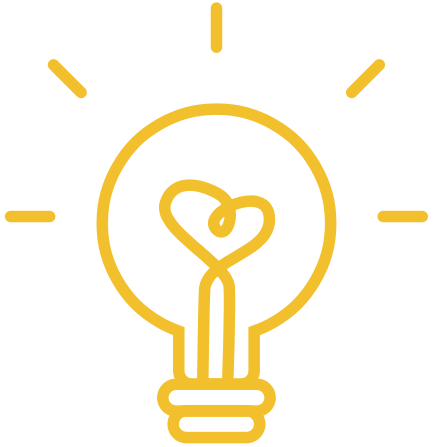
caching و بهینه‌سازی

validation داده‌ها

error handling

timing و پروفایلینگ

توابع بازگشتی (Recursive Functions)



توابع بازگشتی (Recursive Functions)

توابع بازگشتی توابعی هستند که خود را فراخوانی می‌کنند تا یک مسئله را به مسائل کوچکتر تقسیم کنند. این توابع برای حل مسائلی که می‌توانند به زیرمسائل مشابه تقسیم شوند بسیار مناسب هستند.

شرط پایه (Base Case): شرطی که بازگشت را متوقف می‌کند. همیشه شرط پایه داشته باشید: بدون شرط پایه، تابع بی‌نهایت اجرا می‌شود

گام بازگشتی (Recursive Step): فراخوانی تابع با ورودی کوچکتر

توابع بازگشتی (Recursive Functions)

```
def factorial(number: int) -> int:
    # Base Case
    if number == 0 or number == 1:
        return 1
    # Recursive Step
    else:
        return number * factorial(number - 1)

print(factorial(6))  # 120
```

توابع بازگشتی (Recursive Functions)

```
def factorial(number: int) -> int:
    # Base Case
    if number == 0 or number == 1:
        return 1
    # Recursive Step
    else:
        return number *
factorial(number - 1)

print(factorial(1001))  # ?
```

از عمق بازگشت زیاد اجتناب کنید: برای مسائل بزرگ از نسخه تکراری استفاده کنید.

برای مسائل مناسب از بازگشت استفاده کنید: مسائلی که به طور طبیعی بازگشتی هستند.

مصرف حافظه: هر فراخوانی در پشته حافظه مصرف می‌کند.

عمق بازگشت: محدودیت عمق بازگشت در پایتون (حدود 1000 سطح)

سادگی: حل مسائل پیچیده را ساده‌تر می‌کنند

طبیعی بودن: برای مسائل ذاتاً بازگشتی (مثل پیمایش درخت) مناسب هستند. کامنت های تو در تو یک پست

توابع بازگشتی (Recursive Functions)

```
# 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …


def fibonacci(number: int) -> int:
    # شرط پایه
    if number <= 1:
        return number
    # گام بازگشتی
    else:
        return fibonacci(number - 1) + fibonacci(number - 2)

print(fibonacci(10))  # 55
```

Itertools Module

Itertools Module

ماژول itertools یکی از ماژول‌های استاندارد پایتون است که شامل توابعی برای کار با iterator ها می‌باشد.
این ماژول ابزارهای قدرتمندی برای ایجاد و ترکیب iterator ها ارائه می‌دهد.

کارایی حافظه: iteratorها مقادیر را به صورت lazy تولید می‌کنند

سرعت: پیاده‌سازی بهینه‌شده در C

خوانایی: کد تمیزتر و قابل فهم‌تر

Itertools Module

```
import itertools

counter = itertools.count(start=1, step=2)

print("Generating numbers(Itertools.count)".center(50, '-'))

for index in range(10):
    print(next(counter))
```

Itertools Module

```
import itertools

# تکرار متناوب یک دنباله

cycler = itertools.cycle(['A', 'B', 'C'])
for index in range(6):
    print(next(cycler))  # output: A, B, C, A, B, C
```

Itertools Module

```
import itertools

# تکرار یک مقدار به تعداد مشخص

repeater = itertools.repeat('Python', 3)
for item in repeater:
    print(item)  # output: Python, Python, Python
```

Itertools Module

```
import itertools

premier_league_teams: list[str] = [
    "Arsenal",
    "Aston Villa",
    "Bournemouth",
    "Brentford",
    "Brighton & Hove Albion",
    "Chelsea",
    "Crystal Palace",
    "Everton",
    "Fulham",
    "Liverpool",
    "Luton Town",
    "Manchester City",
    "Manchester United",
    "Newcastle United",
    "Nottingham Forest",
    "Sheffield United",
    "Tottenham Hotspur",
    "West Ham United",
    "Wolverhampton Wanderers",
    "Burnley"
]

matches = itertools.permutations(premier_league_teams, 2)

# print(f"Total Matches: {len(list(matches))}") ???

print("Premier League Matches".center(50, "-"))
for match in matches:
    print(f"{match[0]} - {match[1]}")

# print(f"Total Matches: {len(list(matches))}") ???
```

Itertools Module

```
import itertools

# ترکیب‌های بدون ترتیب
items = ['Dembele', 'Lamine', 'Salah']
combination_list = itertools.combinations(items, 2)

print('The combination'.center(50, '-'))
for combination in combination_list:
    print(combination)
```

Itertools Module

```
import itertools

# اتصال چندین دنباله

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]

result = itertools.chain(list1, list2, list3)
print(list(result))  # output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Itertools Module

```
import itertools

# اتصال دنباله‌های تو در تو

nested_lists = [[1, 2], [3, 4], [5, 6]]
result = itertools.chain.from_iterable(nested_lists)
print(list(result))  # [1, 2, 3, 4, 5, 6]
```

Itertools Module

```
from itertools import groupby

# گروه‌بندی دانشجویان بر اساس نمره

students = [
    {'name': 'Ali', 'grade': 'A'},
    {'name': 'Reza', 'grade': 'B'},
    {'name': 'Sara', 'grade': 'A'},
    {'name': 'Maryam', 'grade': 'C'},
    {'name': 'Mohammad', 'grade': 'B'},
]

# مرتب‌سازی بر اساس grade (الزامی برای groupby)

students.sort(key=lambda student: student['grade'])

print("Grouping student(by grade)".center(50, '-'))
for grade, group in groupby(students, key=lambda x: x['grade']):
    print(f"Grade {grade}: {[student['name'] for student in group]}")
```

Functional Programming

Functional Programming

برنامه‌نویسی تابعی یک پارادایم برنامه‌نویسی است که در آن از توابع خالص (Pure Functions) استفاده می‌شود و از تغییر حالت (State) و داده‌های mutable اجتناب می‌شود.

```
# ایجاد متغیر جدید به جای تغییر متغیر اصلی
# incorrect
numbers = ["Dembele", "Yamal", "Salah"]
numbers.append("Rafinia")  # change the main data

# correct
numbers = ["Dembele", "Yamal", "Salah"]
new_numbers = numbers + ["Rafinia"]  # create a new list without reference to the main data

new_numbers.append("Salah")

print(numbers)
print(new_numbers)
```

مصرف حافظه بیشتر

Functional Programming

```
# توابعی که توابع دیگر را به عنوان ورودی می‌گیرند یا برمی‌گردانند

def apply_function(func, number):
    """run another function and return the result"""
    return func(number)

def square(number):
    return number * number

result = apply_function(square, 5)  # 25
```

Mini Project

Premier League

```
premier_league_teams: list[str] = [
    "Arsenal",
    "Aston Villa",
    "Bournemouth",
    "Brentford",
    "Brighton & Hove Albion",
    "Chelsea",
    "Crystal Palace",
    "Everton",
    "Fulham",
    "Liverpool",
    "Luton Town",
    "Manchester City",
    "Manchester United",
    "Newcastle United",
    "Nottingham Forest",
    "Sheffield United",
    "Tottenham Hotspur",
    "West Ham United",
    "Wolverhampton Wanderers",
    "Burnley"
]
```

وظایف:

- ساخت لیست جدول تیم ها همراه با ردیف و امتیاز (لیستی از دیکشنری)
- مرتب سازی کتاب ها براساس امتیاز و حروف الفبا
- ساخت بازی ها آینده براساس generator همراه با زمان بازی ها
- اضافه کردن نتایج و بررسی امتیازات

THANK YOU

h.farhadi.py@gmail.com