Шпаргалка: функции и ошибки

Кратко

Что	Как	О чём помнить

Объявить функцию	`def <имя функции>():`
	`  <тело функции>`
	Пример
	`def hello():`
	`  print('Привет, тестировщик!')`
		Тело нужно отбивать 4 пробелами

Вызвать функцию	`<имя функции>():` **Пример** `hello()` напечатает «Привет, тестировщик!»	Функцию можно вызывать сколько угодно раз

Объявить и вызвать функцию с параметрами	`def hello(name):`
	`  print('Привет, ',name)`
	`hello('Елисей')`
	Напечатает: Привет, Елисей!	

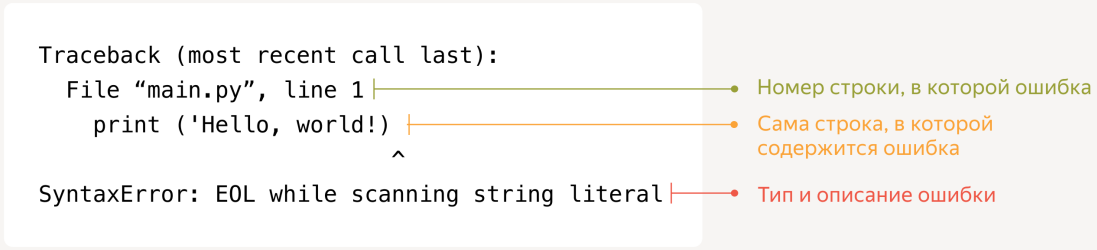
Попросить функцию вернуть значение	`def calc(a,b):`
	`  result = a * b`
	`  return result`	

Задать значение параметра по умолчанию	`def print_occupation(name='любой человек', occupation='умница'):` `  print(name + ' — ' + occupation)`	

Указать именованный аргумент	`def print_occupation(name='любой человек', occupation='умница'):`
	`  print(name + ' — ' + occupation)`
	`print_occupation(name='Ван Гог')`	

Импортировать библиотеку	`import math` `square_root = math.sqrt(16)`	

Трейсбэк — это сообщение об ошибке



Что за ошибка	Когда появляется	О чём помнить
Неизвестное имя, `NameError`	Ошибки `name is not defined` появляются, когда какую-то переменную не объявили, но пытаются использовать.	
Ошибки синтаксиса, `SyntaxError`	Появляются, когда нарушили какое-то из правил синтаксиса Python. Например, пропущено двоеточие или чего-то не хватает.	Если код состоит из несколько строк, ошибка с незакрытой скобкой показывается не на той строчке, где забыли закрыть скобку, а на следующей.
Ошибка в типах данных	Проводят операции с несовместимыми типами	

Подробно, с примерами

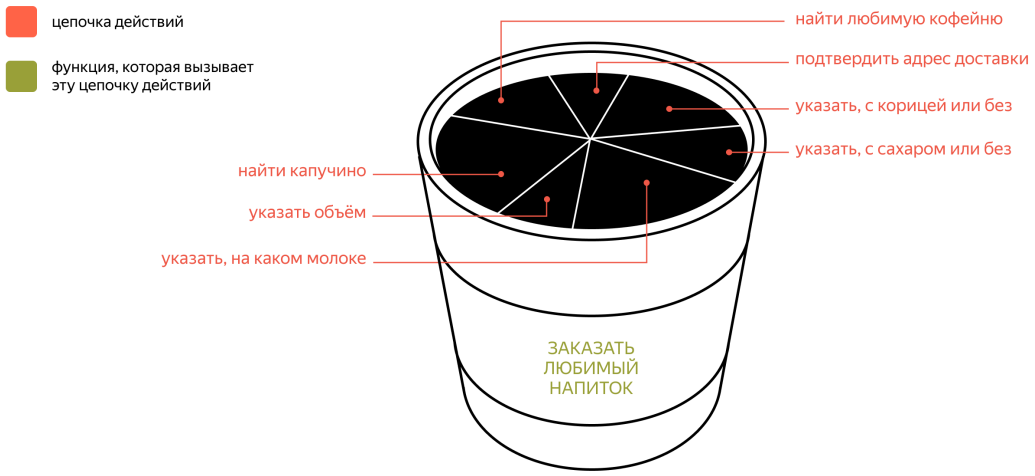
Что такое функция

Функция — это блок кода, который выполняет какую-то задачу. Скажем, печатает текст на экран или проверяет почту. У этого блока есть имя. Например, `print()` или `checkmail()`.

Представь, что ты часто заказываешь кофе в сервисе доставки. Чтобы сделать заказ, нужно открыть приложение доставки, найти кофейню, выбрать напиток, указать дополнительные пожелания и адрес.

И тут в приложении появляется кнопка «Заказать любимый кофе». Целая цепочка действий заменяется одним тапом. Так и работает функция: когда ты её

вызываешь, выполняются действия, которые в неё зашиты.



В Python есть много заготовленных, или встроенных функций: `print()`, `str()`, `int()`, `float()`, `len()`. Но можно создавать и собственные.

Зачем нужны функции

Иногда в разных местах программы нужно выполнять одинаковые действия. Например, программа проверяет, не пришли ли деньги на счёт. Как только деньги пришли, она присылает уведомление «Баланс пополнен, время для роскошной жизни!».

Программа проверяет баланс каждую минуту. Получается много одинаковых строчек кода:

```
balance_increase = 0
if balance_increase > 0:
    print('Баланс пополнен, время для роскошной жизни!')
else:
    print('Деньги не пришли :(')
# Тут код, отсчитывающий минуту

# И снова проверим
if balance_increase > 0:
    print('Баланс пополнен, время для роскошной жизни!')
else:
    print('Деньги не пришли :(')
```

```
# Тут код, отсчитывающий минуту

# И снова проверим

if balance_increase > 0:
    print('Баланс пополнен, время для роскошной жизни!')
else:
    print('Деньги не пришли :(')
# Тут код, отсчитывающий минуту

# За это время поступили деньги:
balance_increase = 5000
# И снова проверка
if balance_increase > 0:
    print('Баланс пополнен, время для роскошной жизни!')
else:
    print('Деньги не пришли :(')
```

Смотрится громоздко и неаккуратно, читать код сложно. Можно написать функцию `check_balance()` — «проверить баланс». В неё будет упакован такой набор команд:

```
if balance_increase > 0:
    print('Баланс пополнен, время для роскошной жизни!')
else:
    print('Деньги не пришли :(')
```

Теперь в программе можно не писать четыре строчки кода каждый раз: они заменяются одним вызовом функции.

```
balance_increase = 0

# Создадим функцию check_balance
# В эту функцию поместим код проверки количества писем.
def check_balance():
    if balance_increase > 0:
        print('Баланс пополнен, время для роскошной жизни!')
    else:
        print('Деньги не пришли :(')


# Тут код, отсчитывающий минуту

# Проверим почту - вызовем функцию check_mail():
# запустим код, который хранится в функции.
check_balance()

# Тут код, отсчитывающий минуту
```

```
# И снова вызовем функцию:
check_balance()

# Тут код, отсчитывающий минуту
# За это время пришло письмо:
balance_increase = 5000

# И снова вызовем функцию:
check_balance()
```

Код стал гораздо короче, а работает точно так же.

Как объявить функцию

Объявить функцию — значит сказать программе: «Здесь мы создаём новую функцию». Вот как это сделать:

- Написать ключевое слово `def`.
- Придумать **имя** функции — например, `hello`. Для имени функции действуют те же правила, что и для переменных.
- Поставить круглые скобки и двоеточие.

Например, так объявили функцию `hello()`:

```
def hello():
```

Пока что эта функция ничего не делает — она просто существует. Чтобы она что-то умела, нужно написать ещё блок кода. Он выполнится, если функцию вызовут. Это **тело функции.**

Например, тело функции `hello()` — команда `print('Привет, тестировщик!')`.

```
# Объявили функцию
def hello():
    # Тело функции
    print('Привет, тестировщик!')
```

📎 Тело функции **нужно отделить четырьмя пробелами** от начала строки. Это строгое техническое условие: без отступов ничего не сработает или будет работать неправильно.

Как вызвать функцию

Итак, есть функция, которая приветствует тестировщика:

```
def hello():
    print('Привет, тестировщик!')
```

Но если выполнить этот код — ничего не произойдёт. Функция выполнится, когда где-то в коде её **вызовут.**

Вызов функции — это команда «функция, делай свою работу!». Пока функция не вызвана, она не выполняется: просто лежит и ждёт своего часа.

Чтобы вызвать функцию, нужно написать её название и поставить круглые скобки.

```
def hello():
    print('Привет, тестировщик!')

hello()
```

Функцию можно вызывать сколько угодно раз. Например, так:

```
def hello():
    print('Привет, тестировщик!')

hello()
hello()
hello()
```

Эта программа три раза выведет «Привет, тестировщик!».

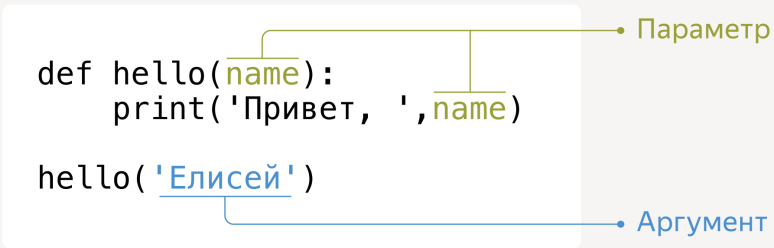
Что такое параметры

Параметры функции — переменные, которые ты напишешь в скобках. Например, `def hello(name)`. Здесь `name` — это параметр. Теперь каждый раз нужно будет

указывать, какое имя печатать.

Аргументы

Когда ты вызовешь функцию, передашь туда конкретное значение. Например, `hello ('Елисей')`. Его называют **аргументом функции**. По сути это просто конкретное значение параметра.



Значение «Елисей» будет подставлено в строку, которую напечатает функция `print()`:

```
# Объявление функции с параметром name
def hello(name):
    # Параметр name можно обрабатывать в теле функции:
    print('Привет, ',name)

# Вызов функции с аргументом 'Елисей'
hello('Елисей')
# Напечатает: Привет, Елисей!
```

Несколько параметров

У функции может быть и несколько параметров: они перечисляются через запятую. Нужно передать аргументы в том порядке, в котором они записаны в аргументах.

Например, функцию объявили так: `def hello(name, bonus)`. Первый аргумент — `name`, второй — `bonus`. Значит, в функции так и напишешь параметры: сначала имя, потом бонус. Например, `'Дарт Вейдер', 'печеньки'`.

```
# Теперь у функции hello() два параметра: name и bonus
def hello(name, bonus):
    # Оба параметра применим в теле функции:
    print(name + ', приветствую тебя! Бери ' + bonus)


hello('Дарт Вейдер', 'печеньки')

# И ещё раз вызовем, с другими аргументами:
hello('Винни Пух', 'мёд')
```

Код, который печатает приветствие, обернули в функцию. В конце кода — несколько вызовов этой функции с разными аргументами.

```
# Код функции say_hello()
def say_hello(language):
    if language == 'Русский':
        print('Здравствуйте')
    elif language == 'Испанский':
        print('¡Hola!')
    elif language == 'Французский':
        print('Bonjour!')
    elif language == 'Английский':
        print('Hello')

# Дальше код написан без отступов: этот код уже вне функции.

# Несколько раз вызовем функцию say_hello() с разными аргументами:
say_hello('Английский')  # Вызов функции say_hello() с аргументом 'Английский'
say_hello('Испанский')  # Вызов функции с аргументом 'Испанский'
```

Как вернуть значение

Обычно задача функции не сводится к печати. Функция вычисляет значение и отдаёт его назад в код. Дальше оно понадобится в программе. Например, она посчитала площадь, а дальше это значение пригодится, чтобы вычислить объём.

Вот представь производство автомобилей. В большой пресс подают лист металла, и получается капот машины. Но это ещё не финальный продукт: капот передают дальше по конвейеру. Его надо покрасить и прикрепить к корпусу автомобиля.

Функция должна работать похожим образом. В неё поступает «заготовка» — аргументы. А функция выдаёт какой-то результат: **возвращает** его в ту часть кода, откуда её вызвали.

Программа применит результат труда функции для каких-то дальнейших вычислений.

Нужно:

- Поставить ключевое слово `return`.
- Указать, какое значение нужно вернуть.

Например, нужно вернуть результат умножения `a * b`:

```
def calc(a,b):
    result = a * b
    return result # Функция возвращает значение переменной result
```

Ещё пример. Функция `calc_square()` получает на вход два аргумента — длины сторон прямоугольника. Она вычисляет и возвращает его площадь.

```
# Функция для вычисления площади прямоугольника;
# от англ. calculate, «вычислять»
def calc_square(side_a, side_b):
    # Вычисляем площадь и присваиваем результат переменной result
    result = side_a * side_b
    # Функция возвращает значение переменной result:
    return result

# Вызываем функцию calc_square() с аргументами 16 и 9.
# Значение, которое вернёт функция, будет присвоено переменной rectangle_area
rectangle_area = calc_square(16, 9)

print(rectangle_area)
```

Обрати внимание: сама функция `calc_square()` ничего не печатает. В ней нет вызова `print()`.

Функция `calc_square()` была вызвана в строке `rectangle_area = calc_square(16, 9)`. Она вычислила результат и вернула его в переменную `rectangle_area`. Это значение напечаталось уже вне функции, в основном коде программы.

Напишем программу, которая вычислит и напечатает суммарную площадь трёх жилых комнат. Их размеры 3x5, 4x6 и 3x6 метров.

Пригодится та же функция `calc_square()`.

Здесь не обойтись без нескольких действий:

1. Получить площадь каждой комнаты: поочерёдно передать в функцию `calc_square()` размеры каждой комнаты.
2. Сохранить результаты вызовов функции в отдельные переменные: в них будут храниться площади трёх комнат.
3. Просуммировать площади.
4. Напечатать результат.

```
def calc_square(side_a, side_b):
    result = side_a * side_b
    # Функция возвращает значение переменной result:
    return result

# Вызовем функцию calc_square(), передадим в неё размеры первой комнаты;
# функция вычислит площадь комнаты и вернёт её;
# вернувшееся значение присвоим переменной room1.
room1 = calc_square(3, 5)

# Вычислим и сохраним в переменную room2 площадь второй комнаты:
room2 = calc_square(4, 6)

# Площадь третьей комнаты сохраним в room3:
room3 = calc_square(3, 6)

# Теперь можно суммировать полученные значения и напечатать результат:

rooms_sum = room1 + room2 + room3
print('Суммарная площадь комнат равна', rooms_sum, 'кв.м')
```

Значение по умолчанию

Это то, что передастся параметру, если функция не получит какой-то аргумент.

Например, у функции есть два аргумента — `print_occupation(name, occupation)`. Можно заранее сказать программе:

— Если не передадут значение для `name`, пиши «любой человек». Если не скажут значение для `occupation`, напиши «умница».

Значение по умолчанию нужно присвоить функции, когда её объявляешь:

```
# присвоили значения по умолчанию в скобках
def print_occupation(name='любой человек', occupation='умница'):
    print(name + ' — ' + occupation)
```

Функция отработает без ошибок, даже если что-то забудут:

```
def print_occupation(name='любой человек', occupation='умница'):
    print(name + ' — ' + occupation)

# Передаём только один аргумент вместо двух
print_occupation('Иосиф Бродский')
```

Позиционные аргументы

Когда ты вызываешь функцию `print_occupation()`, значения передаются в параметры в том порядке, как и написаны. Первый аргумент — в первый параметр, второй аргумент — во второй. Это называется **позиционные аргументы**.

Это просто и удобно, но может вызвать путаницу.

В функцию забыли передать имя. Теперь род деятельности — первый аргумент. Он попадёт в параметр `name`:

```
def print_occupation(name='любой человек', occupation='умница'):
    print(name + ' — ' + occupation)

# Передаём только один аргумент вместо двух
print_occupation('поэт')
```

Получилось не то. Python не понял, какой именно аргумент передали.

Чтобы избежать такой неразберихи, при вызове функции можно передавать **именованные аргументы.**

Именованные аргументы

Они явно указывают, какому параметру какой аргумент соответствует. Ты прямо прописываешь, где что. Например, вызываешь функцию так:

`print_occupation(name='Иосиф Бродский')`.

Именованные аргументы можно передавать в любом порядке. Функция увидит имена и разберётся. Например, можно поменять аргументы местами в вызове функции: `print_occupation(occupation = 'поэт', name=' Иосиф Бродский')`.

```
def print_occupation(name='любой человек', occupation='умница'):
    print(name + ' — ' + occupation)

# Передаём именованный параметр:
# явно указываем, что значение 'Ван Гог' предназначено для параметра name
print_occupation(name='Ван Гог')

# Ещё раз вызовем функцию: передадим два именованных параметра,
# но не в том порядке, как они указаны в объявлении функции:
print_occupation(occupation='художник', name='Ван Гог')
```

Импорт библиотек

Библиотека, или модуль — это набор готовых функций. Их объединяют по тематикам. Например, в библиотеке `math` собрали функции для подсчёта математических величин. А в библиотеке `random` — инструменты, которые помогают получать случайные значения.

Чтобы получить доступ к готовым функциям, нужно импортировать библиотеку.

Где импортировать. Библиотеки импортируют в начале программы — с помощью команды `import`. Например, для импорта библиотеки `math` нужно написать `import math`. Всё, теперь можно пользоваться функциями из библиотеки. Например, `sqrt()` извлечёт квадратный корень из числа.

Как вызвать. Чтобы вызвать функцию из импортированной библиотеки, к ней обращаются через имя библиотеки: `имя_библиотеки.имя_функции`. Например, `math.sqrt(16)`.

Вот как это выглядит целиком:

```
# Импорт библиотеки math
import math

# Теперь в программе можно применять любые функции из неё
square_root = math.sqrt(16)
print(square_root)
```

Ещё пример. Импортируем библиотеку `random`. Станут доступны её функции, например:

- Функция `random.randint(min, max)` выберет случайное целое число в диапазоне от числа min до числа max.
- Функция `random.choice(список)` вернёт случайный элемент из списка.
- Функция `random.random()` вернёт случайное дробное число от 0.0 до 1.0.

Программа выбирает, какой фильм посмотреть вечером. В код импортирована не вся библиотека `random`, а лишь одна функция из неё. В таком случае к этой функции обращаются напрямую, без указания имени библиотеки.

Запусти этот код несколько раз и посмотри, что он выведет.

```
from random import choice  # Импорт одной функции из библиотеки

def choose_a_movie(movies):
    # Обращаемся к функции напрямую: choice(), а не random.choice()
    return choice(movies)

print(choose_a_movie(['Дюна', 'Король', 'Не смотрите наверх']))
print(choose_a_movie(['Маленькие женщины', 'Дождливый день в Нью-Йорке', 'Интерстеллар']))
```

Часть функций не нуждается в импорте. Они встроены в Python и доступны без дополнительных манипуляций. Например, `print()`.

Псевдоним для библиотеки

Иногда у библиотек очень длинные имена. Если не хочется загромождать код, можно дать библиотеке короткий «псевдоним». Ты будешь писать его при вызове функции. Например, библиотеку `random` можно сократить до `r`.

Понадобится ключевое слово `as`:

```
import random as r

# Теперь к библиотеке random нужно обращаться только через псевдоним r:
print(r.randint(0, 100)) # Случайное целое число от 0 до 100
```

Библиотека `datetime` и псевдоним

Для работы со временем в Python импортируют библиотеку datetime. В ней есть не только отдельные функции, но и целый новый тип данных. Он тоже называется datetime.

Это ещё один тип данных: как int или string. Он нужен, чтобы хранить информацию о конкретном моменте времени: год, месяц, день, час, минуты, секунды и микросекунды.

Название типа данных точно совпадает с библиотекой. Это не слишком удобно. Чтобы не путаться, подключай библиотеку под именем `dt`:

```
import datetime as dt
```

Вызов функции из функции

Иногда в программе получается много функций. Их приходится вызывать по отдельности.

Допустим, программа вычисляет параметры куба:

```
# Функция для вычисления периметра куба.
def calc_cube_perimeter(side):
    return side * 12

# Вызов функции calc_cube_perimeter() с аргументом 3
one_cube_perimeter = calc_cube_perimeter(3)
full_length = one_cube_perimeter * 8
print('Неоходимый метраж палок для 8 кубов:', full_length)

# Функция для вычисления площади куба.
def calc_cube_area(side):
    one_face = side * side
    cube_area = one_face * 6
    return cube_area

# Вызов функции calc_cube_perimeter() с аргументом 3
one_cube_area = calc_cube_area(3)
full_area = one_cube_area * 8
print('Неоходимая площадь стекла для 8 кубов, кв.м:', full_area)
```

Она работает, но в ней есть недочёты. Работать с кодом неудобно, фрагменты повторяются:

- приходится по отдельности вызывать две функции;

- в функции передаётся один и тот же аргумент — длина ребра куба, `3`.

Можно написать функцию `calc_cube()`. Она сама будет вызывать обе функции.

Тогда можно будет обойтись лишь одним вызовом:

```
# Функция для вычисления периметра куба.
def calc_cube_perimeter(side):
    return side * 12


# Функция для вычисления площади куба.
def calc_cube_area(side):
    one_face = side * side
    cube_area = one_face * 6
    return cube_area


# Основная функция, которая принимает длину ребра куба
def calc_cube(side):
    # Вызываем функцию, рассчитывающую периметр
    # и передаём в неё размер куба
    one_cube_perimeter = calc_cube_perimeter(side)
    full_length = one_cube_perimeter * 8

    # Вызываем функцию, рассчитывающую площадь стекла
    # и передаём в неё размер куба
    one_cube_area = calc_cube_area(side)
    full_area = one_cube_area * 8

    print('Для 8 кубов понадобится палок (м):', full_length, 'и стекла (кв.м):', full_area)


# В результате остался лишь один вызов "основной" функции,
# а она уже вызовет две вспомогательные
calc_cube(3)
```

Иначе говоря, это вызов функции из функции. В программе получается одна точка входа — стартовый код, в который надо передать все данные.

Есть и ещё одно преимущество: размер куба задаётся только в одном месте кода — при вызове `calc_cube()`. Это уменьшает вероятность ошибки и упрощает работу с программой.

Ошибка `RecursionError`

Функция перемножает два числа и печатает результат:

```
def multiplication(multiplier_1, multiplier_2):
    print(multiplier_1 * multiplier_2)

# Дальше начинается код, который расположен вне функции:
# Python понимает это по отсутствию отступов.

# Вызов функции multiplication()
multiplication(7, 6)
```

В этом коде вызов функции на том уровне, что и `def`.

Если отделить вызов четырьмя отступами, программа не сработает. Не будет напечатано ничего.

```
def multiplication(multiplier_1, multiplier_2):
    print(multiplier_1 * multiplier_2)

    multiplication(7, 8)
```

Python решит, что строка `multiplication(7, 8)` — часть тела функции. А значит, нужно дождаться, пока её вызовут.

Но вот проблема: если всё же вызвать эту функцию, она начнёт вызывать сама себя. Программа зациклится и появится ошибка `RecursionError`.

Запусти этот код. Вывод в консоль будет очень длинным. Однако ничего не сломается.

```
def multiplication(multiplier_1, multiplier_2):
    print(multiplier_1 * multiplier_2)

    multiplication(7, 8)

multiplication(5, 6)
```

При первом вызове функция напечатает `30`, а затем закрутится в бесконечном повторении.

Похожая ошибка может появиться, если вызывать функцию из функции из функции.

Например, `func_one()` вызывает `func_two()`, а `func_two()` вызывает `func_one()`. Получается такой круг:

```
def func_one():
    print('Раз')
    func_two()

def func_two():
    print('Два')
    func_one()

func_one() # Вызываем функцию func_one()
```

В программировании это называется **рекурсия**, или же рекурсивная функция. Это такая функция, которая вызывает саму себя.

Рекурсию можно сравнить с матрёшкой. Первая кукла самая большая: за ней идёт точно такая же кукла, но поменьше. И так пока мы не дойдём до последней матрешки, которая и прервёт цикл.

Python умеет обрывать такие циклы сам. Он выбрасывает ошибку `RecursionError`.

Увидишь эту ошибку — проверь:

- не вызывается ли какая-то функция сама из себя;
- всё ли в порядке с отступами — именно они определяют начало и конец тела функции;
- не вызывают ли функции друг друга по кругу.

Сообщения об ошибках

Python постарается помочь найти ошибку: «Я сломался, ошибка на строке 8, там забыли закрыть скобку!». Его сообщения об ошибках нужно уметь читать.

Сообщения об ошибках называются **traceback**: по-русски говорят «трейсбэк».

Из чего он состоит:

- номер строки, в которой ошибка;
- сама проблемная строка;
- тип и описание ошибки.

Есть разные виды ошибок: какие-то возникают из-за проблем в синтаксисе, какие-то — из-за неизвестного значения.

Ошибка «неизвестное имя»

Посмотри на этот код:

```
one_hundred = 100
print(onehundred)
```

Во второй строке опечатка: в имени переменной пропущено подчёркивание:

```
one_hundred = 100
print(onehundred)
```

Python споткнулся о неизвестное имя переменной и выдал сообщение об ошибке. Вот как его читать:

- В какой строчке кода ошибка — `line 2`.
- Тип ошибки — `NameError`.
- Из-за чего произошла ошибка — `name 'onehundred' is not defined`. Это переводится как «никаких сущностей с именем `onehundred` ещё не объявлялось».

📎 Ошибки `name is not defined` появляются, когда какую-то переменную не объявили, но пытаются использовать.

Ошибки синтаксиса

Появляются, когда нарушили какое-то из правил синтаксиса Python. Например, пропущено двоеточие или чего-то не хватает.

Например, так: `print(3 + )`. Здесь в скобках должно быть два слагаемых, но второе забыли. В итоге непонятно, что прибавлять к трём:

```
print(3 + )
```

Как переводится. SyntaxError — «ошибка синтаксиса». Сообщение invalid syntax переводится как «недопустимый синтаксис».

Когда ещё появляется. Другую разновидность ошибки SyntaxError можно получить, если забыть скобку или кавычку. Например, так: `print(3 + 5`:

```
print(3 + 5
```

В тексте `SyntaxError: unexpected EOF while parsing` сокращение **EOF** означает end of file — «конец файла». А само сообщение переводится как «неожиданный конец файла во время разбора кода программы».

Python увидел начало вызова функции, но внезапно код программы закончился.

На какой строчке будет ошибка

Если запустить этот код, ошибку поймают во второй строке, хотя она в первой:

```
print(3 + 5
print('Всё в порядке!')
```

Почему. Python разрешает ставить открывающую и закрывающую скобки на разных строчках. Поэтому он терпеливо дожидается, пока код совершенно утратит смысл. И только тогда бьёт тревогу.

> 📎 Если код состоит из несколько строк, ошибка с незакрытой скобкой показывается не на той строчке, где забыли закрыть скобку, а на следующей.

При любой ошибке программа останавливается. Код, написанный после строки с ошибкой, не выполнится.

Ошибка в типах данных

Представь, что складывают строку и число: `100+'500'`.

Попытка сложить число `100` и строку `'500'` — это проблема:

```
one_hundred = 100
five_hundred = '500'
print('В ответе можно получить 600, а можно и 100500!')
print(one_hundred + five_hundred)
print('Вот мы и получили 600')
```

Ошибка означает, что складывают несовместимые типы:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```