BaseFunctionsPy_RU

Вступление

В этом гайде вы сможете получить основные зания по Python. Причиной написания гайда была сложность и запутаность которую я заметил в другий гайдах и руководствах. Безусловно я не эксперт в Python, но при этом имею какой-то опыт и думаю, что могу помочь этими знаниями другим :)

Полезные ссылки

Так как этот гайд достаточно субъективный я не уверен что захвачу все темы и не уверен что у нас не появяться вопросы после прочтения, по-этому в этом блоке прикреплю хорошие ссылки который можно использовать в таких случаях.

В первую очередь вы можете написать мне на почты и я попробую вам помочь:

boyarkin.gleb@gmail.com

source.boar@gmail.com

Так же почты подублированы у меня в профиле.

Кроме меня вы можете воспользоваться:

https://pythonworld.ru/samouchitel-python

https://metanit.com/python/tutorial/

Эти источники гарантирую что являються проверенными, сам по ним учился, но как по мне многие вещи там расписаны слишком сложными словами.

Коротко про Python

Python — достаточно старый, но популярный в наше время язык. На нем очень легко писать мелкие программы, по-этому он в основном используеться в маленьких проектах или для мелких автоматизаций.

Синтаксис

Синстаксис в Python не являеться C-подобным и различаеться достаточно сильно, но от этого не становится сложнее.

Если проходиться по основным правилам синтаксиса то:

- символа для окончания строки нету
- `:` – переход на уровень ниже, открытие блока кода
- `snake_case` – стиль именования переменных и функций
- `PascalCase` – стиль именования классов
- `#` – символ для комментария одной строки
- `''' '''` – символы для комментирования блока кода

Подробнее про правила написания кода написано в официальном постановлении `PEP8` **:**
https://peps.python.org/pep-0008/

Работа с консолью

`input('''текст сообщения''')` – ввод с консоли

`print('''текст сообщения''')` – вывод в консоль

Переменные

Как я думаю вам известно, или станет известо сейчас, по факту весь язык программирования держиться на переменных и операциях над ними.

В Python переменная — это указатель на область или ячейку памяти в которой хранится какое то зничение. Факт про память пока можно просто положить в голову, к нему мы прийдем потом.

Для объявления не нужны ключевые слова.

```
a1 = 1
a2 = True
a3 = "hello"
```

При именовании переменных, как указано было выше, используеться стиль snake_case.

Что ж в него входит?

snake_case:

- все буквы нижнего регистра
- слова разделяються `_`

Так же если конкретно про Python то в нем название переменной может начинать с символа `_` или буквы любого регистра, последующие символы уже могут быть такие же плюс цифры(0-9). Так же стоит уточнить что Python чувствителен к регистру, то есть в нем `a` и `A` это разные имена.

Явно локальность или глобальность переменной в Python не указывается. По умолчанию они локальны для блока кода в котором они объявляються. Для использования глобальных или переменных внешнего блока кода используеться `global` и `nonlocal`

global

Это ключевое слово используется для обозначения того, что переменная в этом блоке кода будет браться из глобального уровня, уровня программы.

Пример:

```
a1 = 1

def func1():
    print(a1) # a1 не объявлена
```

```
def func2():
    global a1
    print(a1) # 1

def func3():
    def fucn4():
        global a1
        print(a1) # 1
```

nonlocal

Это ключевое слово используется для обозначения того, что переменная в этом блоке кода будет браться из блока кода на уровень выше.

Пример:

```
a1 = 1

def func1():
    a1 = 2
    def fucn2():
        global a1
        print(a1) # 1

    def func3():
        nonlocal a1
        print(a1) # 2
```

Типы данных

Тип данных — типы того, какого типа значение может хранить переменная.

Базовые типы данных в Python:

- `bool` – логический тип данных, хранит 2 вида значений: `True` или `False`
- `int` – целочисленный тип данных, хранит целые числа, без ограничения по значению, оно зависит от мощности компьютера
- `float` – дробный тип данных, хранит дробные числа
- `complex` – комплексный тип данных, хранит комплексные числа
- `str` – строчный тип данных, хранит строку текста почти неограниченого размера

Типизация

Типизация Python – это динамическая, сильная, неявная.

А теперь попробуем понять:

- Динамическая – значит что переменная не привязана к единому типу при объявлении, переменная может хранить разные типы по ходу использования программы

- Сильная – значит что при проделование действий с переменной одного типа действий другого типа будет вызывать ошибку. *Например:* `1+"2"` будет вызывать ошибку
- Неявная – значит что при объявлении не нужно "явно" указывать тип переменной. *Например как это в C++ с явной типизацией:* `string a = "a"` , когда в Python хватит `a = "a"` , без указание типа

Статья, где эта тема раскрыта глубже: https://tproger.ru/explain/tipizacija-jazykov-programmirovanija-razbiraemsja-v-osnovah/

Что же нам дают эти знания?

Из типизации Python мы можем извлечь те факты, что:

- Нам не нужно следить за типами на уровне программы
- Нам не нужно объявлять переменную указывая её тип
- Мы можем для одной переменной присваивать значения разных типов

Рассмотрим конкретно разные типы

bool

Это логический тип данных, который используется в местах, где нужно в переменной хранить 2 варианта значения: ДА (`True`) или НЕТ (`False`)

Объявления переменных этого типа могут выглядить так:

```
t = True
f = False
```

Используются они, в основном, в условных конструкциях, которые мы рассмотрим позже

int

Это целочисленный тип данных, хранит в себе целые числа.

Объявления переменных этого типа могут выглядить так:

```
p = 1
n = -2
o = 0
```

Для конвертации других типов в `int` можно использовать функцию `int()`

float

Тоже самое что и `int` только хранит в себе дробные числа

Для конвертации других типов в `float` можно использовать функцию `float()`

str

Строчный тип данных, хранит в себе текст.

Объявления переменных этого типа могут выглядить так:

```
let txt1 = "Hello"
let txt2 = 'World'
```

Разницы между кавычками нету , главное что б они открывали и закрывали строку одновременно.

Для конвертации других типов в `str` можно использовать функцию `str()`

Так же для строк существует очень большое количество методов, перечень которых и инструкции к ним можно найти по ссылке: https://metanit.com/python/tutorial/5.2.php

Операторы

Операторы присваивания

`=` – присваивание

`+=` – присваивание со сложением

`-=` – присваивание с вычитанием

`*=` – присваивание с умножением

`/=` – присваивание с делением

`//=` – присваивание с делением нацело

`%=` – присваивание с делением с остатком

`**=` – присваивание с возведением в степень

`<<=` – присваивание с левым сдвигом

`>>=` – присваивание с правым сдвигом

`&=` – присваивание с побитовым `И`

`|=` – присваивание с побитовым `ИЛИ`

`^=` – присваивание с побитовым исключающим `ИЛИ`

Логические операторы

`==` – равенство

`!=` – неравенство

`>` – больше

`>=` – больше или равно

`<` – меньше

`<=` – меньше или равно

`and` – логическое `И`, то есть оба условия в конструкции `(условие 1)&&(условие 2)` должны быть `True`

`or` – логическое `ИЛИ`, то есть хотя б одно условие в конструкции `(условие 1)||(условие 2)` должно быть `True`

Арифметические операторы

`+` – плюс

`-` – минус

`*` – умножение

`/` – деление

`//` – деление нацело

`%` – деление с остатком

`**` – возведение в степень

Побитовые операторы

`&` – побитовое `И`

`|` – побитовое `ИЛИ`

`^` – побитовое исключающее `ИЛИ`

`~` – побитовое `НЕ`

`<<` – сдвиг влево

`>>` – сдвиг вправо

Мелкая важная заметка

`pass` — ключевое слово которое означает что блок пустой. Нужно использовать, если вам нужно оставить блок кода пустым, потому-что без этого слова будет ошибка.

Условные конструкции

Переменные это конечно хорошо, но надо ж как-то с ними работать и вот условные конструкции это один из способов прописывания логики программы.

if...else

Самая известная и часто используемая конструкция.

В `if` передаёться условие, если условие истинное то выполняеться то, что в блоке `if` в ином случае то, что в блоке `else`

Синтаксис выглядит примерно так:

```
if '''условие: или логическое выражение или переменная типа bool''':
    # действие если условие == True
else:
    # действие если условие == False
```

Так же в эту конструкцию можно добавить `elif`

Пример применения:

```
if '''условие 1: или логическое выражение или переменная типа bool''':
    # действие если условие 1 == True
elif '''условие 2: или логическое выражение или переменная типа bool''':
    # действие если условие 1 == False, но условие 2 == True
...
else:
    # действие если условие 1 == False и условие 2 == False
```

Таких конструкций `elif` можно добавлять любое количество раз

try...except...finally

Условный блок, при котором условием выполнения будет наличие или отсутвие ошибки в блоке кода `try`

Синтаксис выглядит примерно так:

```
try:
    # блок кода который будет воспроизводить на поиск ошибки
except:
    # действие если ошибка
finally:
    # действие после выполнения блока, которое не будет воспроизводить на поиск
ошибки
```

Циклы

while

Цикл при котором действие выполняется пока условие истинно

Синтаксис выглядит примерно так:

```
while '''условие: или логическое выражение или переменная типа bool''':
    # действие если условие True
```

for...in

Вид цикла с действием которое выполняеться до начала обхода, условием выполнения и действием которое выполняеться в конце каждого прохода

Синтаксис выглядит примерно так:

```
for '''переменная''' in '''набор значений''':
    # действие пока переменная в наборе значений
```

range

`range()` – функция для создание объекта значений

Синтаксис выглядит примерно так:

```
range('''с''', '''до''', '''шаг, по умолчанию 1''')
```

Ключевые слова `break` и `continue`

`break` используеться для досрочного выхода из цикла, то есть завершение работы цикла вне зависимости от условия

Пример использования:

```
for i in range(0, 10):
    if i % 3 == 0: # если число делится на 3 без остатка
        break
    print(i)
```

То есть цикл выполнялся пока не наткнулся на первое число которое делится на 3

`continue` используеться для досрочного прекращения прохода цикла и перехода на следующий проход, то есть используется когда вам надо что б все после его вызова не выполнялось в это проходе, а выполнялось сразу сделующий проход

```
for i in range(0, 10):
    if i % 3 == 0: # если число делится на 3 без остатка
        continue
    print(i)
```

То есть для чисел которые деляться на 3 не выполнялся `print()`, а сразу выполнялся следующий прогон цикла

Практические задания на данных знаниях

Для лучшего закрепления предлогаю выполнить несколько простых задач, которые проверят, насколько вы поняли этот уровень материала, все знания которые могут быть нужны для выполнения задач есть выше.

Все решения задач будут в папке "Задачи" и будут пронумерованы, но не рекомендую вам ими пользваться пока у вас самих не получится

1. Допишите в начало кода такие объявления переменных или ключевые слова в начале функций, что б остальной код заработал как написано в коментарии. Не все места для написания кода обязаны быть заполнены

```
# запишите ваши объявления сюда
print(a) # 1
def func1():
    # или сюда
    print(a) # 1

def func2():
    a = 3
    def func3():
        # или сюда
        print(a) # 3
    func3()

func1()
func2()
```

(Должны произойти все выводы в консоль)

2. Напишите цикл который выведет в консоль все числа от 1 до 100 включительно которые деляться на 5 или на 7 (`n % /*число*/ == 0` – условие делимости `n` на какое либо число)
3. Напишите блок кода который будет в случае выполнения действия `a = 1 / 0` выводилась фраза "На ноль делить нельзя" и после этого выводилась фраза "Ошибки обработаны"

Фукнции

Функции — ещё один важдный инструмент из программирования. Если простыми словами это объединение какого либо участка кода для его дальнейшего вызова.

Каждая функция в любом языке программирования имеет:

- Имя
- Список аргументов (параметров) – переменные, значения которых задаёться при вызове функции. Объявляються они в `()` через запятую. Так же при вызове их значения указываются при вызове так же в `()`. В блоке кода нашей функции они отыгрывают роль обычных переменных с блочной областью видимости, только не нуждаються в объявлении, так как объявляються при вызове функции
- Блок кода который будет выполняться при вызове. Записываеться после `:`

Синтаксис выглядит примерно так:

```
def '''имя функции'''('''аргумент1''', '''аргумент2'''...'''аргументN'''):
    # блок исполняемого кода
```

`def` – ключевое слово которое означает объявление функции

Для вызова функции достаточно написать её имя и `()` в которых можно перечислить аргументы если они были объявлены

Пример функции:

```
def f(a1, a2, a3):
    print(a1)
    print(a2)
    print(a3)

f(1, 2, 3)
```

Фукнции и процедуры

Если отходить от темы Python, где синтаксической разницы между этими понятиями по факту нету, то это очень важный вопрос.

В чем же разница?

Функция — блок кода, который результате своего выполнения возвращет какое либо занчение.

Процедура — функция, которая не возвращает значения

За возвращение значения в Python отвечает ключевое слово `return`

Давайте расмотрим разницу на примерах:

```
def func(a1, a2, a3, a4): # функция
    print(a1)
    print(a2)
    print(a3)
    return a4

def proc(b1, b2, b3, b4): # процедура
    print(b1)
    print(b2)
    print(b3)

let a = func(1, 2, 3, 4) # == 4(a4)
let b = proc(1, 2, 3, 4) # == None
```

Что же случилось?

При объявлении **функции** мы использовали `return` по-этому вызов функции можно будет присвоить переменной и он будет иметь значение которое мы укзали в `return`

А при объявленнии **процедуры** `return` мы не использовали и-за чего наш вызов не имеет значения, то есть `None`

lambda функции и присваивание функции переменной

В Python есть возможность присовить переменной функцию, а если быть точнее ссылку на функцию. Таким образом её можно "переименовать"

То есть это может выглядить например так:

```
def func(a1, a2, a3):
    print(a1)
```

```
    print(a2)
    print(a3)

proc = func
```

Такой способ не является не особо коректным с логической точки зрения, но при этом возможным.

Если же хочеться сократить код и не ломать логику можно использовать **lamda функции**

Синтаксис выглядит примерно так:

```
let '''имя функции''' = lambda '''аргумент1''', '''аргумент2'''...'''аргументN''': # блок испольняемого кода
```

Есть несколько важных аспектов:

- Всё пишется в одну строку, перенос невозможен
- Блок исполняемого кода автоматически возвращается

Пример:

```
sq = lambda x: x ** 2 # функция возведения в квадрат
```

Упорядочиные типы данных

Списки

В Python массивы представлены **списком**. На самом деле в остальных языках: **массив** — это тип данных с фиксированым размером, а **список** — с возможностью изменения размера. Но некотороые языки пренебрегают этим фактом, но не Python.

Для объявления можно просто перечислить элементы в `[]` и присвоить переменной

```
arr = [1, 2, 3]
```

Для лучшего понимания воспользуемся картинкой выше. Как видно на ней массив(список) — это грубо говоря набор "ячеек" со значениями к которым параллельно прикриплены индексы по которым можно получить значение которому они соответствуют.

!!!Индесация в языках программирования начинается с 0!!!

Для ображения по индексу достаточно имени списка в `[]` указать индекс. Для обращения с конца можно использовать отрицательные индексы(-1 – последний, -2 – предпоследний...)

```
arr = [1, 2, 3]
a = arr[0] # 1
arr[-1] = 4 # arr == [1, 2, 4]
```

При ображении к несуществующему индексу в Python будет выдаваться ошибка.

```
arr = [1, 2, 3]
a = arr[4] # Ошибка
```

Основные методы и функции массивов

- `len()` – возвращает длину списка

  ```
  arr = [1, 2, 3]
  l = len(arr) # 3
  ```

- `.append(item)` – добавляет элемент `item` в конец списка

  ```
  arr = [1, 2, 3]
  arr.append(4) # arr == [1, 2, 3, 4]
  ```

- `.insert(index, item)` – добавляет элемент `item` в список по индексу `index`

  ```
  arr = [1, 2, 3]
  arr.insert(1, 4) # arr == [1, 4, 2, 3]
  ```

- `.extend(items)` – добавляет набор элементов `items` в конец списка

  ```
  arr = [1, 2, 3]
  arr.append([4, 5, 6]) # arr == [1, 2, 3, 4, 5, 6]
  ```

- `.clear()` – удаление всех элементов из списка

  ```
  arr = [1, 2, 3]
  arr.clear() # arr == []
  ```

- `.index(item)` – возвращает индекс элемента `item`. Если элемент не найден, генерирует исключение `ValueError`

  ```
  arr = [1, 2, 3]
  i = arr.index(2) # 1
  j = arr.index(4) # ValueError
  ```

- `.find(item)` – возвращает индекс элемента `item`. Если элемент не найден возвращаеться -1

```
arr = [1, 2, 3]
i = arr.find(2) # 1
j = arr.find(4) # -1
```

- `.pop(index)` – удаляет и возвращает элемент по индексу `index`. Если индекс не передан, то просто удаляет последний элемент.

```
arr = [1, 2, 3]
i = arr.pop(1) # arr == [1, 3]
```

- `.count(item)` – возвращает количество вхождений элемента `item` в список

```
arr = [1, 2, 2, 3]
c = arr.count(2) # 2
```

- `.reverse()` – расставляет все элементы в списке в обратном порядке

```
arr = [1, 2, 3]
i = arr.reverse() # arr == [3, 2, 1]
```

- `'''разделитель'''.join()` – объеденяет массив в строку и вставляет между элементами разделитель который мы укажем в аргументе

```
arr = [1, 2, 3]
s = "-".join(arr) # "1-2-3"
```

- `min()` – возвращает наименьший элемент списка

```
arr = [1, 2, 3]
a = min(arr) # 1
```

- `max()` – возвращает наибольший элемент списка

```
arr = [1, 2, 3]
a = max(arr) # 3
```

Про остальные можно узнать по ссылке: https://metanit.com/python/tutorial/3.1.php

Срез

Для стреза списка в Python используеться такая колнструкция:

```
arr['''с''':'''до''']
```

Если один из параметров не указан он автоматически становиться краем списка с его стороны.

Так же конструкция `[::-1]` будет ивертировать список.

Перебор массива

for...in...range

```
for i in range(0, len(arr)):
    # элемент массива arr[i]
```

for...in

```
for a in arr:
    # элемент массива a
```

Словари

Иногда возникают ситуации, когда надо что б индексами были не цифры, а что-то своё. Для такого были созданы **словари**.

Словари – это массив с самоуказаными индексами.

Синтаксис объявления

```
dic = {
    '''название индекса 1''': '''занчение 1''',
    '''название индекса 2''': '''занчение 2''',
    ...
    '''название индекса N''': '''занчение N'''}
```

Обращение и присваивание такое же, только индексы не обязательно цифры:

```
dic = {"name": "Sam", "age": 17}
d = dic["name"] # "Sam"
dic["age"] = 18
```

Кортежи

Кортеж — последовательность элементов, похож на список, за исключением того, что не может изменяться.

Объявляеться как массив, за исключением того что не в `[]`, а в `()`. Но при обращении к элементам так же `[]`.

Пример использования:

```
tup = ("Tom", 17)
t = tup[0] # "Tom"
tup[1] = 18 # невозможно
```

Работа со строками

После того, как мы ознакомились с массивами можно поговорить про строки.

По факту: **строка** – это массив символов. К её элементам можно обращаться как в массиве, но нельзя изменять как элементы в массиве.

```
s = "abc"
c = s[0] # "a"
s[0] = "d" # невозможно
```

Так же в Python строки можно прибавлять и умножать

```
s1 = "hello"
s2 = "world"
sp = s1 + " " + s2 # "hello world"
sm = s1 * 3 # "hellohellohello"
```

Методы и функции строк

- `len()` – возвращает длину строки

  ```
  s = "abc"
  l = len(s) # 3
  ```

- `.lower()` – возвращает строку в нижнем регистре

  ```
  s1 = "AbC"
  s2 = s1.lower() # "abc"
  ```

- `.upper()` – возвращает строку в верхнем регистре

  ```
  s1 = "AbC"
  s2 = s1.upper() # "ABC"
  ```

- `.split('''разделитель''')` – разбивает строку на массив строк по разделителю

  ```
  s = "a-b-c"
  arr = s.split("-") # ["a", "b", "c"]
  ```

- `ord()` – получение кода элемента

  ```
  c = ord("a") # 97
  ```

- `chr()` – получение элемента по коду

  ```
  c = chr(97) # "a"
  ```

Практические задания на данных знаниях

4. Придумайте как перевернуть строку в обратном порядке
5. Найдите способ строку `"abcdef"` превратить в `"cdefgh"`
6. Напишите функцию которая будет возвращать строку созданую по правилу `snake_case` (разделитель должен передаваться вторым аргументом)

   Пример:

   `hello world` -> `hello_world`
7. Создайте функцию для заполнения массива словарей, которые содержат информацию про человека

   Структура:

   ```
   people = [] # основной массив
   '''структура словаря
   {
       "firstName": "",
       "secondName": "",
       "age": "",
       "country": "",
       "city": ""
   }
   '''
   ```
8. *Придумайте как можно отсортировать массив

"*" – сложная задача

Всё!

В принцепе это всё! Я дал вам всё что хотел. Надеюсь этот гайд помог вам.

Так же у меня есть гайд посложнее с более сложными темами, но он больше просто сборник решений, но кому интересно можете почитать: https://github.com/s0urce18/AdditionalFunctions

Всем спасибо! Надеюсь ещё увидимся :)