BaseFunctionsPy_UA

Вступ

У цьому гайді ви зможете отримати базові знання по Python. Причиною написання гайда була важкість і заплутаність яку я помітив в інших гайдах. Безумовно я не експерт в Python, але при цьому маю якийсь досвіт та думаю, що можу допомогти цими знаннями іншим :)

Корисні посилання

Так як цей гайд достатньо суб'єктивний я не впевнений, що захвачу усі теми і не впевнений, що у вас не з'являться питання після читання, через що в цьому блокі прикріплено гарні посилання, які можна використовувати в таких випадках.

В першу чергу ви можете написати мені на пошту і я спробую вам допомогти:

boyarkin.gleb@gmail.com

source.boar@gmail.com

Також пошти продубльовані у мене в профілі.

Окрім мене ви можете використати:

https://pythonworld.ru/samouchitel-python

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

Ці джерела гарантую, що є перевіренними, сам по ним навчався, але, як на мене, багато речей там розписані надто складними словами.

Стисло про Python

Python — достатньо стара, але популярна у наші часи мова. На ній дуже легко писати невеликі програми, через це вона в основному використовується в маленьких проектах або для маленьких автоматизацій.

Синтаксис

Синтасис в Python не є C-подібним і відрізняється достатньо сильно, але від цього не стає важче.

Якщо проходитися по осноним праивлам синтаксису то:

- символа для закінчення рядка немає
- `:` – перехід на рівень нижче, відкриття блока коду
- `snake_case` – стиль іменування змінних и функцій
- `PascalCase` – стиль іменування класів
- `#` – символ для коментування одного рядка
- `''' '''` – символи для коментування блока коду

Докладніше про правила написания коду написано в офіційному постановленні `PEP8` :
https://peps.python.org/pep-0008/

Робота с консоллю

`input('''текст сообщения''')` – ввід з консолі

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

Змінні

Як я думаю вам відомо, або стане відомо зараз, по факту уся мова програмування тримається на змінних та оперціях над ними. У Python змінна — це вказівник на область або комірку пам'яті у котрій зберігається будь-яке значення. Факт про пам'ять поки можна просто покласти в голосу, до нього ми прийдемо потім.

Для оголошення не потрібні ключові слова.

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

При іменування змінних, як було вказано вище, використовується стиль snake_case.

Что ж в нього входить?

snake_case:

- усі літери нижнього регістру
- слова розділяються `_`

Так само якщо конкретно про Python то в ньому назва змінної може починатися з символа `_` або літери будь-якого регістру, наступні символи всеж можуть буди такі ж самі плюс цифри(0-9). Так само є сенс зазначити, що Python чутливий к регістру, тобто у ньому `a` і `A` це різні імена.

Явно локальність або глобальність змінної не вказується. За замовченням вони локальні для блока коду в котрому вони оголошені. Для використання глобальних або змінних зовнішнього блока коду використовується `global` і `nonlocal`

global

Це ключове слово використовується для позначення того, що змінна в цьому блоці коду буде братися з глобального рівня, рівня програми.

Приклад:

```
a1 = 1

def func1():
    print(a1) # a1 не оголошена

def func2():
    global a1
```

```
    print(a1) # 1

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

nonlocal

Це ключове слово використовується для позначення того, що змінна в цьому блоці коду буде братися з блока коду на рівень вище.

Приклад:

```
a1 = 1

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

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

Типи даних

Тип даних — тип того, якого типа значення може зерігати змінна.

Базові типи даних в Python:

- `bool` – логічний тип даних, зберігає 2 вида значень: `True` або `False`
- `int` – цілочисленний тип даних, зберігає цілі числа, без обмеження по значенню, воно залежить від потужності комп'ютера
- `float` – дробовий тип даних, зберігає дробові числа
- `complex` – комплексний тип даних, зберігає комплексні числа
- `str` – рядковий тип даних, зберігає рядок текстау майже необмеженого разміру

Типізація

Типізація Python – это динамічна, сильна, неявна.

А теперь попробуем понять:

- Динамічна – означає, що змінна не прив'язана до єдиного типу при оголошенні, змінна може зберігати різні типи під час використання програми
- Сильна – це означає, що при виконанні дій зі змінною одного типу дій іншого типу буде викликати помилку. *Наприклад:* `1+"2"` буде викликати помилку

- Неявна - означає, що при оголошенні не потрібно "явно" вказувати тип змінної. *Наприклад як це в C++ з явною типізацією:* `string a = "a"`, коли в Python вистачить `a = "a"`, без зазначення типу

Стаття, де цю тему розкрито глибше: https://tproger.ru/explain/tipizacija-jazykov-programmirovanija-razbiraemsja-v-osnovah/

Що ж нам дають ці знання?

З типизації Python ми можемо вилучити ті факти, що:

- Нам не потрібно стежити за типами на рівні програми
- Нам не потрібно оголошувати змінну, вказуючи її тип
- Ми можемо на одну змінну надавати значення різних типів

Розглянемо конкретно різні типи

bool

Це логічний тип даних, який використовується в місцях, де потрібно в змінній зберігати 2 варіанти значення: ТАК (`True`) або НІ (`False`)

Оголошення змінних цього типу можуть виглядати так:

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

Використовується вони в основному в умовних конструкціях, які ми розглянемо пізніше.

int

Це цілочисельний тип даних, що зберігає в собі цілі числа.

Оголошення змінних цього типу можуть виглядати так:

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

Для конвертації інших типів в `int` можна використовувати функцію `int()`

float

Те саме що і `int` тільки зберігає в собі дробові числа

Для конвертації інших типів `float` можна використовувати функцію `float()`

str

Рядковий тип даних, зберігає у собі текст.

Оголошення змінних цього типу можуть виглядати так:

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

Різниці між лапками немає, головне щоб вони відкривали і закривали рядок одночасно.

Для конвертації інших типів в `str` можна використовувати функцію `str()`

Також для рядків існує дуже велика кількість методів, перелік яких та інструкції до них можна знайти за посиланням: https://metanit.com/python/tutorial/5.2.php

Оператори

Оператори присвоєння

`=` – присвоєння

`+=` – присвоєння з додаваням

`-=` – присвоєння з відніманням

`*=` – присвоєння з множенням

`/=` – присвоєння з діленням

`//=` – присвоєння з діленням націло

`%=` – присвоєння з діленням с залишком

`**=` – присвоєння зі зведенням у ступінь

`<<=` – присвоєння с лівим сдвигом

`>>=` – присвоєння с правим сдвигом

`&=` – присвоєння с побітовим `I`

`|=` – присвоєння с побітовим `АБО`

`^=` – присвоєння с побітовим виключаючим `АБО`

Логические операторы

`==` – рівність

`!=` – нерівність

`>` – більше

`>=` – більше або дорівнює

`<` – менше

`<=` – менше або дорівнює

`and` – логічне `I`, тобто обидві умови в конструкції `(умова 1)&&(умова 2)` повинні бути `True`

`or` – логічне `АБО` ,тобто хоча б одна умовв в конструкції `(умова 1)||(умова 2)` повинна бути `True`

Арифметические операторы

`+` – плюс

`-` – мінус

`*` – множення

`/` – ділення

`//` – ділення націло

`%` – ділення с остатком

`**` – зведення у ступінь

Побітові оператори

`&` – побітовое `І`

`|` – побітовое `АБО`

`^` – побітовое виключаюче `АБО`

`~` – побітовое `НІ`

`<<` – сдвиг вліво

`>>` – сдвиг вправо

Маленька важлива примітка

`pass` — ключове слово, яке означає, що блок порожній. Потрібно використовувати, якщо вам потрібно залишити блок коду порожнім, бо без цього слова буде помилка.

Умовні конструкції

Змінні це звичайно добре, але ж треба якось з ними працювати і ось умовні конструкції це один із способів прописування логіки програми.

if...else

Найвідоміша і найчастіше використовувана конструкція.

У `if` передається умова, якщо умова істинна то виконується те, що в блоці `if` в іншому випадку те, що в блоці `else`

Синтаксис виглядає приблизно так:

```
if '''умова: або логічне вираження чи змінна типу bool''':
    # дія якщо умова == True
else:
    # дія якщо умова == False
```

Також в цю конструкцію можна додати `elif`

Приклад застосування:

```
if '''умова 1: або логічне вираження чи змінна типу bool''':
    # дія якщо умова 1 == True
elif '''умова 2: або логічне вираження чи змінна типу bool''':
    # дія якщо умова 1 == False, але умова 2 == True
...
else:
    # дія якщо умова 1 == False і умова 2 == False
```

Таких конструкцій `elif` можна додавати будь-яку кількість разів

try...except...finally

Умовний блок, за якого умовою виконання буде наявність або відсутність помилки в блоці коду `try`

Синтаксис виглядає приблизно так:

```
try:
    # блок коду який відтворюватиме на пошук помилки
except:
    # действие если ошибка
finally:
    # дію після виконання блоку, яке не відтворюватиме на пошук помилки
```

Цикли

while

Цикл при якому дія виконується доки умова істинна

Синтаксис виглядає приблизно так:

```
while '''умова: або логічне вираження чи змінна типу bool''':
    # дія якщо умова True
```

for...in

Вид циклу з дією, яка виконується до початку обходу, умовою виконання та дією, яка виконується в кінці кожного проходу

Синтаксис виглядає приблизно так:

```
for '''змінна''' in '''набір значень''':
    # дія поки що змінна в наборі значень
```

range

`range()` – функція створення об'єкта значень

Синтаксис виглядає приблизно так:

```
range('''з''', '''по''', '''крок, за замовчуванням 1''')
```

Ключові слова `break` та `continue`

`break` використовується для дострокового виходу з циклу, тобто завершення роботи циклу незалежно від умови

Приклад використання:

```
for i in range(0, 10):
    if i % 3 == 0: # якщо число ділиться на 3 без залишку
        break
    print(i)
```

Тобто цикл виконувався доки не наткнувся на перше число, яке ділиться на 3

`continue` використовується для дострокового припинення проходу циклу і переходу на наступний прохід, тобто використовується коли вам треба, щоб все після його виклику не виконувалося в цьому проході, а виконувалося відразу зробляє прохід

```
for i in range(0, 10):
    if i % 3 == 0: # якщо число ділиться на 3 без залишку
        continue
    print(i)
```

Тобто для чисел які ділитися на 3 не виконувався `print()`, а одразу виконувався наступний прогін циклу

Практичні завдання на даних знаннях

Для кращого закріплення пропоную виконати кілька простих завдань, які перевірять, наскільки ви зрозуміли цей рівень матеріалу, всі знання, які можуть бути потрібні для виконання завдань, є вище.

Всі розв'язки задач будуть у папці "Завдання" і будуть пронумеровані, але не рекомендую вам ними користуватися доки у вас самих не вийде

1. Допишіть на початок коду такі оголошення змінних або ключові слова на початку функцій, щоб інший код запрацював як написано в коментарі. Не всі місця для написання коду мають бути заповнені

```
# запишіть ваші оголошення сюди
print(a) # 1
def func1():
    # або сюди
    print(a) # 1

def func2():
    a = 3
    def func3():
```

```
        # або сюди
        print(a) # 3
    func3()

func1()
func2()
```

(Мають відбутися всі виводи в консоль)

2. Напишіть цикл який виведе в консоль всі числа від 1 до 100 включно які ділитися на 5 або на 7 (`n % /*число*/ == 0` – умова подільності `n` на якесь число)
3. Напишіть блок коду який буде у разі виконання дії `a = 1/0` виводилася фраза "На нуль ділити не можна" і після цього виводилася фраза "Помилки оброблені"

Фукнції

Функції – ще один важливий інструмент із програмування. Якщо простими словами це об'єднання якоїсь ділянки коду для його подальшого виклику.

Кожна функція у будь-якій мові програмування має:

- Ім'я
- Список аргументів (параметрів) – змінні, значення яких задається під час виклику функції. Оголошуються вони в `()` через кому. Так само при виклику їх значення вказуються при виклику так само в `()` . У блоці коду нашої функції вони відіграють роль звичайних змінних з блоковою областю видимості, тільки не потребують оголошення, тому що оголошуються при виклику функції
- Блок коду, який буде виконуватися при викликі. Записується після `:`

Синтаксис виглядає приблизно так:

```
def '''ім'я функції'''('''аргумент1''', '''аргумент2'''...'''аргументN'''):
    # блок коду, що виконується
```

`def` – ключове слово, яке означає оголошення функції

Для виклику функції достатньо написати її ім'я та `()` у яких можна перерахувати аргументи якщо вони були оголошені

Приклад функции:

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

f(1, 2, 3)
```

Фукнції та процедури

Якщо відходити від теми Python, де синтаксичної різниці між цими поняттями за фактом немає, це дуже важливе питання.

У чому ж різниця?

Функція — блок коду, який в результаті свого виконання поверне якесь закінчення.

Процедура — функція, яка не повертає значення

За повернення значення Python відповідає ключове слово `return`

Давайте розглянемо різницю на прикладах:

```
def func(a1, a2, a3, a4): # функція
    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
```

Що ж сталося?

При оголошенні **функції** ми використовували `return` тому виклик функції можна буде привласнити змінної і він буде мати значення яке ми укзали в `return`

А при оголошенні **процедури** `return` ми не використовували і через що наш виклик не має значення, тобто `None`

lambda функції та присвоєння функції змінної

У Python є можливість додати змінну функцію, а якщо бути точніше посилання на функцію. Таким чином її можна "перейменувати"

Тобто це може виглядати наприклад так:

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

proc = func
```

Такий спосіб не є особливо коректним з логічної точки зору, але при цьому можливим.

Якщо ж хочеться скоротити код і не ламати логіку, можна використовувати **lamda функції**

Синтаксис виглядає приблизно так:

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

Є кілька важливих аспектів:

- Все пишеться в один рядок, перенесення неможливе
- Блок виконуваного коду автоматично повертається

Приклад:

```
sq = lambda x: x ** 2 # функція зведення в квадрат
```

Упорядковані типи даних

Списки

У Python масиви представлені **списком**. Насправді в інших мовах: **масив** — тип даних з фіксованим розміром, а **список** — з можливістю зміни розміру. Але деякі мови нехтують цим фактом, але не Python.

Для оголошення можна просто перерахувати елементи в `[]` та присвоїти змінній

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

Для кращого розуміння скористаємося вище картинкою. Як видно на ній масив (список) - це грубо кажучи набір "осередків" зі значеннями до яких паралельно прикріплені індекси, за якими можна отримати значення, якому вони відповідають.

!!!Індесація в мовах програмування починається з 0!!!

Для відображення за індексом достатньо імені списку в [[]' вказати індекс. Для поводження з кінця можна використовувати негативні індекси (-1 – останній, -2 – передостанній...)

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

При зверненні до неіснуючого індексу Python буде видаватися помилка.

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

Основні методи та функції масивів

- `len()` – повертає довжину списку

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

- `.append(item)` – додає елемент `item` до кінця списку

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

- `.insert(index, item)` – додає елемент `item` до списку за індексом `index`

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

- `.extend(items)` – додає набір елементів `items` до кінця списку

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

- `.clear()` – видалення всіх елементів зі списку

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

- `.index(item)` – повертає індекс елемента `item`. Якщо елемент не знайдено, генерує виняток `ValueError`

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

- `.find(item)` – повертає індекс елемента `item`. Якщо елемент не знайдено повертається -1

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

- `.pop(index)` – видаляє та повертає елемент за індексом `index`. Якщо індекс не передано, просто видаляє останній елемент.

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

- `.count(item)` – повертає кількість входжень елементу `item` до списку

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

- `.reverse()` – розставляє всі елементи у списку у зворотному порядку

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

- ```"'розділювач"'.join()` – об'єднує масив у рядок і вставляє між елементами роздільник який ми вкажемо в аргументі

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

- `min()` – повертає найменший елемент списку

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

- `max()` – повертає найбільший елемент списку

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

Про решту можна дізнатися за посиланням: https://metanit.com/python/tutorial/3.1.php

Зріз

Для стрезу списку в Python використовується така конструкція:

```
arr['''с''':'''до''']
```

Якщо один із параметрів не вказано, він автоматично ставати краєм списку з його боку.

Так само конструкція `[::-1]` івертуватиме список.

Перебір масиву

for...in...range

```
for i in range(0, len(arr)):
    # елемент масиву arr[i]
```

for...in

```
for a in arr:
    # елемент масиву a
```

Словники

Іноді виникають ситуації, коли треба, щоб індексами були не цифри, а щось своє. Для такого були створені "словники".

Словники – це масив із самозазначеними індексами.

Синтаксис оголошення

```
dic = {
     '''назва індексу 1''': '''значення 1''',
     ''' Назва індексу 2''': '''значення 2''',
     ...
     '''назва індексу N''': '''значення N'''}
```

Звернення та присвоєння таке ж, тільки індекси не обов'язково цифри:

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

Кортежі

Кортеж — послідовність елементів, схожих на список, за винятком того, що не може змінюватися.

Оголошується як масив, за винятком того що не в `[]`, а в `()`. Але при зверненні до елементів так само `[]`.

Приклад використання:

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

Робота з рядками

Після того, як ми ознайомилися з масивами, можна поговорити про рядки.

По факту: ** рядок ** – це масив символів. До її елементів можна звертатися як і масиві, але не можна змінювати як елементи в масиві.

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

Так само в Python рядки можна додавати та множити

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

Методи та функції рядків

- `len()` – повертає довжину рядка

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

- `.lower()` – повертає рядок у нижньому регістрі

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

- `.upper()` – повертає рядок у верхньому регістрі

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

- `.split('''розділювач''')` – розбиває рядок на масив рядків по роздільнику

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

- `ord()` – отримання коду елемента

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

- `chr()` – отримання елемента за кодом

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

Практичні завдання на даних знаннях

4. Придумайте як перевернути рядок у зворотному порядку
5. Знайдіть спосіб рядок `abcdef` перетворити на `cdefgh`
6. Напишіть функцію яка повертатиме рядок створений за правилом `snake_case` (розділювач повинен передаватися другим аргументом)

   Приклад:

   `hello world` -> `hello_world`

7. Створіть функцію для заповнення масиву словників, які містять інформацію про людину

   Структура:

```
people = [] # основний масив
'''структура словника
{
    "firstName": "",
    "secondName": "",
    "age": "",
    "country": "",
    "city": ""
}
'''
```

8. *Придумайте як можна відсортувати масив

"*" - складна задача

Усе!

Здаэться це все! Я дав вам все, що хотів. Сподіваюся: цей гайд допоміг вам.

Так само у мене є гайд складніший з складнішими темами, але він більш просто збірник рішень, але кому цікаво можете почитати: https://github.com/s0urce18/AdditionalFunctions

Всім дякую! Сподіваюсь ще побачимось :)