PYTHON
ХЕНДБУК

FREE EBOOK

2023

Хендбук Python

February 20, 2023

Карманная книга по Python + Топ 100 вопросов по Python 2023

Содержание

Скачать PDF

Здравствуйте!

Я рад приветствовать вас на страницах этого хендбука по языку программирования Python.

Python - это мощный и гибкий язык программирования, который находит применение в самых различных областях, от разработки веб-приложений и научных исследований до создания игр и машинного обучения. Однако, как и любой другой язык программирования, Python может показаться довольно сложным для начинающих.

Этот хендбук призван помочь вам изучить основы языка Python, научиться использовать различные библиотеки и инструменты, а также дать ответы на самые часто задаваемые вопросы о языке. Я постарался сделать материал доступным и понятным для всех, даже для тех, кто никогда не программировал.

Хендбук по Python - это идеальное решение для тех, кто хочет быстро овладеть основами языка и начать программировать. Книга содержит более 50 тем, которые охватывают все основные конструкции языка, от простых типов данных и операторов до продвинутых тем, таких как объектно-ориентированное программирование и обработка исключений.

Каждая тема описывается кратко и доступно, чтение каждой занимает около 2 минут. Это означает, что вы можете быстро пройти через всю книгу и получить краткий, но полный обзор языка.

Хендбук по Python - это идеальный выбор для тех, кто хочет быстро начать программировать на языке Python. Вы сможете быстро овладеть основными конструкциями и перейти к созданию своих собственных программ.

Я уверен, что этот хендбук поможет вам стать более уверенными в использовании Python, и поможет вам создавать более эффективные программы. Не стесняйтесь задавать вопросы и искать помощи, если вам что-то не ясно - только так можно добиться настоящих результатов в программировании. Удачи!

I - Основы

Данная глава посвящена изучению основ языка программирования Python, который используется для решения различных задач в области науки, инженерии, экономики и многих других областях. В ходе обучения вы познакомитесь с различными типами данных, операторами, условиями, циклами, функциями и классами, а также научитесь работать с файлами и модулями. Этот материал будет полезен как новичкам, так и опытным программистам, желающим расширить свои знания в области Python.

Установка Python

Для установки Python на MacOS можно использовать менеджер пакетов brew. Для этого необходимо выполнить команду:

```
brew install python
```

Для операционных систем на базе Linux также существуют менеджеры пакетов, которые можно использовать для установки Python. Например, для Ubuntu можно использовать команду:

```
sudo apt-get install python
```

Для Windows можно загрузить установочный пакет с официального сайта https://www.python.org/downloads/.

После установки Python необходимо убедиться, что версия Python, установленная на компьютере, соответствует требованиям для запуска необходимых библиотек и инструментов, которые будут использоваться в процессе разработки.

Проверить версию можно в терминале или командной строке набрав команду `python`.

Как только мы запутили Python, можно писать код в терминале.

Среда разработки

Для удобной работы с Python требуется хорошо настроенная рабочая среда. Я предпочитаю использовать **Visual Studio Code** - бесплатный редактор кода, разработанный Microsoft.

Для начала, нужно установить Visual Studio Code на свой компьютер. Это можно сделать с помощью официального сайта https://code.visualstudio.com/.

Установка Python на MacOS и Linux очень проста. Для MacOS можно использовать менеджер пакетов brew, который позволяет установить последнюю версию Python одной командой:

```
brew install --cask visual-studio-code
```

Для Linux, в зависимости от дистрибутива, используется свой менеджер пакетов. Например, для Ubuntu это можно сделать командой:

```
sudo apt-get install code
```

После установки необходимо установить расширение **Python**. Для этого необходимо перейти во вкладку “Extensions”, найти “Python” и нажать кнопку “Install”.

Создайте файл для проекта, например, `example_1.py`.

Напишите код `print("Hello, world!")` и запустите его, нажав на кнопку с треугольником справа вверху:

VSCode запустит код и в нижнем окне программы в терминале вы увидите результат:

Типы данных

Python - это язык программирования, который обладает динамической типизацией, что означает, что тип переменной может меняться в процессе выполнения программы.

В Python есть несколько основных типов данных:

- **Строковые типы** (string)
- **Числовые типы** (целые числа, числа с плавающей запятой, комплексные числа)
- **Логический тип** (True/False)
- **Списки** (list) - это упорядоченная коллекция элементов, которые могут быть различных типов данных.

  Списки создаются при помощи квадратных скобок `[ ]` и элементы списка разделяются запятыми.
- **Кортежи** (tuple) - это упорядоченная коллекция элементов, которые могут быть различных типов данных.

  Кортежи создаются при помощи круглых скобок `( )` и элементы кортежа разделяются запятыми.
- **Словари** (dictionary) - это неупорядоченная коллекция пар “ключ-значение”, где каждый ключ связан со значением.

  Словари создаются при помощи фигурных скобок `{ }` и пары “ключ-значение” разделяются двоеточием, а элементы словаря разделяются запятыми.
- **Множества** (set) - это неупорядоченная коллекция уникальных элементов.

  Множества создаются при помощи фигурных скобок `{ }` и элементы множества разделяются запятыми.

Например, вот как можно создать списки, кортежи, словари и множества в Python:

```
my_list = [1, 2, 3, "four", 5.0]
my_tuple = (1, "two", 3.0, "four", 5)
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_set = {1, 2, 3, 4, 5}
```

Числа

Числовые типы данных в Python могут быть целыми числами (**int**), числами с плавающей запятой (**float**) и комплексными числами (**complex**).

Целые числа - это числа без дробной части, а числа с плавающей запятой - это числа с дробной частью.

Комплексные числа представляются парой вещественных чисел и используются в математических расчетах.

```
a = 5  # целоечисло
b = 3.14  # вещественноечисло
c = 2 + 3j  # комплексноечисло

print(type(a))  # выведет <class 'int'>
print(type(b))  # выведет <class 'float'>
print(type(c))  # выведет <class 'complex'>
```

Python поддерживает все стандартные арифметические операции: сложение, вычитание, умножение, деление, возведение в степень, целочисленное деление и остаток от деления.

```
a = 10
b = 3

print(a + b)  # сложение, выведет 13
print(a - b)  # вычитание, выведет 7
print(a * b)  # умножение, выведет 30
print(a / b)  # деление, выведет 3.3333333333333335
print(a ** b)  # возведениевстепень  , выведет 1000
print(a // b)  # целочисленноеделение  , выведет 3
print(a % b)  # остатокотделения  , выведет 1
```

Строки

В Python существует несколько типов данных. Основные типы данных, с которыми вы, вероятно, будете чаще всего встречаться, - это строка, целое число, плавающая цифра, список, словарь и кортеж. В этой главе мы рассмотрим строковый тип данных. Вы удивитесь, как много вещей можно делать со строками в Python прямо из коробки. Существует также модуль string, который можно импортировать для получения доступа к еще большей функциональности, но мы не будем рассматривать его в этой главе. Вместо этого мы рассмотрим следующие темы:

- Как создавать строки
- Конкатенация строк
- Методы работы со строками
- Нарезка строк
- Подстановка строк

Создание

Строки обычно создаются одним из трех способов. Вы можете использовать одинарные, двойные или тройные кавычки. Давайте посмотрим!

```
>>> text1 = 'Привет, мир!'
>>> text2 = "Python - это замечательный язык программирования"
>>> text3 = '''Строка с тройными кавычками может быть выполнена с помощью трех одинарных или трех двойных кавычек.
При выводе сохраняются разрывы строк.'''
```

Существует еще один способ создания строки - это использование метода **str**:

```
>>> my_number = 123
>>> my_string = str(my_number)
>>>
>>> my_string
'123'
```

Методы

Строка - это объект в Python. Фактически, все в Python является объектом.

Строки в Python поддерживают множество операций, включая конкатенацию (объединение строк), повторение, индексацию, извлечение срезов и многое другое.

```
string1 = 'Привет, '
string2 = 'мир!'
string3 = string1 + string2  # конкатенациястрок
print(string3)  # выведетПривет ', мир!'

string4 = 'Python '
string5 = string4 * 3  # повторениестроки
print(string5)  # выведет 'Python Python Python'

string6 = 'Hello, world!'
print(string6[7])  # индексациясимволов , выведет 'w'
```

```
12
13  string7 = 'Python is awesome'
14  print(string7[0:6])  # извлечениесреза , выведет 'Python'
```

Существует множество других методов работы со строками. Например, если бы вы хотели, чтобы все было в нижнем регистре, вы бы использовали метод **lower()**. Если бы вы хотели удалить все пробелы в начале и в конце строки, вы бы использовали метод **strip()**. Чтобы получить список всех методов работы со строками, введите в интерпретатор следующую команду:

```
>>> dir(my_string)
```

В итоге вы должны увидеть нечто похожее на это:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Вы можете смело игнорировать методы, начинающиеся и заканчивающиеся двойными знаками, такие как **add**. Они не используются в повседневном кодировании на Python. Вместо этого сосредоточьтесь на других методах.

Если вы хотите узнать, что делает один из них, просто попросите **помощи**. Например, вы хотите узнать, для чего нужна **capitalize**. Чтобы узнать это, введите

```
>>> help(my_string.capitalize)
```

Это вернет следующую информацию:

```
Help on built-in function capitalize:

capitalize() method of builtins.str instance
    Return a capitalized version of the string.

    More specifically, make the first character have upper case and the rest lower
    case.
```

Возвращает копию строки S, в которой заглавным является только первый символ.

Вы только что узнали немного о теме, называемой **интроспекцией**. Python позволяет легко проводить интроспекцию всех своих объектов, что делает его очень удобным в использовании. По сути, интроспекция позволяет вам спрашивать Python о самом себе. В одном из предыдущих разделов вы узнали о преобразовании. Возможно, вы задавались вопросом, как определить тип переменной (например, int или string). Вы можете попросить Python рассказать вам об этом!

```
>>> type(my_string)
    <type 'str'>
```

Переменная `my_string` имеет тип `str`.

Разделение строки на подстроки

Одной из задач с которой вы будете часто заниматься в реальном мире, - это разделение строк. Давайте посмотрим, как работает нарезка на примере следующей строки:

```
>>> my_string = "I like Python!"
```

Каждый символ в строке может быть доступен с помощью нарезки. Например, если я хочу получить только первый символ, я могу сделать следующее:

```
>>> my_string[0:4]
'I li'
```

Это захватит первый символ в строке до 4-го символа, но **не включая** его. Да, Python основан на нулях. Это будет немного проще понять, если мы обозначим позицию каждого символа в таблице:

```
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13
I |   | l | i | k | e |   | P | y | t |  h |  o | n  |  !
```

Таким образом, у нас есть строка длиной 14 символов, начинающаяся с нуля и заканчивающаяся тринадцатью. Давайте рассмотрим еще несколько примеров, чтобы лучше закрепить эти понятия в голове.

```
>>> my_string[:1]
'I'
>>> my_string[0:12]
'I like Pytho'
>>> my_string[0:13]
'I like Python'
>>> my_string[0:14]
'I like Python!'
>>> my_string[0:-5]
'I like Py'
>>> my_string[:]
'I like Python!'
>>> my_string[2:]
'like Python!'
```

Форматирование строк

Для форматирования строк в Python есть несколько способов, но одним из наиболее распространенных является метод `format()`. Он позволяет объединять строки и значения переменных, заданных в скобках {}. Например:

```
name = "Alice"
age = 30
print("Меня зовут {}, и мне {} лет".format(name, age))
```

В этом примере мы использовали фигурные скобки для обозначения места, где нужно вставить переменные `name` и `age`. Метод `format()` позволяет использовать несколько переменных, их значения будут подставлены в порядке следования внутри скобок.

Кроме того, можно задать формат вывода для каждой переменной. Например, чтобы вывести значение переменной `age` в шестнадцатеричном формате, можно использовать следующий код:

```
age = 30
print("Мне {} лет, что в шестнадцатеричной системе счисления равно {}".format(age, hex(age)))
```

В результате мы получим вывод: ‘Мне 30 лет, что в шестнадцатеричной системе счисления равно 0x1e’.

Также в Python 3.6 и выше есть более удобный способ форматирования строк, называемый “f-strings” (форматированные строки). В этом случае мы используем символ f перед открывающей кавычкой, а переменные вставляем прямо внутрь фигурных скобок. Например:

```
name = "Alice"
age = 30
print(f"Меня зовут {name}, и мне {age} лет")
```

Этот код даст тот же результат, что и предыдущий.

Ресурсы:

- Официальная документация Python по типу str
- Форматирование строк
- Подробнее о форматировании строк

Списки

Список в Python похож на массив в других языках.

Создание

В Python пустой список может быть создан следующими способами.

```
    my_list = []
    >>> my_list = list()
```

Можно обращаться к элементам списка и кортежа по индексу, начиная с нуля. Например, чтобы получить доступ к первому элементу списка, можно использовать индекс 0:

```
    my_list = [1, 2, 3, "four", 5.0]
    print(my_list[0]) # выводит 1
```

Можно также использовать срезы (**slices**) для получения подмножества элементов списка или кортежа. Например, чтобы получить первые три элемента списка, можно использовать срез `[0:3]`:

```
    my_list = [1, 2, 3, "four", 5.0]
    print(my_list[0:3]) # выводит [1, 2, 3]
```

Вы также можете создавать списки списков следующим образом:

```
    >>> my_nested_list = [my_list, my_list2]
    >>> my_nested_list
    [[1, 2, 3], ['a', 'b', 'c']]
```

Иногда возникает необходимость объединить два списка вместе. Первый способ - использовать метод `extend`:

```
    >>> combo_list = []
    >>> one_list = [4, 5]
    >>> combo_list.extend(one_list)
    >>> combo_list
    [4, 5]
```

Можно просто сложить два списка вместе:

```
    >>> my_list = [1, 2, 3]
    >>> my_list2 = ["a", "b", "c"]
    >>> combo_list = my_list + my_list2
    >>> combo_list
    [1, 2, 3, 'a', 'b', 'c']
```

Методы

Методы списков - это функции, которые могут быть применены к спискам. Некоторые из наиболее распространенны методов:

- `append()`: добавляет элемент в конец списка.
- `insert()`: добавляет элемент в указанное место списка.
- `pop()`: удаляет последний элемент списка и возвращает его.
- `remove()`: удаляет первый элемент списка с указанным значением.
- `sort()`: сортирует элементы списка по возрастанию.
- `reverse()`: переворачивает порядок элементов списка.

Примеры использования методов:

```
    fruits = ['apple', 'banana', 'cherry']
    fruits.append('orange') # ['apple', 'banana', 'cherry', 'orange']
    fruits.insert(1, 'grape') # добавитьпоиндексу   1: ['apple', 'grape', 'banana', 'cherry', 'orange']
    fruits.pop() # ['apple', 'grape', 'banana', 'cherry']
    fruits.remove('banana')
    fruits.sort() #['apple', 'cherry', 'grape']
```

Кортежи

Кортежи в Python - это неизменяемые последовательности элементов, очень похожие на списки.

Создание

Создаются с использованием круглых скобок и могут содержать любые типы данных, в том числе и другие кортежи.

Создание кортежей очень похоже на создание списков, только используются круглые скобки вместо квадратных скобок. Например:

```
t = (1, 2, 3)
another_tuple = tuple()
abc = tuple([4, 5, 6])
```

Мы создали кортеж t, содержащий три элемента. Теперь мы можем обратиться к каждому элементу этого кортежа по его индексу, так же как и в списках:

```
print(t[0])  # выведет 1
print(t[1])  # выведет 2
print(t[2])  # выведет 3
```

Кортежи также могут содержать элементы разных типов данных:

```
t = ("apple", 42, True)
```

Как и в списках, мы можем использовать отрицательные индексы для обращения к элементам кортежа с конца:

```
print(t[-1])  # выведет True
```

Кортежи поддерживают срезы (slicing). Например, мы можем получить подкортеж, состоящий из элементов с индексами от 1 до 2:

```
print(t[1:3])  # выведет (42, True)
```

Методы

Кортежи имеют ряд методов, которые позволяют производить некоторые операции с ними. Однако, поскольку они неизменяемы, многие методы, доступные для списков, недоступны для кортежей. Вот несколько примеров доступных методов:

- count(x) - возвращает количество элементов в кортеже, равных x.
- index(x) - возвращает индекс первого элемента в кортеже, равного x.

Например, мы можем использовать метод `count()` для подсчета количества элементов “apple” в кортеже:

```
t = ("apple", 42, True, "apple", "banana")
print(t.count("apple"))  # выведет 2
```

Или мы можем использовать метод `index()` для поиска индекса первого вхождения элемента “banana” в кортеже:

```
t = ("apple", 42, True, "apple", "banana")
print(t.index("banana"))  # выведет 4
```

Применение

Кортежи могут быть очень полезны, когда вам нужно создать неизменяемый набор данных. Они также могут быть использованы в качестве ключей словаря, потому что они неизменяемы.

Словари

Словарь - это коллекция, которая позволяет хранить пары ключ-значение. В отличие от списков, словари не имеют порядка, и доступ к элементам словаря осуществляется по ключу, а не по индексу.

Создание

Для создания словаря используется фигурная скобка {} с ключами и значениями, разделенными двоеточием. Можно также использовать функцию `dict()` для создания словаря.

Пример создания словаря:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
my_dict_2 = dict(name='Mary', age=30, city='London')
```

Для доступа к элементам словаря используется ключ. Например, чтобы получить значение, связанное с ключом “name”, можно использовать следующий синтаксис:

```
name = my_dict['name']
name = my_dict.get('name', None) # вернет None еслитакогоключанету
```

Чтобы добавить новый элемент в словарь, просто создайте новую пару ключ-значение:

```
my_dict['occupation'] = 'engineer'
```

Методы

- keys(): возвращает все ключи словаря
- values(): возвращает все значения словаря
- items(): возвращает все пары ключ-значение словаря в виде кортежей

А также: `'clear'`, `'copy'`, `'fromkeys'`, `'get'`, `'items'`, `'keys'`, `'pop'`, `'popitem'`, `'setdefault'`, `'update'`, `'values'`

```
keys = my_dict.keys() # Получениевсехключейсловаря
values = my_dict.values() # Получениевсехзначенийсловаря
items = my_dict.items() # Получениевсехпарключзначение - словаря
```

Применение

Словари - это очень мощный инструмент, который часто используется в программировании для хранения и управления данными.

Множества

Создание

Множество можно создать, используя фигурные скобки {} или функцию `set()`:

```
my_set = {1, 2, 3}
print(my_set) # {1, 2, 3}

my_set = set([1, 2, 3])
print(my_set) # {1, 2, 3}
```

Методы

- add(): добавляет элемент в множество.
- remove(): удаляет элемент из множества. Если элемента нет в множестве, возбуждается исключение.
- discard(): удаляет элемент из множества. Если элемента нет в множестве, ничего не происходит.
- union(): возвращает объединение двух множеств.
- intersection(): возвращает пересечение двух множеств.
- difference(): возвращает разность двух множеств.
- symmetric_difference(): возвращает симметрическую разность двух множеств.

Также: `'copy'`, `'difference'`, `'difference_update'`, `'discard'`, `'intersection'`, `'intersection_update'`, `'isdisjoint'`, `'issubse`

```
my_set = {1, 2, 3}
print(my_set) # {1, 2, 3}

# Добавлениеэлемента
my_set.add(4)
print(my_set) # {1, 2, 3, 4}

# Удалениеэлемента
my_set.remove(2)
print(my_set) # {1, 3, 4}

# Объединениемножеств
other_set = {3, 4, 5}
union_set = my_set.union(other_set)
print(union_set) # {1, 3, 4, 5}

# Пересечениемножеств
intersection_set = my_set.intersection(other_set)
print(intersection_set) # {3, 4}

# Разностьмножеств
difference_set = my_set.difference(other_set)
print(difference_set) # {1}

# Симметрическаяразностьмножеств
symmetric_difference_set = my_set.symmetric_difference(other_set)
print(symmetric_difference_set) # {1, 5}
```

Применение

Множества могут использоваться для проверки наличия элемента или для удаления дубликатов из списка:

```
my_list = [1, 2, 2, 3, 4, 4, 5]
my_set = set(my_list)
print(my_set) # {1, 2, 3, 4, 5}

# Проверканаличияэлемента
if 3 in my_set:
    print("3 есть в множестве")

# Удалениедубликатовизсписка
my_list = list(my_set)
print(my_list) # [1, 2, 3, 4, 5]
```

Ресурсы:

- Множества в Python

Операторы

Оператор присваивания

Оператор присваивания “=” используется для присвоения значения переменной. Например:

```
x = 5
```

Арифметические операторы

Арифметические операторы используются для выполнения математических операций над числами.

```
+  # Сложение
-  # Вычитание
```

```
    *  # Умножение
    /  # Деление
    %  # Остатокотделения
    ** # Возведениевстепень
    // # Целочисленноеделение
```

Пример:

```
x = 5
y = 2
print(x + y)  # 7
print(x - y)  # 3
print(x * y)  # 10
print(x / y)  # 2.5
print(x % y)  # 1
print(x ** y) # 25
print(x // y) # 2
```

Операторы сравнения

Операторы сравнения используются для сравнения значений.

```
    == # Равно
    != # Неравно
    >  # Больше
    <  # Меньше
    >= # Большеилиравно
    <= # Меньшеилиравно
```

```
x = 5
y = 2
print(x == y)  # False
print(x != y)  # True
print(x > y)   # True
print(x < y)   # False
print(x >= y)  # True
print(x <= y)  # False
```

Булевы операторы

Булевы операторы используются для выполнения логических операций.

```
    and # ЛогическоеИ
    or  # ЛогическоеИЛИ
    not # ЛогическоеНЕ
```

```
x = 5
y = 2
z = 0
print(x > y and x > z) # True
print(x > y or x < z)  # True
print(not x > y)       # False
```

Побитовые операторы

Побитовые операторы используются для выполнения операций с двоичными числами.

```
    &  # ПобитовоеИ
    |  # ПобитовоеИЛИ
    ^  # ПобитовоеисключающееИЛИ
    ~  # ПобитовоеНЕ
    << # Сдвигвлево
    >> # Сдвигвправо
```

```
x = 5  # 0b101
y = 3  # 0b011
print(x & y)  # 1  (0b001)
print(x | y)  # 7
```

Операторы “is” и “in”

Оператор `is` используется для проверки, являются ли два объекта одним и тем же объектом в памяти.

```
x = [1, 2, 3]
y = [1, 2, 3]

print(x is y)  # False, потомучтоэтодверазныхобъектавпамяти
print(x == y)  # True, потомучтосодержаниеписководинаковое
```

Оператор `in` используется для проверки, находится ли элемент в последовательности.

```
x = [1, 2, 3]
print(2 in x)  # True
print(4 in x)  # False
```

Тернарный оператор

Тернарный оператор - это оператор, который позволяет записать короткое условие в одну строку. Он имеет следующий синтаксис: `value_if_true if condition else value_if_false`.

```
x = 10
y = 20
max_value = x if x > y else y
print(max_value)  # 20
```

В этом примере, если `x` больше `y`, то `max_value` будет равен `x`, иначе `y`.

Условия

В каждом компьютерном языке есть хотя бы один условный оператор. Чаще всего этот оператор представляет собой структуру **if/elif/else**.

В Python 3.10 добавилась структура **match/case**

Оператор if

Позволяет выполнить блок кода, если определенное условие истинно

```
x = 5
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```

match/case

```
def get_day_name(day):
    match day:
        case 1:
            return "Monday"
        case 2:
            return "Tuesday"
        case 3:
```

```
            return "Wednesday"
        case 4:
            return "Thursday"
        case 5:
            return "Friday"
        case 6:
            return "Saturday"
        case 7:
            return "Sunday"
        case _:
            return "Invalid day"
```

```
match command.split():
    case ["quit"]:
        print("Goodbye!")
        quit_game()
    case ["look"]:
        current_room.describe()
    case ["get", obj]:
        character.get(obj, current_room)
    case ["go", direction]:
        current_room = current_room.neighbor(direction)
```

or/and/not

- `or` означает, что если любое условие, которое "перечислено" вместе, равно True, то выполняется следующее утверждение
- `and` означает, что для выполнения следующего утверждения все утверждения должны быть True
- `not` означает, что если условие оценивается как False, то оно является True. На мой взгляд, это самый запутанный вариант.

```
x = 5
y = 10
if x > 0 and y > 0:
    print("Both x and y are positive")
if x > 0 or y > 0:
    print("At least one of x and y is positive")
if not x < 0:
    print("x is not negative")
```

```
my_list = [1, 2, 3, 4]
x = 10
if x not in my_list:
    print("'x' is not in the list, so this is True!")
```

Проверка на ничто (None)

В Python `None` используется, чтобы обозначить отсутствие значения. Это можно использовать в условных операторах, чтобы проверить, имеет ли переменная значение `None`.

Например, если мы хотим проверить, имеет ли переменная `x` значение `None`, мы можем написать:

```
if x is None:
    print("x is None")
```

Мы также можем использовать оператор `is not` для проверки, имеет ли переменная значение, отличное от `None`:

```
if x is not None:
    print("x is not None")
```

Здесь мы используем условный оператор if, чтобы проверить, имеет ли переменная x значение None. Если это так, мы выводим сообщение “x is None”. Если переменная x имеет какое-то другое значение, мы ничего не выводим.

Это может быть полезно, если мы не знаем, какое значение будет иметь переменная, или если переменная может быть пустой.

if name == “main”

Оператор if name == “main” используется для определения, запущен ли файл напрямую или импортирован как модуль. Если файл запущен напрямую, блок кода внутри этого условия будет выполнен, если же файл импортирован как модуль, этот блок кода не будет выполнен:

```
    if __name__ == "__main__":
        # код, которыйбудетвыполнентолькопризапускефайланапрямую
```

Располагается в конце файла. Это говорит Python, что вы хотите выполнить следующий код, только если эта программа будет выполнена как отдельный файл.

Функции

Функция - это структура, которую вы определяете. Вы можете решать, есть ли у них аргументы или нет. Вы можете добавить аргументы в виде ключевых слов и аргументы по умолчанию.

Функция - это блок кода, который начинается с ключевого слова `def`, имени функции и двоеточия. Вот простой пример:

```
    def a_function():
        print("You just created a function!")
```

Эта функция ничего не делает, кроме вывода какого-то текста.

```
    def add(a, b):
        result = a + b
        return result
```

В этом примере мы создали функцию `add`, которая принимает два аргумента `a` и `b` и возвращает их сумму. Вызов функции происходит путем указания имени функции, за которым следуют аргументы в скобках. Пример:

```
    result = add(2, 3)
    print(result) # выводит 5
```

Пустая функция (заглушка)

Иногда, когда вы пишете код, вы просто хотите написать определения функций, не вставляя в них никакого кода.

```
    def empty_function():
            pass
```

Все функции что-то возвращают. Если не указать ей, что она должна что-то вернуть, то она вернет `None`.

Аргументы с ключевыми словами

Функции также могут принимать аргументы в виде ключевых слов! На самом деле они могут принимать как обычные аргументы, так и аргументы с ключевыми словами. Значит, вы можете указать, какие ключевые слова какими являются, и передать их. Вы видели такое поведение в предыдущем примере.

```
    def keyword_function(a=1, b=2):
            return a+b

    keyword_function(b=4, a=5) # 9
```

Вы также могли бы вызвать эту функцию без указания ключевых слов. Эта функция также демонстрирует концепцию аргументов по умолчанию. Каким образом? Ну, попробуйте вызвать функцию вообще без аргументов!

```
keyword_function() # 3
```

*args и **kwargs

Также функции могут принимать переменное число аргументов или аргументы с произвольными именами (как в словарях). Это делается с помощью операторов * и **.

```
def myfunc(*args):
    for arg in args:
        print(arg)
```

Эта функция принимает переменное число аргументов и выводит их все на экран.

Функции в Python также могут иметь аргументы со значениями по умолчанию. Если аргумент не передан при вызове функции, то будет использовано значение по умолчанию. Например:

```
def myfunc(a, b=10):
    result = a + b
    return result
```

В этом примере мы создали функцию `myfunc`, которая принимает два аргумента: `a` и `b` (по умолчанию равный `10`). Если при вызове функции не указан второй аргумент, то он будет равен `10`.

Функции также могут принимать аргументы с ключевыми словами, которые представляют собой пары “ключ-значение”. Эти аргументы позволяют явно указать, какое значение должно быть использовано для каждого параметра функции. Для определения аргументов с ключевыми словами используются двойные звездочки (**).

```
def print_values(**kwargs):
    for key, value in kwargs.items():
        print(key, value)

print_values(name='John', age=25, city='New York')
```

В этом примере функция print_values() принимает произвольное количество аргументов с ключевыми словами и выводит их на экран. При вызове функции передаются аргументы с ключевыми словами name, age, и city, и функция выводит их значения.

Аргументы с ключевыми словами особенно полезны, когда у функции есть множество параметров, и вы хотите явно указать, какое значение должно быть использовано для каждого параметра. Это также может быть полезно, если вы используете библиотеку, которая принимает множество аргументов, и вы хотите быть уверены, что вы передаете значения правильно.

Ресурсы:

- https://vegibit.com/python-function-tutorial/

Область видимости

Область видимости, или **scope**, определяет, где переменные могут быть использованы в программе. В Python есть две основные области видимости: **глобальная** и **локальная**.

Переменные, определенные **внутри функции**, имеют **локальную** область видимости. Это означает, что они могут быть использованы только внутри этой функции. Если попытаться использовать их вне функции, будет вызвано исключение.

Переменные, определенные вне функции, имеют глобальную область видимости. Они могут быть использованы в любом месте программы, в том числе и внутри функций. Если внутри функции определить переменную с тем же именем, что и глобальная переменная, то функция будет использовать локальную переменную.

Пример:

```
x = 10  # глобальнаяпеременная

def my_func():
    x = 5  # локальнаяпеременная
```

```
        print("Значение x внутри функции:", x)

    my_func()
    print("Значение x вне функции:", x)
```

Вывод:

```
Значение
  x внутрифункции : 5Значение
  x внефункции : 10
```

В этом примере мы создали глобальную переменную x со значением 10, а затем определили функцию my_func(), в которой мы создали локальную переменную x со значением 5. Внутри функции мы выводим значение локальной переменной x, а затем вызываем функцию и выводим значение глобальной переменной x.

 Если мы попробуем изменить значение глобальной переменной x внутри функции, то мы получим ошибку:

```
    x = 10  # глобальнаяпеременная

    def my_func():
        x = x + 5  # ошибка: переменная x неопределена
        print("Значение x внутри функции:", x)

    my_func()
    print("Значение x вне функции:", x)
```

В этом примере мы пытаемся изменить значение глобальной переменной x внутри функции, но получаем ошибку, так как переменная x не определена внутри функции.

Чтобы изменить значение глобальной переменной, нужно использовать оператор global.

```
    x = 10  # глобальнаяпеременная

    def my_func():
        global x
        x = x + 5
        print("Значение x внутри функции:", x)

    my_func()
    print("Значение x вне функции:", x)
```

В этом примере мы используем оператор global для того, чтобы указать, что мы хотим использовать глобальную переменную x.

Классы

Все в Python является объектом. Это означает, что каждая сущность в Python имеет методы и значения. Причина в том, что в основе всего лежит класс.

```
    >>> x = "Some String"
    >>> dir(x)
    ['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
    '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
    '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
    '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
    '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
    '_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count',
    'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum',
    'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
    'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
    'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
    'translate', 'upper', 'zfill']
```

Здесь у нас есть строка, присвоенная переменной **x**. Может показаться, что это не так много, но у этой строки есть много методов. Если вы используете ключевое слово **dir** в Python, то сможете получить список всех методов, которые можно вызвать для вашей строки.

Технически мы не должны напрямую вызывать **методы**, начинающиеся с символов подчеркивания, но их можно вызвать.

Это значит, что строка основана на классе, а **x**- это **экземпляр** этого класса!

В Python мы можем создавать свои собственные классы.

Создание класса

Создать класс в Python очень просто. Вот очень простой пример:

```
class MyClass:
    my_attribute = 42

    def my_method(self):
        print("Hello, world!")
```

Здесь мы создали класс `MyClass`, который имеет атрибут `my_attribute` со значением `42` и метод `my_method`, который просто выводит сообщение в консоль.

Атрибуты класса могут быть доступны как через экземпляр класса, так и напрямую через класс:

```
print(MyClass.my_attribute)  # 42

my_object = MyClass()
print(my_object.my_attribute)  # 42
```

Методы класса принимают в качестве первого аргумента экземпляр класса (`self`) и могут иметь доступ к атрибутам класса и вызывать другие методы класса:

```
class MyClass:
    my_attribute = 42

    def my_method(self):
        print(self.my_attribute)

    def my_other_method(self):
        self.my_method()
```

Здесь мы добавили метод `my_other_method`, который просто вызывает метод `my_method`.

В Python существуют специальные методы, которые определяются с помощью двойного подчеркивания в начале и в конце названия метода. Например, метод `__init__` используется для инициализации экземпляра класса при его создании (конструкторы):

```
class MyClass:
    def __init__(self, name):
        self.name = name

    def say_hello(self):
        print("Hello, " + self.name + "!")
```

Здесь мы определили метод `__init__`, который принимает аргумент name и сохраняет его в атрибуте name. Метод say_hello использует этот атрибут для вывода сообщения.

Классы могут **наследовать** друг от друга, позволяя переопределять и расширять функциональность базового класса. Для этого используется ключевое слово super:

Что такое self?

Классы Python нуждаются в способе обращения к самим себе. Это не какое-то самовлюбленное созерцание класса. Напротив, это способ отличить один экземпляр от другого.

Слово `self` - это способ самоописания любого объекта, в буквальном смысле.

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

    def introduce(self):
        print("My name is {} and I'm {} years old.".format(self.name, self.age))
```

```
person1 = Person("Alice", 25)
person1.introduce()
```

Здесь `self.name` и `self.age` представляют атрибуты объекта `Person`, который вызывает метод `introduce`. Без использования `self` мы не могли бы получить доступ к атрибутам объекта из метода.

Наследование

Наследование - это механизм, который позволяет создавать новый класс на основе уже существующего, наследуя его свойства и методы. Класс, от которого наследуется новый класс, называется родительским классом, а новый класс - дочерним классом.

Дочерний класс может использовать свойства и методы родительского класса, а также добавлять свои собственные свойства и методы. Это позволяет создавать более сложные иерархии классов, где дочерние классы наследуют общие свойства и методы от родительского класса, но могут быть уникальными в других отношениях.

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def speak(self):
        print("I am an animal.")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species="Canis")
        self.breed = breed

    def speak(self):
        print("Woof!")

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name, species="Felis")
        self.color = color

    def speak(self):
        print("Meow!")

dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Luna", "Black")

print(dog.name)   # Output: Buddy
print(dog.breed)  # Output: Golden Retriever
dog.speak()       # Output: Woof!

print(cat.name)   # Output: Luna
print(cat.color)  # Output: Black
cat.speak()       # Output: Meow!
```

В этом примере класс `Animal` является **родительским** классом для классов `Dog` и `Cat`.

Класс `Dog` **наследует** свойства `name` и `species` от класса `Animal` и добавляет свой собственный атрибут `breed`.

Класс `Cat` также наследует свойства `name` и `species` от класса `Animal` и добавляет свой собственный атрибут `color`.

У каждого класса есть свой метод `speak`, который переопределяет метод `speak` родительского класса `Animal`. Когда вызывается метод `speak` для экземпляра класса Dog, выводится строка “Woof!”, а когда вызывается для экземпляра класса `Cat`, выводится строка “Meow!”. (Полиморфизм)

Ресурсы:

- https://vegibit.com/python-abstract-base-classes/

Циклы

Цикл while

Цикл `while` повторяет набор инструкций, пока заданное условие истинно. Каждый раз, когда выполняется набор инструкций, условие проверяется снова, и если оно продолжает быть истинным, то набор инструкций выполняется снова.

```
    i = 1
    while i < 6:
        print(i)
        i += 1

    1
    2
    3
    4
    5
```

Цикл for

Цикл for используется для прохождения через элементы в последовательности, такой как список или строка. В отличие от цикла `while`, в цикле `for` не нужно определять начальное условие или шаг увеличения.

```
    fruits = ["apple", "banana", "cherry"]
    for x in fruits:
        print(x)

    apple
    banana
    cherry
```

Операторы break и continue

Оператор break позволяет выйти из цикла, когда выполнено определенное условие, даже если условие продолжает оставаться истинным. Оператор continue позволяет пропустить определенные итерации цикла, когда выполняется определенное условие, и продолжить следующую итерацию.

Пример:

```
    i = 0
    while i < 6:
        i += 1
        if i == 3:
            continue
        print(i)
        if i == 5:
            break

    1
    2
    4
    5
```

else в циклах

Конструкция `else` в циклах в Python выполняется, когда цикл завершается нормально, то есть без использования оператора break. Если оператор break используется в цикле, то блок кода, указанный после else, не будет выполняться.

В цикле `while`, конструкция `else` будет выполнена, когда условие цикла станет ложным, и все итерации будут выполнены.

В цикле for, конструкция `else` будет выполнена после последней итерации, когда больше нет элементов для итерации.

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

for num in numbers:
    if num == 3:
        print("Found 3")
        break
else:
    print("3 not found")
```

В этом примере, если число 3 найдено в списке numbers, то будет выведено “Found 3”. Если число 3 не найдено в списке, то после окончания цикла будет выведено “3 not found”.

Генераторы

В языке Python есть несколько методов создания списков и словарей, которые известны как генераторы.

Генераторы списков

Генератор списка - это выражение, которое генерирует список значений на основе каких-то правил. Вместо того, чтобы создавать список целиком и хранить его в памяти, генератор списка генерирует значения по мере их запроса.

```
squares = [x*x for x in range(10)]
```

Эта строка создает генератор списка, который генерирует квадраты чисел от 0 до 9. Затем можно перебрать элементы этого генератора с помощью цикла:

В Python есть функция `range`, которая может возвращать список чисел. По умолчанию она возвращает целые числа, начиная с 0 и заканчивая числом, которое вы ей передали, но не включая его. В данном случае она возвращает список, содержащий целые числа `0-9`.

```
for square in squares:
    print(square)
```

Генераторы словарей

Генератор словаря работает аналогично генератору списка, но вместо списка мы создаем словарь с помощью фигурных скобок и пары “ключ: значение”.

```
my_dict = {x: x**2 for x in range(5)}
print(my_dict)
# Вывод: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Генератор множеств

Генератор множества используется аналогично генератору списка, но вместо списка мы создаем множество с помощью фигурных скобок.

```
my_set = {x**2 for x in range(5)}
print(my_set)
# Вывод: {0, 1, 4, 9, 16}
```

Здесь мы создаем множество `my_set` с элементами, равными квадратам чисел от 0 до 4.

Ресурсы:

- https://vegibit.com/python-comprehension-tutorial/

Ввод данных пользователем

В Python вы можете запросить у пользователя ввод данных во время выполнения программы. Для этого используется функция input(), которая приостанавливает выполнение программы, ожидает ввода от пользователя и возвращает введенные данные в виде строки.

```
name = input("Введите ваше имя: ")
print("Привет, " + name + "!")
```

При запуске этого кода пользователь увидит приглашение “Введите ваше имя:”, после чего он может ввести свое имя и нажать клавишу Enter. Затем программа поприветствует пользователя по имени.

Вы также можете использовать функцию int() для преобразования введенной строки в целое число. Например:

```
age = int(input("Сколько вам лет? "))
print("В следующем году вам будет", age + 1)
```

Этот код запросит у пользователя возраст, преобразует его в целое число и выводит сообщение о том, сколько ему будет лет в следующем году.

Обратите внимание, что функция input() всегда возвращает строку, поэтому необходимо преобразовывать введенные данные в нужный тип, если это необходимо.

Обработка исключений

Обработка исключений - это механизм, который позволяет обработать возможную ошибку, которая может возникнуть в процессе выполнения программы.

В Python эта конструкция исключений обычно обернута в так называемый **try/except**.

Оператор try-except - это основной инструмент для обработки исключений. Код, который может вызвать исключение, помещается в блок try. Если исключение возникает, то Python переходит в блок except, где вы можете обработать исключение и выполнить соответствующий код.

```
try:
    x = int(input("Введите число: "))
    result = 100 / x
except ZeroDivisionError:
    print("Деление на ноль!")
else:
    print(f"Результат: {result}")
finally:
    print("Конец программы")
```

В этом примере программа просит пользователя ввести число, которое будет использоваться в делении на 100. Если пользователь вводит 0, то возникает исключение ZeroDivisionError, которое обрабатывается блоком except.

В случае, если исключение не возникает, программа выполняет блок else. Независимо от того, возникает исключение или нет, блок finally всегда будет выполнен.

Кроме того, вы можете использовать несколько блоков except для обработки разных типов исключений.

```
try:
    x = int(input("Введите число: "))
    result = 100 / x
except ZeroDivisionError:
    print("Деление на ноль!")
except ValueError:
    print("Неверный формат числа!")
else:
    print(f"Результат: {result}")
finally:
    print("Конец программы")
```

Помимо этого, можно использовать операторы try-except внутри функций, чтобы обрабатывать исключения, возникающие во время их выполнения.

В Python используются операторы raise и assert, которые позволяют вызвать исключение в явном виде, когда это необходимо.

Пример использования оператора raise:

```
    x = -1
    if x < 0:
        raise ValueError("Число должно быть положительным!")
```

Пример использования оператора `assert`:

```
    x = 10
    assert x < 0, "Число должно быть отрицательным!"
```

Оператор `assert` проверяет истинность заданного выражения, и если оно является ложным, вызывает исключение `AssertionError`.

Работа с файлами

Чтение файла

Чтобы прочитать файл в Python, вам нужно сначала открыть файл. Вы можете сделать это, используя функцию `open()`. Эта функция принимает два аргумента: **имя файла** и **режим открытия файла**.

Режим открытия файла может быть “r” (чтение), “w” (запись) или “a” (добавление).

Пример, который читает файл “example.txt” в режиме чтения и выводит его содержимое на экран:

```
    with open("example.txt", "r") as f:
        content = f.read()
        print(content)
```

Мы используем оператор `with`, который автоматически закрывает файл после его использования. Функция `read()` читает содержимое файла и возвращает его в виде строки.

Как читать файлы по частям

Если файл очень большой, то может быть более эффективным читать его по частям.

Самый простой способ читать файл по частям - использовать цикл. Для первого примера мы будем использовать цикл **for**:

```
    handle = open("test.txt", "r")

    for line in handle:
        print(line)

    handle.close()
```

Здесь мы открываем файл в дескрипторе в режиме “только чтение”, а затем используем цикл **for** для итерации по нему.

Вот пример, который читает файл по 100 байтов за раз:

```
    with open("example.txt", "r") as f:
        while True:
            chunk = f.read(100)
            if not chunk:
                break
            print(chunk)
```

Здесь мы используем цикл `while` для чтения файла по частям. Функция `read()` читает 100 байтов за раз и возвращает их в виде строки. Если возвращаемая строка пустая, значит, мы достигли конца файла, и мы выходим из цикла.

Запись файлов

Чтобы записать данные в файл в Python, вам также нужно открыть файл с помощью функции `open()`, но в режиме записи (“w”) или добавления (“a”). Затем вы можете записать данные в файл, используя функцию `write()`.

Вот пример, который записывает строку в файл “example.txt”:

```
    with open("example.txt", "w") as f:
        f.write("Hello, world!")
```

Здесь мы используем функцию `write()`, чтобы записать строку в файл.

Использование оператора with

В Python есть небольшой встроенный оператор **with**, который можно использовать для упрощения чтения и записи файлов. Оператор **with** создает то, что в Python известно как **менеджер контекста**, который автоматически закроет файл, когда вы закончите его обработку. Давайте посмотрим, как это работает:

```
    with open("test.txt") as file_handler:
        for line in file_handler:
            print(line)
```

Импорт модулей

Python поставляется с большим количеством готового кода. Эти части кода известны как модули и пакеты.

Модуль - это один импортируемый файл Python, а пакет состоит из двух или более модулей. Пакет может быть импортирован так же, как и модуль.

В Python вы можете импортировать модули из других файлов, чтобы использовать функции и переменные, определенные в этих модулях.

import

Python предоставляет ключевое слово **import** для импорта модулей.

Допустим, у нас есть два файла:

Файл `dog.py`, содержащий следующий код:

```
    def bark():
        print('Гавгав-!')
```

Файл `main.py`, в котором мы хотим использовать функцию `bark` из `dog.py`:

```
    import dog

    dog.bark()
```

Мы импортируем модуль `dog` в `main.py` с помощью оператора import и затем можем вызывать функцию `bark()` через точку и имя модуля.

from X import Y

Мы также можем импортировать определенные функции или переменные из модуля с помощью оператора `from`.

Допустим, у нас есть файл `math.py`, содержащий функцию `square`, которая возводит число в квадрат:

```
    def square(x):
        return x ** 2
```

В файле `main.py` мы можем импортировать только функцию `square` из `math.py`:

```
    from math import square

    result = square(5)
    print(result)
```

Мы можем использовать `square`, как будто она была определена в `main.py`, и не нужно вызывать ее через точку и имя модуля.

Обратите внимание, что если мы попытаемся вызвать какую-то другую функцию из `math.py`, которая не была импортирована, мы получим ошибку:

```
from math import square

# Ошибка: name 'add' is not defined
result = add(5, 6)
```

import *

В Python можно импортировать все функции из модуля одной командой. Для этого используется символ звездочки (*).

Вот пример:

```
from math import *
```

Эта команда импортирует все функции и константы из модуля math, и мы можем использовать их в нашем коде без префикса math.

Однако, такой подход не рекомендуется, так как может привести к конфликту имен и ухудшить читаемость кода. Вместо этого, лучше явно указывать, какие функции и константы нужны для нашей программы.

Модуль csv Модуль configparser Логирование Модуль sys Модуль os Модуль email / smtplib Модуль sqlite Модуль subprocess Модуль потоков Thread Модуль asyncio

II - Стандартные модули

В Python есть множество встроенных модулей, которые помогают ускорить и упростить написание программ. В этой главе мы рассмотрим некоторые из стандартных модулей Python и их возможности.

Модуль **logging** позволяет вести журнал событий в приложении. С помощью этого модуля можно создавать различные уровни логирования и настраивать их вывод.

Модуль **sys** предоставляет доступ к системным переменным и функциям Python. Например, с помощью этого модуля можно получить информацию о текущей версии Python или переданных параметрах командной строки.

Модуль **os** предоставляет функции для работы с операционной системой, такие как создание и удаление файлов и директорий, запуск новых процессов и многое другое.

Модуль **email и smtplib** используются для отправки электронной почты. Модуль email позволяет создавать электронные письма, а smtplib отправляет их.

Модуль **subprocess** позволяет запускать новые процессы в операционной системе и взаимодействовать с ними.

Модуль **threading** позволяет создавать и управлять потоками выполнения в Python.

Модуль **asyncio** позволяет создавать асинхронный код, что может быть полезным для работы с сетевыми приложениями.

Модуль **datetime** предоставляет классы для работы с датами и временем. С помощью этого модуля можно легко выполнять различные операции с датами и временем, такие как форматирование, расчет разницы между датами и многое другое.

Модуль **configparser** позволяет работать с INI-файлами - это формат файлов конфигурации. С помощью этого модуля можно легко считывать и записывать параметры конфигурации.

Модуль **argparse** в Python предоставляет удобный способ обрабатывать аргументы командной строки.

Модуль os

Модуль `os` предоставляет функции для работы с операционной системой. Этот модуль позволяет получить доступ к файловой системе, управлять процессами, получать информацию об окружении и другие.

- os.listdir - получение списка файлов и директорий в указанной директории:
- os.mkdir() - создание директории
- os.system() - выполнение команды в командной строке
- os.getenv()
- os.putenv()
- os.remove() - удаление файла
- os.rename()
- os.startfile()

- os.walk() - дает способ итерации по пути корневого уровня
- os.environ: словарь, содержащий переменные окружения, доступные в текущем процессе. Можно использовать для получения значения переменной окружения или для установки ее значения.
- os.getcwd(): возвращает текущую рабочую директорию.
- os.chdir(path): изменяет текущую рабочую директорию на указанную.
- os.path.join(path1, path2, ⋯): объединяет несколько путей в один, используя правильный разделитель для операционной системы.
- os.path.exists(path): возвращает True, если файл или директория по указанному пути существует.
- os.path.isfile(path): возвращает True, если путь указывает на существующий файл.
- os.path.isdir(path): возвращает True, если путь указывает на существующую директорию.
- os.makedirs(path): создает директории (в том числе вложенные), если они не существуют.
- os.rmdir(path): удаляет директорию, если она пуста.

```
import os

files = os.listdir(".")
print(f"Files in current directory: {files}") #['file1.txt', 'file2.txt']

os.remove("file.txt")
os.system("ls -l")

# Получениезначенияпеременнойокружения
home_dir = os.environ['HOME']

# Установказначенияпеременнойокружения
os.environ['MY_VAR'] = 'my_value'

# Получениетекущейрабочейдиректории
current_dir = os.getcwd()

# Сменарабочейдиректории
os.chdir('/path/to/new/dir')

# Объединениенескольких путей
full_path = os.path.join('/path/to', 'file.txt')

# Проверканаличияфайла
file_exists = os.path.exists('/path/to/file.txt')

# Проверканаличиядиректории
dir_exists = os.path.isdir('/path/to/dir')

# Созданиедиректории
os.makedirs('/path/to/new/dir')

# Удалениедиректории
os.rmdir('/path/to/dir')

# Итерацияпокаталогам
for root, dirs, files in os.walk(path):
  print(root)
  for _dir in dirs:
      print(_dir)
  for _file in files:
      print(_file)
```

Модуль sys

Модуль **sys** предоставляет специфические для системы параметры и функции. Он содержит системную информацию и функции для взаимодействия со стандартными потоками ввода/вывода, аргументами командной строки и другими модулями Python.

`sys.argv` - список аргументов командной строки, переданных в программу при ее запуске. Первым аргументом обычно является имя файла программы.

`sys.executable` - путь к интерпретатору Python, который используется для запуска текущей программы.

`sys.exit([arg])` - завершает выполнение программы. Если задан аргумент, то он возвращается в качестве кода выхода.

`sys.modules` - словарь, содержащий все загруженные модули Python, включая стандартные и сторонние модули.

`sys.path` - список путей поиска модулей Python. Включает директории, содержащие стандартные модули, а также директории, перечисленные в переменной окружения `PYTHONPATH`.

`sys.platform` - строка, содержащая название операционной системы, на которой запущен Python.

`sys.stdin`, `sys.stdout`, `sys.stderr` - объекты для взаимодействия со стандартными потоками ввода/вывода.

Мы можем использовать `sys.argv` для получения доступа к аргументам командной строки:

```
import sys

# Запуск: python my_program.py arg1 arg2
print(sys.argv)  # ['my_program.py', 'arg1', 'arg2']
```

Атрибут `sys.executable` может быть полезен, если требуется запустить текущую программу с другим интерпретатором Python:

```
import sys
import subprocess

if 'win' in sys.platform:
    python_executable = 'python.exe'
else:
    python_executable = 'python'

subprocess.call([python_executable, 'my_program.py'])
```

`sys.exit()` используется для выхода из программы. Можно передать ей код возврата в качестве аргумента, который будет использоваться для определения статуса выхода:

```
import sys

if len(sys.argv) < 2:
    print('Please specify a file to read')
    sys.exit(1)

filename = sys.argv[1]

# Чтениефайла ...
```

Мы можем использовать `sys.modules` для получения списка всех загруженных модулей:

```
import sys

for name, module in sys.modules.items():
    print(name)
```

Константы `sys.stdin`, `sys.stdout` и `sys.stderr` являются стандартными потоками ввода, вывода и ошибок соответственно.

Например, если мы хотим написать программу, которая запрашивает у пользователя ввод и выводит результат на экран, мы можем использовать `sys.stdin` и `sys.stdout`:

```
import sys

name = input("What is your name? ")
sys.stdout.write(f"Hello, {name}!\n")
```

Здесь мы запрашиваем у пользователя ввод с помощью функции `input()` и выводим результат на экран с помощью `sys.stdout.write()`.

Аналогично, мы можем перенаправить вывод в файл, например:

```
import sys

with open('output.txt', 'w') as f:
    sys.stdout = f
    print('Hello, world!')
```

Здесь мы перенаправляем стандартный вывод в файл “output.txt” с помощью операции присваивания `sys.stdout = f`. Далее, когда мы вызываем функцию print(), результат будет записан в файл вместо вывода на экран.

Модуль logging

Модуль логирования `logging` является одним из стандартных модулей Python и предоставляет возможности для записи логов в приложении. Логирование используется для записи информации о работе приложения, которую можно использовать для отслеживания ошибок и диагностики проблем.

В модуле `logging` определены три основных компонента: логгеры (loggers), обработчики (handlers) и форматировщики (formatters). Логгеры представляют собой объекты, которые используются для записи сообщений лога. Обработчики определяют, куда будут записываться сообщения, а форматировщики определяют, как будут отформатированы эти сообщения.

Пример использования модуля `logging`:

```
import logging

# Созданиелоггера
logger = logging.getLogger('example')

# Установкауровнялогирования
logger.setLevel(logging.INFO)

# Созданиеобработчика
handler = logging.FileHandler('example.log')

# Установкауровнялогированиядляобработчика
handler.setLevel(logging.INFO)

# Созданиеформатировщика
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# Установкаформатировщикадляобработчика
handler.setFormatter(formatter)

# Добавлениеобработчикаклоггеру
logger.addHandler(handler)

# Записьсообщенийлога
logger.debug('Debug message')
logger.info('Info message')
logger.warning('Warning message')
logger.error('Error message')
logger.critical('Critical message')
```

Этот пример создает логгер `example`, который записывает сообщения в файл `example.log`. Уровень логирования установлен на уровень INFO, что означает, что будут записаны сообщения с уровнем INFO и выше. Созданный обработчик определяет, что сообщения будут записываться в файл, а форматировщик определяет, как будут отформатированы сообщения.

Методы `debug`, `info`, `warning`, `error` и `critical` используются для записи сообщений лога разного уровня. В этом примере мы записываем сообщения всех уровней, поэтому в лог-файле будут отображены все эти сообщения.

Это только базовый пример использования модуля `logging`. В реальном приложении вы можете создать несколько логгеров с разными уровнями логирования и разными обработчиками для каждого из них, в зависимости от вашей конкретной задачи.

Модуль datetime/time

datetime

Модуль `datetime` в Python предоставляет классы для работы с датами и временем. Он позволяет создавать объекты даты, времени и даты-времени, а также выполнять операции с этими объектами.

Класс datetime является основным классом модуля datetime и представляет дату и время в формате "ГГГГ-ММ-ДД ЧЧ:ММ:СС". Класс date представляет только дату, а класс time - только время.

Форматирование дат и времени может выполняться с помощью метода strftime, который позволяет создавать строку с заданным форматом даты и времени. Также существует метод strptime, который позволяет преобразовать строку в объект даты и времени.

Для работы со временем и датами можно использовать методы класса datetime, такие как now для получения текущей даты и времени, date и time для получения объектов даты и времени соответственно, а также методы year, month, day, hour, minute, second для получения соответствующих значений.

Класс timedelta позволяет выполнять арифметические операции над объектами дат и времени, такие как сложение и вычитание.

```
import datetime

# Созданиеобъекта  datetime
now = datetime.datetime.now()
print(now)

# Получениеобъекта  date
today = datetime.date.today()
print(today)

# Получениеобъекта  time
current_time = datetime.time(hour=12, minute=30, second=0)
print(current_time)

# Форматированиедатыивремени
formatted_date = now.strftime("%d-%m-%Y")
print(formatted_date)

# Преобразованиестрокивобъект    datetime
date_string = "2022-02-15 18:00:00"
date_object = datetime.datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
print(date_object)

# Использование timedelta
one_day = datetime.timedelta(days=1)
yesterday = today - one_day
print(yesterday)

# Переводдатывстрокуиобратно
date_string = today.strftime("%Y-%m-%d")
date_object = datetime.datetime.strptime(date_string, "%Y-%m-%d")
print(date_object)
```

time

Модуль time в Python предоставляет доступ к системному времени и позволяет работать с временными значениями, такими как время в секундах, часах, минутах и т.д. Этот модуль также содержит функции для задержки выполнения программы, вычисления прошедшего времени и других операций, связанных со временем.

Вот некоторые из наиболее распространенных функций time:

- time(): возвращает текущее время в секундах, начиная с начала эпохи Unix (1 января 1970 года 00:00:00 GMT).
- ctime(): принимает время в секундах в качестве аргумента и возвращает строку с форматированным временем в удобочитаемом формате.
- sleep(): приостанавливает выполнение программы на заданное количество секунд.
- gmtime(): принимает время в секундах в качестве аргумента и возвращает объект структурированного времени, представленного в UTC (координированное всемирное время).
- localtime(): принимает время в секундах в качестве аргумента и возвращает объект структурированного времени, представленного в локальной временной зоне.
- strftime(): преобразует объект структурированного времени в строку с заданным форматом.

```
import time

# Получениетекущеговременивсекундах
current_time = time.time()
print(current_time)

# Отображениевременивудобочитаемомформате
formatted_time = time.ctime(current_time)
print(formatted_time)

# Приостановкавыполненияпрограммына 5 секунд
time.sleep(5)

# Получениеобъектаструктурированноговремени
gm_time = time.gmtime(current_time)
print(gm_time)

# Преобразованиеобъектаструктурированноговременивстроку
formatted_gm_time = time.strftime('%Y-%m-%d %H:%M:%S', gm_time)
print(formatted_gm_time)
```

Модуль потоков threading

Модуль threading в Python предоставляет возможность создавать и управлять потоками выполнения. Потоки - это легковесные процессы, которые выполняются параллельно в пределах одного процесса, что позволяет лучше использовать ресурсы компьютера.

Для создания нового потока необходимо создать объект Thread и передать в его конструктор функцию, которую вы хотите запустить в отдельном потоке. Затем вызовите метод start() у этого объекта, чтобы запустить поток. Если вы хотите дождаться завершения потока, вызовите метод join(), который блокирует текущий поток, пока поток, на который вы вызываете join(), не завершится.

Пример использования модуля threading:

```
from time import sleep
import threading

def print_numbers():
    for i in range(10):
        sleep(1) # задержкапечатидляпримера
        print(i)

def print_letters():
    for letter in ['a', 'b', 'c', 'd', 'e']:
        print(letter)

if __name__ == '__main__':
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

t1.start()
t2.start()
print("Done!")
t1.join()
t2.join()

print("Done!")
```

Здесь мы создали две функции print_numbers() и print_letters(), каждая из которых печатает набор символов в консоль. Затем мы создали два потока, один для каждой из этих функций, и запустили их, вызвав метод start(). Затем мы дождались завершения каждого потока, вызвав метод join(), и напечатали сообщение “Done!”.

В последнем примере кода мы увидим, что каждый поток будет печатать свою информацию в консоль, в произвольном порядке, так как потоки будут конкурировать за доступ к ресурсу (в данном случае, к выводу в консоль).

Результат может отличаться от запуска к запуску программы, так как порядок выполнения потоков не гарантирован и зависит от того, как ОС распределяет ресурсы между потоками.

Модуль `threading` также предоставляет другие полезные классы, такие как `Lock`, `Condition`, `Semaphore`, которые помогают управлять доступом к ресурсам между несколькими потоками.

Модуль subprocess

Модуль `subprocess` является одним из наиболее мощных и распространенных модулей Python для управления другими процессами в операционной системе.

Основная цель `subprocess` заключается в том, чтобы предоставить простой и удобный способ создания новых процессов, подключения к уже существующим процессам, их управления и взаимодействия с ними.

Одним из основных классов в модуле `subprocess` является класс `Popen`, который представляет собой объект, связанный с запущенным в операционной системе процессом.

Например, чтобы запустить новый процесс с помощью `Popen`, мы можем использовать следующий код:

```
import subprocess

process = subprocess.Popen(['ls', '-l'], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
stdout, stderr = process.communicate()
print(stdout.decode())
```

В этом примере мы создаем новый процесс, который выполняет команду `ls -l` в командной строке операционной системы.

Затем мы отправляем строку в стандартный ввод процесса, используя метод `communicate()`, и получаем результат его работы в переменной output.

Наконец, мы выводим содержимое переменной output на экран.

Кроме того, модуль `subprocess` также предоставляет удобный способ проверки состояния завершения процессов с помощью метода `poll()` и ожидания их завершения с помощью метода `wait()`.

В целом, модуль `subprocess` является очень полезным инструментом для управления процессами в операционной системе и взаимодействия с ними из Python.

Модуль argparse

Модуль `argparse` позволяет легко парсить аргументы командной строки.

Это может быть полезно для создания сценариев командной строки, которые должны принимать аргументы от пользователя, например, при написании утилит командной строки.

Пример:

```
#script.py
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

В этом примере мы создали парсер аргументов командной строки с помощью argparse, который принимает целочисленные значения и может вычислить их сумму или максимальное значение. При запуске скрипта мы можем указать значения, например:

```
python script.py 1 2 3 4 --sum
```

Resources:

- argparse tutorial | python.org

Модуль configparser

Модуль `configparser` позволяет работать с конфигурационными файлами в Python.

Для использования модуля `configparser` нужно сначала импортировать его:

```
import configparser
```

Для чтения конфигурационного файла используется метод `configparser.ConfigParser()` с методом `read()`:

```
config = configparser.ConfigParser()
config.read('config.ini')
```

Для записи в конфигурационный файл используется метод write():

```
config.set('section', 'option', 'value')
with open('config.ini', 'w') as f:
    config.write(f)
```

Пример работы с конфигурационным файлом:

```
import configparser

# Создаемобъект ConfigParser
config = configparser.ConfigParser()

# Читаемконфигурационныйфайл
config.read('config.ini')

# Получаемзначениепараметраизсекции
db_name = config.get('database', 'db_name')

# Меняемзначениепараметраизаписываемизменениявфайл
config.set('database', 'db_name', 'new_db_name')
with open('config.ini', 'w') as f:
    config.write(f)
```

Конфигурационный файл может иметь несколько секций, каждая из которых может иметь набор параметров со значениями. Например:

```
[database]
db_name=my_db
db_user=user_name
db_password=secret_password

[server]
host=127.0.0.1
port=8080
```

В данном примере есть две секции: `[database]` и `[server]`. Каждая секция содержит набор параметров со значениями.

Модуль `configparser` позволяет легко работать с этими параметрами, как с обычными переменными. Например, для получения значения параметра db_name из секции database нужно выполнить следующий код:

```
db_name = config.get('database', 'db_name')
```

Параметры в файле могут быть определены без значения, только с именем параметра. В этом случае для получения значения параметра нужно использовать метод `getboolean()`, `getint()` или `getfloat()` в зависимости от типа значения параметра.

Модуль email / smtplib

Модуль `smtplib` в Python предоставляет возможность отправки электронных писем через Simple Mail Transfer Protocol (SMTP).

Он предоставляет класс SMTP, который упрощает отправку электронной почты из Python-скрипта. Модуль `smtplib` позволяет отправлять электронные письма, как с аутентификацией, так и без, и можно отправлять как простые текстовые сообщения, так и письма с HTML-контентом.

Вот пример кода для отправки простого текстового сообщения:

```
import smtplib

smtp_server = 'smtp.yandex.ru'
port = 587
login = 'example@yandex.ru'
password = 'password'
from_addr = 'example@yandex.ru'
to_addr = 'example2@yandex.ru'
message = 'Hello, world!'

with smtplib.SMTP(smtp_server, port) as server:
    server.starttls()
    server.login(login, password)
    server.sendmail(from_addr, to_addr, message)
```

В этом примере мы создаем объект SMTP, указывая адрес сервера и номер порта. Затем мы используем `starttls()`, чтобы начать безопасное соединение и `login()`, чтобы авторизоваться на сервере. Затем мы отправляем электронное письмо с помощью метода `sendmail()`.

Модуль asyncio

> Асинхронное программирование — это концепция программирования, при применении которой запуск длительных операций происходит без ожидания их завершения и не блокирует дальнейшее выполнение программы.
>
> Корутина — это более общая форма подпрограмм. Подпрограммы имеют одну точку входа и одну точку выхода. А корутины поддерживают множество точек входа, выхода и возобновления их выполнения.

Python модуль `asyncio` позволяет заниматься асинхронным программированием с применением конкурентного выполнения кода, основанного на корутинах.

Вот план использования модуля `asyncio`:

```
import asyncio

# Определениеасинхроннойфункцииспомощьюключевогослова async.
async def my_coroutine():
    # code here

# Созданиецикласобытий
loop = asyncio.get_event_loop()

# Запускасопрограммы
loop.run_until_complete(my_coroutine())

# обходасинхронногоитератора
async for item in async_iterator:
    print(item)
```

Можно использовать функцию `asyncio.gather()` для выполнения нескольких сопрограмм параллельно:

```
async def coroutine1():
    print("coroutine1 start")
    await asyncio.sleep(1)
    print("coroutine1 end")

async def coroutine2():
    print("coroutine2 start")
    await asyncio.sleep(2)
    print("coroutine2 end")

async def main():
    await asyncio.gather(coroutine1(), coroutine2())

loop.run_until_complete(main())
```

В этом примере две сопрограммы `coroutine1()` и `coroutine2()` запускаются параллельно с помощью функции `asyncio.gather()`, которая возвращает результаты выполнения всех сопрограмм.

Ресурсы:

- руководство по модулю asyncio в Python | habr

III - Расширенные возможности

В третьей части вы узнаете о некоторых внутренних компонентах Python, которые многие относят к владению Python среднего уровня. Вы перешли от молока и готовы к мясу! В этой части мы рассмотрим следующие темы:

- Отладка
- Декораторы
- Оператор лямбда
- Профилирование кода
- Тестирование

В первой главе этого раздела вы познакомитесь с модулем отладки Python, **pdb**, и узнаете, как использовать его для отладки кода. Следующая глава посвящена декораторам. Вы узнаете о том, как их создавать, и о некоторых декораторах, встроенных в Python. В третьей главе мы рассмотрим оператор лямбда, который, по сути, создает однострочную анонимную функцию. Это немного странно, но весело! В четвертой главе речь пойдет о том, как профилировать свой код. Эта дисциплина дает вам возможность найти возможные узкие места в вашем коде, чтобы вы знали, на чем сосредоточиться для оптимизации кода. Последняя глава этого раздела посвящена тестированию кода. В ней вы узнаете, как тестировать свой код с помощью нескольких встроенных модулей Python.

Лямбда

Лямбда-функции в Python - это безымянные функции, которые можно определить в одной строке и не требуют ключевого слова def. Они используются для написания коротких функций внутри других функций или выражений, где требуется функция в качестве аргумента.

Лямбда-функция определяется ключевым словом **lambda**, за которым следуют параметры функции, после чего через двоеточие указывается выражение, которое нужно вернуть из функции.

Пример:

```
add = lambda x, y: x + y
print(add(2, 3)) # Output: 5
```

Здесь мы определяем лямбда-функцию `add`, которая принимает два аргумента `x` и `y` и возвращает их сумму. Затем мы вызываем эту функцию, передав ей аргументы 2 и 3, и выводим результат, который равен 5.

Лямбда-функции могут использоваться в качестве аргументов для функций высшего порядка, таких как `map`, `filter` или `reduce`. Например, следующий код использует лямбда-функцию для фильтрации списка:

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # Output: [2, 4, 6]
```

Здесь мы используем функцию `filter`, чтобы отфильтровать только четные числа из списка `numbers`. В качестве первого аргумента передаем лямбда-функцию, которая проверяет, является ли число четным. Результат фильтрации преобразуем в список и выводим на экран.

Лямбда-функции также могут использоваться для создания простых обработчиков событий или для задания ключей сортировки. В целом, лямбда-функции могут быть удобным инструментом для написания коротких функций на лету.

Декораторы

Декораторы в Python позволяют изменять поведение функций и методов, оборачивая их в другую функцию. В этом разделе мы рассмотрим несколько встроенных декораторов и создание собственного декоратора.

@classmethod

Декоратор `@classmethod` используется для создания методов класса в Python. Методы класса имеют доступ к состоянию класса и могут использоваться без необходимости создания экземпляра класса. Методы класса можно вызывать как от самого класса, так и от его экземпляров.

Декоратор `@classmethod` применяется к методам класса. Он принимает первым аргументом класс (`cls`) вместо экземпляра класса (`self`).

```
    class MyClass:
        @classmethod
        def my_class_method(cls, arg1, arg2):
            print('Class:', cls, 'arg1:', arg1, 'arg2:', arg2)

    MyClass.my_class_method('a', 'b')
```

@staticmethod

Декоратор `@staticmethod` используется для создания статических методов в Python. Статические методы не имеют доступа к состоянию класса и могут использоваться без необходимости создания экземпляра класса. Статические методы можно вызывать как от самого класса, так и от его экземпляров.

Декоратор `@staticmethod` также применяется к методам класса. Он не принимает первый аргумент, связанный с классом.

```
    class MyClass:
        @staticmethod
        def my_static_method(arg1, arg2):
            print('arg1:', arg1, 'arg2:', arg2)

    MyClass.my_static_method('a', 'b')
```

@property

Декоратор `@property` используется для создания свойств класса в Python. Свойства класса обеспечивают доступ к закрытым переменным класса, так что они могут быть использованы без необходимости создания экземпляра класса. Доступ к свойствам можно получить как чтением, так и записью.

Декоратор `@property` используется для превращения метода в атрибут объекта. Метод, декорированный `@property`, может быть вызван как атрибут объекта, а не как метод.

```
    class MyClass:
        def __init__(self, x):
            self._x = x

        @property
        def x(self):
            return self._x

    my_obj = MyClass(10)
    print(my_obj.x) # 10
```

@contextmanager

Декоратор `@contextmanager` используется для создания менеджера контекста в Python. Менеджеры контекста позволяют определять блоки кода, которые должны быть выполнены с определенными контекстными условиями, такими как открытие и закрытие файлов, установка и восстановление состояния объекта и т. д.

`@contextmanager` позволяет использовать функцию как менеджер контекста с использованием ключевого слова `with`.

```
    from contextlib import contextmanager

    @contextmanager
    def my_context():
        print('entering context')
```

```
        yield
        print('exiting context')

    with my_context():
        print('inside context')
```

@lru_cache

Декоратор `@lru_cache` используется для кэширования результатов функции. Он сохраняет результаты вызовов функции в памяти, чтобы избежать повторных вычислений.

`@lru_cache` использует алгоритм LRU (least recently used) для автоматического удаления наиболее неиспользуемых элементов из кэша.

```
from functools import lru_cache

@lru_cache(maxsize=128)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(30))
```

Создание декоратора

Для создания собственного декоратора в Python нужно определить функцию-обертку, которая будет принимать функцию в качестве аргумента и возвращать новую функцию, изменяющую поведение исходной функции.

Например, создадим декоратор, который будет выводить время выполнения функции:

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Function '{func.__name__}' executed in {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timer
def my_func():
    time.sleep(2)

my_func()
```

Здесь мы определили функцию-обертку `wrapper`, которая принимает любое количество позиционных и именованных аргументов и вызывает исходную функцию `func` с этими аргументами. Затем мы измеряем время выполнения функции, выводим результат и возвращаем его.

Замыкания

Замыкание в Python - это функция, которая запоминает значения из внешней области видимости, даже если эта область видимости больше не существует. Таким образом, замыкание позволяет функции использовать переменные, которые были определены вне самой функции.

Пример:

```
def outer_func(x):
    def inner_func(y):
        return x + y
    return inner_func

```

```
    closure = outer_func(10)
    result = closure(5)
    print(result)  # выводит 15
```

В этом примере `outer_func` возвращает `inner_func`, которая запоминает значение `x`. Затем `outer_func` вызывается, и возвращаемая функция сохраняется в `closure`. Затем `closure` вызывается с аргументом 5, и она использует сохраненное значение x (которое равно 10), чтобы вернуть результат 15.

Замыкания могут быть полезны для создания функций, которые сохраняют состояние между вызовами, а также для создания функций, которые могут быть адаптированы к различным сценариям использования, например для создания функций, которые возвращают другие функции в зависимости от переданных аргументов.

Ниже приведен другой пример замыкания, который возвращает функцию, которая будет умножать аргумент на заданное число:

```
    def multiply_by(num):
        def multiplier(n):
            return n * num
        return multiplier

    double = multiply_by(2)
    triple = multiply_by(3)
    print(double(5))  # выводит 10
    print(triple(5))  # выводит 15
```

В этом примере `multiply_by` возвращает функцию `multiplier`, которая запоминает значение `num`. Затем мы вызываем `multiply_by` два раза с аргументами 2 и 3 соответственно, и сохраняем возвращаемые функции в переменных `double` и `triple`.

Затем мы вызываем каждую из этих функций с аргументом 5, и каждая функция использует сохраненное значение `num` (которое равно 2 для `double` и 3 для `triple`) для умножения аргумента и возврата результата.

Отладка Python

Python поставляется с собственным модулем отладчика, который называется **pdb**. Этот модуль предоставляет интерактивный отладчик исходного кода для ваших программ на Python. Вы можете устанавливать брейкпоинты, просматривать код, изучать кадры стека и многое другое. Мы рассмотрим следующие аспекты этого модуля:

Например, чтобы установить точку останова в коде, можно вставить следующую строку в месте, где вы хотите остановить выполнение программы:

```
    import pdb; pdb.set_trace()
```

После запуска программы выполнение остановится на этой строке, и вы сможете использовать различные команды отладчика для изучения переменных и выполнения других операций.

Также можно запустить python модуль в режиме отладчика:

```
    python3 -m pdb myscript.py
```

Кроме встроенного отладчика Python, есть также сторонние инструменты, такие как PyCharm, Visual Studio Code и Eclipse, которые предоставляют расширенные функции отладки, такие как автоматическое определение ошибок и возможность управления отладкой из пользовательского интерфейса.

Некоторые из основных команд `pdb`:

- break: установить точку останова в коде
- continue: продолжить исполнение программы до следующей точки останова
- step: перейти к следующей строке в коде, вызванной из текущей строки
- next: перейти к следующей строке в коде, не вызывая функции, если таковые имеются
- return: выполнить оставшуюся часть текущей функции и вернуться к вызывающей функции
- list: отобразить несколько строк кода вокруг текущей строки
- print: напечатать значение переменной

Ресурсы:

- The Python Debugger

Тестирование

unittest

Python поставляется со встроенным модулем для тестирования - `unittest`.

Пример теста:

```
import unittest

def square(x):
    return x * x

class TestSquare(unittest.TestCase):
    def test_positive(self):
        self.assertEqual(square(2), 4)
        self.assertEqual(square(3), 9)
        self.assertEqual(square(4), 16)

    def test_negative(self):
        self.assertEqual(square(-2), 4)
        self.assertEqual(square(-3), 9)
        self.assertEqual(square(-4), 16)

if __name__ == '__main__':
    unittest.main()
```

В этом примере мы создаем тестовый класс `TestSquare`, который наследуется от `unittest.TestCase`. В этом классе мы определяем два метода: `test_positive` и `test_negative`. Эти методы используют метод `assertEqual` для проверки ожидаемых результатов.

Метод `assertEqual` сравнивает два значения и генерирует исключение, если они не равны. Если тест проходит успешно, то мы не получаем никаких сообщений.

Запуск тестов можно выполнить из командной строки с помощью следующей команды:

```
python test_square.py
```

Да, в модуле `unittest` есть возможность делать моки с помощью встроенного класса `unittest.mock.Mock`. Это позволяет заменить реальный объект на имитацию, чтобы упростить тестирование и избежать внешних зависимостей.

Вот пример, который демонстрирует, как можно использовать моки в `unittest` для тестирования функции, которая зависит от внешнего сервиса:

```
from unittest import TestCase, mock

def get_external_data():
    # Этовнешнийсервис  , которыйможетвернутьмногоданных
    # Нодлятестированиянасинтересуеттолькопервыйэлемент
    return ['data1', 'data2', 'data3']

def process_data():
    data = get_external_data()
    return data[0]

class TestProcessData(TestCase):

    @mock.patch('__main__.get_external_data')
    def test_process_data(self, mock_get_external_data):
        mock_get_external_data.return_value = ['test_data1', 'test_data2', 'test_data3']
        result = process_data()
        self.assertEqual(result, 'test_data1')
```

Здесь мы используем декоратор `@mock.patch` для замены реального `get_external_data` на имитацию. В тесте мы устанавливаем возвращаемое значение имитации и проверяем, что функция `process_data` вернула ожидаемый результат.

Ресурсы:

- Документация по unittest

IV - Внешние модули

Некоторые из самых популярных пакетов Python по количеству загрузок через PyPI (Python Package Index) за последнее время включают:

- requests: библиотека для HTTP-запросов и взаимодействия с веб-серверами.
- numpy: библиотека для работы с массивами и матрицами в Python.
- pandas: библиотека для обработки и анализа данных в Python.
- matplotlib: библиотека для создания графиков и визуализации данных в Python.
- Flask: легковесный веб-фреймворк для создания веб-приложений на Python.

Установка пакетов

Python-пакеты можно искать на официальном репозитории PyPI (Python Package Index) по адресу https://pypi.org/. В PyPI представлены большинство сторонних пакетов для Python, их можно устанавливать с помощью менеджера пакетов pip.

Также существуют другие источники для поиска и установки Python-пакетов, например, Anaconda, Conda-forge и т.д.

Для установки пакетов в Python существует несколько способов. Рассмотрим наиболее распространенные из них:

Установка с помощью pip

pip - это менеджер пакетов для Python, который упрощает установку, удаление и обновление пакетов. Чтобы установить пакет с помощью pip, необходимо выполнить команду в терминале:

```
pip install package_name
```

Здесь `package_name` - название пакета, который вы хотите установить. Можно также указать конкретную версию пакета:

```
pip install package_name==version_number
```

Кроме того, можно установить пакет из файла, используя команду:

```
pip install path/to/package.whl
```

Установка с помощью Anaconda

Anaconda - это дистрибутив Python, который включает в себя множество научных пакетов и библиотек. Установка пакетов в Anaconda происходит с помощью менеджера пакетов `conda`. Для установки пакета необходимо выполнить команду:

```
conda install package_name
```

Установка из исходников

При установке пакета из исходников необходимо скачать исходный код пакета, распаковать его, перейти в папку с исходниками и выполнить команду:

```
python setup.py install
```

Эта команда выполнит установку пакета.

Важно отметить, что при установке пакетов необходимо убедиться в том, что используется правильная версия Python и что пакеты совместимы с используемой версией Python.

Пакет requests

Модуль `requests` - это сторонняя библиотека Python для отправки HTTP-запросов. Он предоставляет удобный и простой API для отправки GET-, POST-, PUT-, DELETE- и других типов запросов.

Установить `requests` можно с помощью менеджера пакетов pip:

```
pip install requests
```

Пример GET-запроса:

```
import requests

response = requests.get("https://www.example.com")
print(response.status_code)
print(response.text)
```

Пример POST-запроса:

```
import requests

payload = {'key1': 'value1', 'key2': 'value2'}
response = requests.post("https://site.org/post", data=payload)
print(response.status_code)
print(response.json())
```

Модуль requests также поддерживает отправку запросов с использованием сессий, установку заголовков, аутентификацию и другие полезные функции для работы с HTTP-запросами.

Ресурсы:

- https://www.w3schools.com/python/module_requests.asp

V - Фреймворки

Рассмотрим основные фреймворки Python.

Flask - легковесный микрофреймворк, который предоставляет необходимые инструменты для быстрой разработки веб-приложений. Flask использует принцип "минимальности", позволяя разработчикам выбирать только необходимые компоненты, что делает его очень гибким и простым в использовании.

Django - это полноценный фреймворк для создания веб-приложений на языке Python. Он предоставляет широкий спектр инструментов и функциональности, которые облегчают разработку, тестирование и масштабирован приложений. Django имеет встроенную административную панель и ORM-систему, что делает его особенно удобным для разработки сложных веб-приложений.

FastAPI - это быстрый и современный веб-фреймворк, который использует Python 3.7+ типы данных и асинхронную синтаксическую модель. Он предоставляет автоматическую документацию API и мощный систему валидации входных данных. FastAPI быстрый и прост в использовании, что делает его особенно полезным для создания высокопроизводительных и масштабируемых веб-приложений.

Tornado - это фреймворк для создания асинхронных веб-приложений на Python. Он предоставляет быструю и масштабируемую платформу для создания высокопроизводительных веб-приложений. Tornado также обеспечивает возможность использования сокетов, что делает его особенно полезным для создания приложений, которые должны быть связаны с другими приложениями или службами.

Flask

Flask - это легковесный фреймворк для создания веб-приложений на языке Python. Он подходит как для небольших проектов, так и для крупных веб-приложений.

Flask не имеет встроенной базы данных или абстракции уровня модели, поэтому вам нужно будет выбрать библиотеку, которая лучше всего подходит для вашего проекта.

```
pip install flask
pip install flask_sqlalchemy
```

Пример CRUD-операций с использованием Flask:

```python
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///example.db'
db = SQLAlchemy(app)

class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100))
    author = db.Column(db.String(100))

@app.route('/books', methods=['GET'])
def get_all_books():
    books = Book.query.all()
    result = [{'id': book.id, 'title': book.title, 'author': book.author} for book in books]
    return jsonify(result)

@app.route('/books/<int:book_id>', methods=['GET'])
def get_book(book_id):
    book = Book.query.get(book_id)
    if book is None:
        return jsonify({'error': 'Book not found'}), 404
    result = {'id': book.id, 'title': book.title, 'author': book.author}
    return jsonify(result)

@app.route('/books', methods=['POST'])
def create_book():
    book = Book(title=request.json['title'], author=request.json['author'])
    db.session.add(book)
    db.session.commit()
    result = {'id': book.id, 'title': book.title, 'author': book.author}
    return jsonify(result), 201

@app.route('/books/<int:book_id>', methods=['PUT'])
def update_book(book_id):
    book = Book.query.get(book_id)
    if book is None:
        return jsonify({'error': 'Book not found'}), 404
    book.title = request.json['title']
    book.author = request.json['author']
    db.session.commit()
    result = {'id': book.id, 'title': book.title, 'author': book.author}
    return jsonify(result)

@app.route('/books/<int:book_id>', methods=['DELETE'])
def delete_book(book_id):
    book = Book.query.get(book_id)
    if book is None:
        return jsonify({'error': 'Book not found'}), 404
    db.session.delete(book)
    db.session.commit()
    return '', 204
```

Данный код использует Flask вместе с библиотекой SQLAlchemy для создания веб-приложения и взаимодействия с базой данных. Роуты приложения обрабатывают HTTP-запросы и возвращают соответствующий HTTP-ответ. В данном примере реализованы операции CRUD (Create, Read, Update, Delete) для модели Book.

Ресурсы:

- Официальная документация Flask

Django

Django - это высокоуровневый фреймворк для веб-приложений на языке Python. Он предоставляет множество инструментов для разработки сайтов, начиная от автоматического создания административного интерфейса до работы с базами данных. Основными принципами, которыми руководствуется Django, являются: быстрота разработки, возможность переиспользования кода и расширяемость.

Установим необходимые пакеты:

```
pip install django
```

Для начала работы с Django нужно создать проект. Для этого в командной строке нужно ввести команду:

```
django-admin startproject project_name
```

После этого будет создан проект с именем “project_name”. Внутри проекта есть файлы настроек и приложения. Приложение - это часть проекта, которая отвечает за определенную функциональность.

Для создания приложения нужно ввести команду:

```
python manage.py startapp app_name
```

Далее можно начинать разработку функциональности внутри приложения.

Пример реализации CRUD операций с использованием Django:

```
from django.shortcuts import render, get_object_or_404
from django.http import HttpResponseRedirect
from django.urls import reverse
from .models import Book

def index(request):
    books = Book.objects.all()
    return render(request, 'index.html', {'books': books})

def create(request):
    if request.method == 'POST':
        book = Book(
            title=request.POST.get('title'),
            author=request.POST.get('author'),
            published_date=request.POST.get('published_date')
        )
        book.save()
        return HttpResponseRedirect(reverse('index'))
    return render(request, 'create.html')

def update(request, book_id):
    book = get_object_or_404(Book, pk=book_id)
    if request.method == 'POST':
        book.title = request.POST.get('title')
        book.author = request.POST.get('author')
        book.published_date = request.POST.get('published_date')
        book.save()
        return HttpResponseRedirect(reverse('index'))
    return render(request, 'update.html', {'book': book})

def delete(request, book_id):
    book = get_object_or_404(Book, pk=book_id)
    book.delete()
    return HttpResponseRedirect(reverse('index'))
```

В данном примере определены функции для отображения списка книг (index), создания новой книги (create), обновления существующей книги (update) и удаления книги (delete). Все эти функции используют модель Book, которая определена в файле models.py. Шаблоны (templates) для каждой из функций находятся в отдельных html-файлах.

Ресурсы:

- Официальная документация Django

FastAPI

FastAPI - это фреймворк для создания веб-приложений на Python, использующий современный подход к созданию API и основанный на ASGI-серверах. Он разработан с упором на скорость и быстродействие, предоставляя возможности асинхронного выполнения запросов, автоматического документирования API и многие другие.

Для установки `FastAPI` нужно выполнить команду `pip install fastapi`. Для запуска приложения можно использовать стандартный инструмент `uvicorn`, который также необходимо установить: `pip install uvicorn`.

Пример CRUD приложения на FastAPI:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import Dict

app = FastAPI()

# Имитациябазыданных
db = {}

# Модельдлясозданияредактирования / задачи
class Task(BaseModel):
    title: str
    description: str

# Модельдляответасосписkomзадач
class TaskList(BaseModel):
    tasks: Dict[int, Task]

# Получениесписказадач
@app.get("/tasks/", response_model=TaskList)
async def get_tasks():
    return TaskList(tasks=db)

# Получениеоднойзадачипо id
@app.get("/tasks/{task_id}")
async def get_task(task_id: int):
    if task_id not in db:
        raise HTTPException(status_code=404, detail="Task not found")
    return db[task_id]

# Созданиеновойзадачи
@app.post("/tasks/")
async def create_task(task: Task):
    task_id = max(db.keys(), default=0) + 1
    db[task_id] = task
    return {"id": task_id}

# Редактированиезадачи
@app.put("/tasks/{task_id}")
async def update_task(task_id: int, task: Task):
    if task_id not in db:
        raise HTTPException(status_code=404, detail="Task not found")
    db[task_id] = task
    return {"message": "Task has been updated"}

# Удалениезадачи
@app.delete("/tasks/{task_id}")
async def delete_task(task_id: int):
    if task_id not in db:
        raise HTTPException(status_code=404, detail="Task not found")
    db.pop(task_id)
    return {"message": "Task has been deleted"}
```

Этот код создает простое приложение с API для управления задачами. Он использует модели Pydantic для валидации данных, а также async/await синтаксис для асинхронной обработки запросов. Код использует декораторы FastAPI для определения конечных точек API (маршрутов), а также для указания

моделей данных, которые используются для запросов и ответов.

Ресурсы:

- Официальная документация FastAPI content/tracks/python-101/400_frameworks/403_fastapi.ru.md

Tornado

Tornado - это еще один быстрый веб-фреймворк, который разработан для обработки больших объемов трафика в режиме реального времени.

Для начала работы с Tornado нам нужно установить его, используя команду pip:

```
pip install tornado
```

```
import tornado.ioloop
import tornado.web
import tornado.escape

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        items = [{'id': 1, 'name': 'Item 1'}, {'id': 2, 'name': 'Item 2'}]
        self.write(tornado.escape.json_encode(items))

class ItemHandler(tornado.web.RequestHandler):
    def get(self, id):
        item = {'id': id, 'name': 'Item ' + id}
        self.write(tornado.escape.json_encode(item))

    def post(self, id):
        item = {'id': id, 'name': self.get_argument('name')}
        self.write(tornado.escape.json_encode(item))

    def put(self, id):
        item = {'id': id, 'name': self.get_argument('name')}
        self.write(tornado.escape.json_encode(item))

    def delete(self, id):
        self.write('Item ' + id + ' deleted')

def make_app():
    return tornado.web.Application([
        (r'/', MainHandler),
        (r'/item/(\d+)', ItemHandler),
    ])

if __name__ == '__main__':
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

В этом примере мы создаем два класса-обработчика, один для главной страницы, другой для работы с конкретным элементом. Для тестирования мы создаем два элемента и возвращаем их в формате JSON при запросе к главной странице.

Когда мы запрашиваем элемент, создается элемент соответствующий запрошенному и возвращается в формате JSON. Методы `post`, `put` и `delete` принимают данные из тела запроса и выполняют соответствующую операцию.

Запуск приложения осуществляется через командную строку:

```
python tornado_app.py
```

После запуска приложения, мы можем обращаться к нему через браузер по адресу http://localhost:8888/. При обращении к адресу http://localhost:8888/item/1, мы получим объект с идентификатором 1 в формате JSON.

При выполнении запроса `post` на тот же URL с параметрами, мы создадим новый элемент.

При запросе `put` мы обновим данные существующего элемента, а при выполнении `delete` - удалим элемент с указанным идентификатором.

Ресурсы:

- Официальная документация Tornado

Топ 100 вопросов по Python

В процессе. Дедлайн: 28/02/2023

Скачать PDF

Junior

1. Что такое Python? Какие преимущества использования Python?

Python - это высокоуровневый интерпретируемый язык программирования общего назначения. Будучи языком общего назначения, он может быть использован для создания практически любого типа приложений при наличии соответствующих инструментов/библиотек.

Кроме того, python поддерживает объекты, модули, потоки, обработку исключений и автоматическое управление памятью, что помогает моделировать реальные проблемы и создавать приложения для решения этих проблем.

Преимущества использования Python:

Python - это язык программирования общего назначения, который имеет простой, легко изучаемый синтаксис, подчеркивающий удобочитаемость и, следовательно, снижающий затраты на сопровождение программ. Более того, язык способен выполнять сценарии, является полностью открытым и поддерживает пакеты сторонних разработчиков, что способствует модульности и повторному использованию кода.

Его высокоуровневые структуры данных в сочетании с динамической типизацией и динамическим связыванием привлекают огромное сообщество разработчиков для быстрой разработки и развертывания приложений.

2. Что такое динамически типизированный язык?

Прежде чем понять, что такое динамически типизированный язык, мы должны узнать, что такое типизация. Типизация относится к проверке типов в языках программирования. В языке с сильной типизацией, таком как Python, “1” + 2 приведет к ошибке типа, поскольку эти языки не допускают “приведения типов” (неявного преобразования типов данных). С другой стороны, слабо типизированный язык, такой как JavaScript, просто выведет “12” в качестве результата.

Проверка типов может быть выполнена на двух этапах:

1. Статический - типы данных проверяются перед выполнением.
2. Динамический - типы данных проверяются во время выполнения.

Python - интерпретируемый язык, каждый оператор выполняется построчно, поэтому проверка типов выполняется на лету, во время выполнения. Следовательно, Python является динамически типизированным языком.

3. Что такое интерпретируемый язык?

Интерпретированный язык выполняет свои утверждения построчно. Такие языки, как Python, JavaScript, R, PHP и Ruby, являются яркими примерами интерпретируемых языков. Программы, написанные на интерпретируемом языке, выполняются непосредственно из исходного кода, без промежуточного этапа компиляции.

4. Что такое PEP 8 и почему он важен?

PEP расшифровывается как Python Enhancement Proposal. PEP - это официальный проектный документ, предоставляющий информацию сообществу Python или описывающий новую функцию для Python или его процессов.

PEP 8 особенно важен, поскольку в нем документированы руководящие принципы стиля для кода Python. Очевидно, что вклад в сообщество разработчиков открытого кода Python требует от вас искреннего и строгого следования этим руководящим принципам стиля.

5. Что такое область видимости в Python?

Каждый объект в Python функционирует в пределах области видимости. **Область видимости** - это блок кода, в котором объект в Python остается актуальным. Пространства имен однозначно идентифицируют все объекты внутри программы.

В Python существует **3 области видимости**:

1. Локальная
2. Глобальная
3. Нелокальная

Однако эти пространства имен также имеют область видимости, определенную для них, где вы можете использовать их объекты без префикса. Ниже приведено несколько примеров областей видимости, создаваемых во время выполнения кода в Python:

Локальная область видимости относится к локальным объектам, доступным в текущей функции.

Локальная область видимости - определенная внутри функции, метода или выражения. Переменные, определенные внутри этой области, недоступны за ее пределами.

```
x = 10

def my_func(a, b):
    print(x)
    print(z)

>>>my_func(1, 2)

10
Traceback (most recent call last):
    File "<pyshell#19>", line 1, in <module>
        my_func(1, 2)
    File "<pyshell#18>", line 3, in my_func
        print(z)
NameError: name 'z' is not defined
```

Глобальная область видимости относится к объектам, доступным во время выполнения кода с момента их создания.

Глобальная область видимости - определенная вне функций, методов и выражений. Переменные, определенные в глобальной области видимости, доступны везде в коде.

Область видимости на уровне модуля относится к глобальным объектам текущего модуля, доступным в программе.

Глобальная область видимости относится ко всем встроенным именам, вызываемым в программе. Объекты в этой области видимости ищутся в последнюю очередь, чтобы найти имя, на которое ссылаются.

```
def my_func(a, b):
    global x
    print(x)
    x = 5
    print(x)

if __name__ == '__main__':
    x = 10
    my_func(1, 2)
    print(x)

10
5
5
```

 Примечание: Объекты локальной области видимости могут быть синхронизированы с объектами глобальной области видимости с помощью таких ключевых слов, как **global**.

В Python 3 было добавлено новое ключевое слово под названием `nonlocal`. С его помощью мы можем добавлять переопределение области во внутреннюю область. Вы можете ознакомиться со всей необходимой на данный счет информацией в PEP 3104. Это наглядно демонстрируется в нескольких примерах. Один из самых простых – это создание функции, которая может увеличиваться:

```
    def counter():
        num = 0
        def incrementer():
            num += 1
            return num
        return incrementer
```

Если вы попробуете запустить этот код, вы получите ошибку UnboundLocalError, так как переменная num ссылается прежде, чем она будет назначена в самой внутренней функции. Давайте добавим nonlocal в наш код:

```
    def counter():
        num = 0
        def incrementer():
            nonlocal num
            num += 1
            return num
        return incrementer

    c = counter()
    print(c) # <function counter.<locals>.incrementer at 0x7f45caf44048>

    c() # 1
    c() # 2
    c() # 3
```

6. Что такое списки и кортежи? В чем ключевое различие между ними?

Списки и кортежи - это типы данных последовательности, которые могут хранить коллекцию объектов в Python. Объекты, хранящиеся в обеих последовательностях, могут иметь различные типы данных. Списки представлены квадратными скобками `['sara', 6, 0.19]`, а кортежи - круглыми `('ansh', 5, 0.97)`.

Но в чем реальная разница между ними? Ключевое различие между ними заключается в том, что списки являются изменяемыми, а кортежи, напротив, неизменяемыми объектами. Это означает, что списки можно изменять, добавлять или нарезать на ходу, а кортежи остаются неизменными и не могут быть изменены никаким образом. Вы можете выполнить следующий пример, чтобы убедиться в разнице:

```
    my_tuple = ('sara', 6, 5, 0.97)
    my_list = ['sara', 6, 5, 0.97]
    print(my_tuple[0])     # output => 'sara'
    print(my_list[0])     # output => 'sara'
    my_tuple[0] = 'ansh'    # modifying tuple => throws an error
    my_list[0] = 'ansh'    # modifying list => list modified
    print(my_tuple[0])     # output => 'sara'
    print(my_list[0])     # output => 'ansh'
```

7. Каковы общие встроенные типы данных в Python?

В Python существует несколько встроенных типов данных. Хотя Python не требует явного определения типов данных при объявлении переменных, ошибки могут возникнуть, если пренебречь знанием типов данных и их совместимости друг с другом. Python предоставляет функции type() и isinstance() для проверки типа этих переменных. Эти типы данных можно сгруппировать в следующие категории-

Тип None:

Ключевое слово **None** представляет нулевые значения в Python. Операция булева равенства может быть выполнена с использованием этих объектов **NoneType**.

NoneType Представляет значения `NULL` в Python.

Числовые типы:

Существует три различных числовых типа - целые числа (integers), числа с плавающей точкой (floating-point) и комплексные числа(complex numbers). Кроме того, булевы числа являются подтипом целых чисел.

- **int** Хранит целочисленные литералы, включая шестнадцатеричные, восьмеричные и двоичные числа, как целые числа

- **float** Хранит литералы, содержащие десятичные значения и/или знаки экспоненты, как числа с плавающей точкой
- **complex** Хранит комплексные числа в виде `(A + Bj)` и имеет атрибуты: real и imag
- **bool** Хранит булево значение (True или False).

Типы последовательностей:
Согласно Python Docs, существует три основных типа последовательностей - списки (lists), кортежи (tuples) и объекты диапазона (range objects). Типы последовательностей имеют операторы `in` и `not in`, определенные для обхода их элементов. Эти операторы имеют тот же приоритет, что и операции сравнения.

- **list** Неизменяемая последовательность, используемая для хранения коллекции элементов.
- **tuple** Неизменяемая последовательность, используемая для хранения коллекции элементов.
- **range** Представляет собой неизменяемую последовательность чисел, генерируемую во время выполнения.
- **str** Неизменяемая последовательность кодовых точек Unicode для хранения текстовых данных.

Стандартная библиотека также включает дополнительные типы для обработки:

1. Двоичные данные
2. Текстовые строки, такие как `str`.

Тип словарь (dict):
Объект отображения может отображать хэшируемые значения на произвольные объекты в Python. Объекты отображения являются изменяемыми, и в настоящее время существует только один стандартный тип отображения - **dict**.

- **dict** Хранит список пар ключ: значение, разделенных запятыми.

Типы множеств:
В настоящее время в Python есть два встроенных типа множеств - **set** и **frozenset**.
Тип **set** является изменяемым и поддерживает такие методы, как `add()` и `remove()`.
Тип **frozenset** является неизменяемым и не может быть изменен после создания.

- **set** Мутабельная неупорядоченная коллекция отдельных хэшируемых объектов.
- **frozenset** Неизменяемая коллекция отдельных хэшируемых объектов.

set является изменяемым и поэтому не может быть использован в качестве ключа словаря. С другой стороны, **frozenset** является неизменяемым и, следовательно, хэшируемым, и может использоваться как ключ словаря или как элемент другого множества.
Модули:
Module - это дополнительный встроенный тип, поддерживаемый интерпретатором Python. Он поддерживает одну специальную операцию, т.е. доступ к атрибуту: `mymod.myobj`, где `mymod` - модуль, а `myobj` ссылается на имя, определенное в модуле.
Таблица символов модуля находится в специальном атрибуте модуля **dict**, но прямое присвоение этому модулю невозможно и не рекомендуется.
Типы Callable:
Callable типы - это типы, к которым может быть применен вызов функции. Это могут быть определяемые пользователем функции, методы экземпляра, функции генератора и некоторые другие встроенные функции, методы и классы.

8. Что такое pass в Python?

Ключевое слово **pass** представляет собой нулевую операцию в Python. Обычно оно используется для заполнения пустых блоков кода, который может выполняться во время исполнения, но еще не написан. Без оператора **pass** в следующем коде мы можем столкнуться с некоторыми ошибками во время выполнения кода.

```
    def myEmptyFunc():
       # do nothing
       pass
    myEmptyFunc()    # nothing happens
    ## Without the pass keyword
    # File "<stdin>", line 3
    # IndentationError: expected an indented block
```

9. Что такое модули и пакеты в Python?

Пакеты Python и модули Python - это два механизма, которые позволяют осуществлять модульное программировани в Python. Модулирование имеет несколько преимуществ:

- Простота: Работа над одним модулем помогает сосредоточиться на относительно небольшой части решаемой задачи. Это делает разработку более простой и менее подверженной ошибкам.
- Удобство обслуживания: Модули предназначены для обеспечения логических границ между различными проблемными областями. Если они написаны таким образом, что уменьшают взаимозависимость, то меньше вероятность того, что изменения в модуле могут повлиять на другие части программы.
- Возможность повторного использования: Функции, определенные в модуле, могут быть легко использованы повторно в других частях приложения.
- Разметка: Модули обычно определяют отдельное пространство имен, что помогает избежать путаницы между идентификаторами из других частей программы.

Модули, в общем случае, это просто файлы Python с расширением `.py`, в которых может быть определен и реализован набор функций, классов или переменных. Они могут быть импортированы и инициализированы один раз с помощью оператора `import`. Если требуется частичная функциональность, импортируйте необходимые классы или функции с помощью оператора `import`: `from foo import bar`.

Пакеты позволяют иерархически структурировать пространство имен модуля с помощью точечной нотации. Как модули помогают избежать столкновений между именами глобальных переменных, так и пакеты помогают избежать столкновений между именами модулей.

Создать пакет очень просто, поскольку он использует присущую системе файловую структуру. Просто поместите модули в папку, и вот оно, имя папки как имя пакета. Для импорта модуля или его содержимого из этого пакета требуется, чтобы имя пакета было префиксом к имени модуля, соединенным точкой.

Примечание: технически вы можете импортировать и пакет, но, увы, это не импортирует модули внутри пакета в локальное пространство имен.

10. Что такое глобальные, защищенные и приватные атрибуты в Python?

Глобальные переменные- это общедоступные переменные, которые определены в глобальной области видимости. Чтобы использовать переменную в глобальной области видимости внутри функции, мы используем ключевое слово **global**.

Защищенные атрибуты (Protected attributes) - это атрибуты, определенные с префиксом подчеркивания к их идентификатору, например, `_sara`. К ним все еще можно получить доступ и изменить их извне класса, в котором они определены, но ответственный разработчик должен воздержаться от этого.

Приватные атрибуты (Private attributes) - это атрибуты с двойным подчеркиванием в префиксе к их идентификатору, например __ansh. Они не могут быть доступны или изменены извне напрямую, и при такой попытке будет выдана ошибка `AttributeError`.

11. Как используется self в Python?

`self` используется для представления экземпляра класса. С помощью этого ключевого слова вы можете получить доступ к атрибутам и методам класса в python.

`self` связывает атрибуты с заданными аргументами. `self` используется в разных местах и часто считается ключевым словом. Но в отличие от C++, `self` не является ключевым словом в Python.

12. Что такое init?

init - это метод-конструктор в Python, который автоматически вызывается для выделения памяти при создании нового объекта/экземпляра. Все классы имеют метод **init**, связанный с ними. Он помогает отличить методы и атрибуты класса от локальных переменных.

```
    # class definition
    class Student:
       def __init__(self, fname, lname, age, section):
            self.firstname = fname
            self.lastname = lname
            self.age = age
            self.section = section
    # creating a new object
    stu1 = Student("Sara", "Ansh", 22, "A2")
```

13. Что такое break, continue и pass в Python?

Оператор **break** немедленно завершает цикл, а управление переходит к оператору после тела цикла.

Оператор **continue** завершает текущую итерацию оператора, пропускает остальной код в текущей итерации, а управление переходит к следующей итерации цикла.

Ключевое слово **pass** в Python обычно используется для заполнения пустых блоков и аналогично пустому утверждению, представленному точкой с запятой в таких языках, как Java, C++, Javascript и т.д.

14. Что такое модульные тесты в Python?

Юнит-тесты - это структура модульного тестирования в Python.

Юнит-тестирование означает тестирование различных компонентов программного обеспечения по отдельности. Можете ли вы подумать о том, почему модульное тестирование важно? Представьте себе сценарий: вы создаете программное обеспечение, которое использует три компонента, а именно A, B и C. Теперь предположим, что в какой-то момент ваше программное обеспечение ломается. Как вы определите, какой компонент был ответственен за поломку программы? Может быть, это компонент A вышел из строя, который, в свою очередь, вышел из строя компонент B, что и привело к поломке программного обеспечения. Таких комбинаций может быть множество.

Вот почему необходимо должным образом протестировать каждый компонент, чтобы знать, какой компонент может быть ответственен за сбой программного обеспечения.

15. Что такое docstring в Python?

docstring - это многострочная строка, используемая для документирования определенного участка кода.

В `docstring` должно быть описано, что делает функция или метод.

16. Что такое срез в Python?

Как следует из названия, “срез” - это взятие частей.

Синтаксис следующий [start : stop : step].

- start - начальный индекс, с которого производится нарезка списка или кортежа
- stop - конечный индекс или место нарезки.
- step - количество шагов для перехода.

Значение по умолчанию для start - 0, stop - количество элементов, step - 1.

Срезы можно выполнять для строк, массивов, списков и кортежей.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[1 : : 2])  #output : [2, 4, 6, 8, 10]
```

17. Объясните, как можно сделать Python Script исполняемым на Unix?

Файл сценария должен начинаться с #!/usr/bin/env python

18. В чем разница между массивами и списками в Python?

Массивы в python могут содержать элементы только одного типа данных, т.е. тип данных массива должен быть однородным. Это тонкая обертка вокруг массивов языка C, и они потребляют гораздо меньше памяти, чем списки.

Списки в python могут содержать элементы разных типов данных, то есть тип данных списков может быть неоднородным. Их недостатком является потребление большого объема памяти.

```
import array
a = array.array('i', [1, 2, 3])
for i in a:
    print(i, end=' ')    #OUTPUT: 1 2 3
a = array.array('i', [1, 2, 'string'])    #OUTPUT: TypeError: an integer is required (got type str)

    a = [1, 2, 'string']
for i in a:
   print(i, end=' ')    #OUTPUT: 1 2 string
```

Middle / Senior

19. Как осуществляется управление памятью в Python?

Управление памятью в Python осуществляется менеджером памяти Python. Память, выделяемая менеджером, представляет собой частное пространство кучи, предназначенное для Python. Все объекты Python хранятся в этой куче, и, будучи частной, она недоступна программисту. Тем не менее, Python предоставляет некоторые основные функции API для работы с частным пространством кучи.

Кроме того, Python имеет встроенную сборку мусора для утилизации неиспользуемой памяти для частного пространства кучи.

20. Что такое пространства имен Python? Зачем они используются?

Пространство имен в Python гарантирует, что имена объектов в программе уникальны и могут использоваться без каких-либо конфликтов. Python реализует эти пространства имен в виде словарей, в которых “имя как ключ” сопоставлено с соответствующим “объектом как значением”. Это позволяет нескольким пространствам имен использовать одно и то же имя и сопоставлять его с отдельным объектом. Ниже приведены несколько примеров пространств имен:

Локальное пространство имен включает локальные имена внутри функции. Пространство имен временно создается для вызова функции и очищается после возвращения функции.

Глобальное пространство имен включает имена из различных импортированных пакетов/модулей, которые используются в текущем проекте. Это пространство имен создается при импорте пакета в скрипт и сохраняется до выполнения скрипта.

Встроенное пространство имен включает встроенные функции ядра Python и встроенные имена для различных типов исключений.

Жизненный цикл пространства имен зависит от области видимости объектов, с которыми они сопоставлены. Если область видимости объекта заканчивается, жизненный цикл этого пространства имен завершается. Следовательно, невозможно получить доступ к объектам внутреннего пространства имен из внешнего пространства имен.

21. Что такое разрешение области видимости в Python?

Иногда объекты в одной области видимости имеют одинаковые имена, но функционируют по-разному. В таких случаях разрешение области видимости в Python происходит автоматически. Вот несколько примеров такого поведения:

Модули Python ‘math’ и ‘cmath’ имеют множество функций, общих для обоих - log10(), acos(), exp() и т.д. Чтобы разрешить эту двусмысленность, необходимо снабдить их префиксом соответствующего модуля, например, math.exp() и cmath.exp().

Рассмотрим приведенный ниже код, объект temp был инициализирован на 10 глобально и затем на 20 при вызове функции. Однако вызов функции не изменил значение temp глобально. Здесь мы можем заметить, что Python проводит четкую границу между глобальными и локальными переменными, рассматривая их пространства имен как отдельные личности.

```
temp = 10   # global-scope variable
def func():
     temp = 20   # local-scope variable
     print(temp)
print(temp)   # output => 10
func()    # output => 20
print(temp)   # output => 10
```

Это поведение может быть переопределено с помощью ключевого слова global внутри функции, как показано в следующем примере:

```
temp = 10   # global-scope variable
def func():
     global temp
     temp = 20   # local-scope variable
     print(temp)
print(temp)   # output => 10
func()    # output => 20
print(temp)   # output => 20
```

22. Что такое декораторы в Python?

Декораторы в Python - это, по сути, функции, которые добавляют функциональность к существующей функции в Python без изменения структуры самой функции. В Python они обозначаются @decorator_name и вызываются по принципу “снизу вверх”. Например:

```
    # decorator function to convert to lowercase
    def lowercase_decorator(function):
       def wrapper():
            func = function()
            string_lowercase = func.lower()
            return string_lowercase
       return wrapper

    # decorator function to split words
    def splitter_decorator(function):
       def wrapper():
            func = function()
            string_split = func.split()
            return string_split
       return wrapper

    @splitter_decorator # this is executed next
    @lowercase_decorator # this is executed first
    def hello():
        return 'Hello World'
    hello()   # output => [ 'hello' , 'world' ]
```

Прелесть декораторов заключается в том, что помимо добавления функциональности к выходу метода, они могут даже принимать аргументы для функций и дополнительно модифицировать эти аргументы перед передачей в саму функцию. Внутренняя вложенная функция, то есть функция-“обертка”, играет здесь важную роль. Она реализуется для обеспечения инкапсуляции и, таким образом, скрывает себя от глобальной области видимости.

```
    # decorator function to capitalize names
    def names_decorator(function):
       def wrapper(arg1, arg2):
            arg1 = arg1.capitalize()
            arg2 = arg2.capitalize()
            string_hello = function(arg1, arg2)
            return string_hello
       return wrapper

    @names_decorator
    def say_hello(name1, name2):
       return 'Hello ' + name1 + '! Hello ' + name2 + '!'
    say_hello('sara', 'ansh')   # output => 'Hello Sara! Hello Ansh!'
```

23. Что такое comprehensions Dict и List?

Python comprehensions, как и декораторы, - это синтаксический сахар, который помогает строить измененные и отфильтрованные списки, словари или множества из заданного списка, словаря или множества. Использование понятий позволяет сэкономить много времени и сэкономить код, который мог бы быть значительно более многословным (содержать больше строк кода). Давайте рассмотрим несколько примеров, в которых понимания могут быть действительно полезны:

Словарные (Dict) comprehension используют фигурные скобки и позволяют создавать новые словари на основе уже существующих.

```
    new_dict = {key: value for key, value in old_dict.items() if value > 2}
```

Выполнение математических операций над всем списком

```
    my_list = [2, 3, 5, 7, 11]
    squared_list = [x**2 for x in my_list]    # list comprehension
    # output => [4 , 9 , 25 , 49 , 121]
    squared_dict = {x:x**2 for x in my_list}    # dict comprehension
```

```
# output => {11: 121, 2: 4 , 3: 9 , 5: 25 , 7: 49}
```

Выполнение операций условной фильтрации для всего списка

```
my_list = [2, 3, 5, 7, 11]
squared_list = [x**2 for x in my_list if x%2 != 0]    # list comprehension
# output => [9 , 25 , 49 , 121]
squared_dict = {x:x**2 for x in my_list if x%2 != 0}    # dict comprehension
# output => {11: 121, 3: 9 , 5: 25 , 7: 49}
```

Объединение нескольких списков в один

```
a = [1, 2, 3]
b = [7, 8, 9]
[(x + y) for (x,y) in zip(a,b)]  # parallel iterators
# output => [8, 10, 12]
[(x,y) for x in a for y in b]    # nested iterators
# output => [(1, 7), (1, 8), (1, 9), (2, 7), (2, 8), (2, 9), (3, 7), (3, 8), (3, 9)]
```

Преобразование многомерного массива в одномерный

Аналогичный подход вложенных итераторов (как описано выше) может быть применен для сглаживания многомерного списка или работы с его внутренними элементами.

```
my_list = [[10,20,30],[40,50,60],[70,80,90]]
flattened = [x for temp in my_list for x in temp]
# output => [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Примечание: генератор списков имеет тот же эффект, что и метод map в других языках. Они используют математическую нотацию построителя множеств, а не функции `map` и `filter` в Python.

24. Что такое лямбда в Python? Почему это используется?

Лямбда - это анонимная функция в Python, которая может принимать любое количество аргументов, но может иметь только одно выражение. Обычно она используется в ситуациях, когда требуется анонимная функция на короткий промежуток времени. Лямбда-функции можно использовать одним из двух способов:

Присвоение лямбда-функций переменной:

```
mul = lambda a, b : a * b
print(mul(2, 5))    # output => 10
```

Обертывание лямбда-функций внутри другой функции:

```
def myWrapper(n):
    return lambda a : a * n
mulFive = myWrapper(5)
print(mulFive(2))    # output => 10
```

25. Как скопировать объект в Python?

В Python оператор присваивания (=) не копирует объекты. Вместо этого он создает связь(ссылку) между существующим объектом и именем целевой переменной. Чтобы создать копии объекта в Python, необходимо использовать модуль `copy`. Существует два способа создания копий для данного объекта с помощью модуля `copy`.

Shallow Copy - это побитовая копия объекта. Созданный скопированный объект имеет точную копию значений в исходном объекте. Если одно из значений является ссылкой на другие объекты, копируются только адреса ссылок на них.

Глубокое копирование рекурсивно копирует все значения от исходного объекта к целевому, т.е. дублирует даже объекты, на которые ссылается исходный объект.

```
from copy import copy, deepcopy
list_1 = [1, 2, [3, 5], 4]
## shallow copy
list_2 = copy(list_1)
list_2[3] = 7
list_2[2].append(6)
list_2    # output => [1, 2, [3, 5, 6], 7]
```

```
    list_1    # output => [1, 2, [3, 5, 6], 4]
    ## deep copy
    list_3 = deepcopy(list_1)
    list_3[3] = 8
    list_3[2].append(7)
    list_3    # output => [1, 2, [3, 5, 6, 7], 8]
    list_1    # output => [1, 2, [3, 5, 6], 4]
```

26. В чем разница между xrange и range в Python?

`xrange()` и `range()` довольно похожи по функциональности. Они оба генерируют последовательность целых чисел, с той лишь разницей, что `range()` возвращает список Python, тогда как `xrange()` возвращает объект `xrange`.

В отличие от `range()`, `xrange()` не генерирует статический список, а создает значение на ходу. Эта техника обычно используется с генератором объектного типа и называется "**yielding**".

Выдача очень важна в приложениях, где память ограничена. Создание статического списка, как в `range()`, может привести к ошибке памяти в таких условиях, в то время как `xrange()` может справиться с этим оптимально, используя только достаточное количество памяти для генератора (значительно меньше по сравнению с другими).

```
    for i in xrange(10):    # numbers from o to 9
        print i       # output => 0 1 2 3 4 5 6 7 8 9
    for i in xrange(1,10):    # numbers from 1 to 9
        print i       # output => 1 2 3 4 5 6 7 8 9
    for i in xrange(1, 10, 2):    # skip by two for next
        print i       # output => 1 3 5 7 9
```

Примечание: `xrange` была устаревшей начиная с Python 3.x. Теперь `range` делает то же самое, что делала `xrange` в Python 2.x, поскольку в Python 2.x было гораздо лучше использовать `xrange()`, чем оригинальную функцию `range()`.

27. Что такое pickling и unpickling?

Библиотека Python предлагает **функцию сериализации** из коробки. Сериализация объекта означает преобразовани его в формат, который можно хранить, чтобы впоследствии можно было десериализовать его и получить исходный объект.

Pickling - это название процесса сериализации в Python. Любой объект в Python может быть сериализован в поток байтов и выгружен в память в виде файла. Процесс `pickling` компактен, но объекты `pickle` могут быть сжаты еще больше. `pickle` отслеживает объекты, которые он сериализовал, и сериализация переносима между версиями.

Для этого процесса используется функция `pickle.dump()`.

Распаковка (**Unpickling**) - является полной противоположностью `pickle`. Он десериализует поток байтов для воссоздания объектов, хранящихся в файле, и загружает объект в память.

Для этого используется функция `pickle.load()`.

28. Что такое генераторы в Python?

Генераторы - это функции, которые возвращают итерируемую коллекцию элементов, по одному за раз, заданным образом. Генераторы, в общем случае, используются для создания итераторов с другим подходом. Они используют ключевое слово `yield`, а не `return` для возврата объекта генератора.

Построим генератор для чисел Фибоначчи:

```
    ## generate fibonacci numbers upto n
    def fib(n):
        p, q = 0, 1
        while(p < n):
            yield p
            p, q = q, p + q

    x = fib(10)    # create generator object

    ## iterating using __next__(), for Python2, use next()
    x.__next__()    # output => 0
```

```
    x.__next__()    # output => 1
    x.__next__()    # output => 1
    x.__next__()    # output => 2
    x.__next__()    # output => 3
    x.__next__()    # output => 5
    x.__next__()    # output => 8
    x.__next__()    # error

    ## iterating using loop
    for i in fib(10):
        print(i)    # output => 0 1 1 2 3 5 8
```