3. Проверка аргументов. Аргументы по умолчанию

✍ В предыдущем юните вы научились создавать собственные функции, запускать их, передавать в них аргументы и получать результат их выполнения.

Теперь вам предстоит научиться проверять аргументы на корректность и сообщать пользователям вашей функции об ошибке, а также задавать аргументы по умолчанию.

ПРОВЕРКА АРГУМЕНТОВ НА КОРРЕКТНОСТЬ

→ Вы уже наверняка сталкивались с различными ошибками, когда выполняли упражнения по *Python*. Это совершенно нормально — ошибки в коде допускают даже очень продвинутые разработчики. Как хорошо, что интерпретатор сообщает пользователю о том, что допущена

ошибка, которую необходимо исправить или учесть обработку появившегося исключения при разработке! Куда хуже было бы, если бы программа продолжала работать, несмотря на необработанные исключения, даже не предупреждая пользователя об этом.

Вам также следует научиться сообщать пользователю или даже себе об ошибке во входных данных в функции.

Как специалисту по *Data Science* вам предстоит обрабатывать базы данных с огромным количеством записей, в которых, к сожалению, могут быть неточности. Если вы обработаете некорректные входные данные, а сам интерпретатор не предупредит вас об ошибке, в итоге вы можете сделать неправильные выводы из расчётов. Чтобы избежать таких ситуаций, в языках программирования, в том числе и в *Python*, присутствуют **исключения**.

В предыдущих модулях вы уже научились обрабатывать исключения с помощью конструкции `try ... except`. Давайте вспомним её.

Вначале посмотрим, какая ошибка возникает, если попробовать получить значение из словаря по ключу, которого в нём нет:

```
# Создадим словарь с оценками студентов
grades = {'Ivanov': 5, 'Smirnov': 3, 'Kuznetsova': 4, 'Tihonova': 5}
# Напечатаем оценку студента, которого нет в словаре
print(grades['Pavlov'])
# Возникнет исключение KeyError: 'Pavlov'
# Оно означает, что переданного ключа (слово "Pavlov")
# нет в словаре
```

Тренировочный Codeboard юнита 3 (Внешний источник)

Открыть Codeboard в новой вкладке

Теперь обработаем это исключение, то есть покажем интерпретатору, что:

- нам известно о том, что такая ошибка может возникнуть;
- мы предусмотрели, что делать в этом случае;
- аварийное прекращение работы скрипта не требуется.

Код будет выглядеть вот так:

```
# Создадим тот же словарь
grades = {'Ivanov': 5, 'Smirnov': 3, 'Kuznetsova': 4, 'Tihonova': 5}
# Только попробуем (try — пробовать) напечатать оценку студента,
# которого нет в словаре
try:
    print(grades['Pavlov'])
# А если возникнет ошибка в ключе (KeyError), скажем,
# что студента нет в словаре
except KeyError:
    print("Student's mark was not found!")
# Будет напечатано:
# Student's mark was not found!
```

После того как вы повторили обработку исключений, настало время **научиться генерировать исключения самостоятельно**. Давайте вспомним функцию из предыдущего юнита, которая считала время в пути для заданного расстояния и скорости:

```
def get_time(distance, speed):
    result = distance / speed
    return result
```

Пока что не имеет значения, какие значения подаются на вход функции: если они являются числами (*int* или *float*), а скорость не равна 0 (иначе возникает деление на ноль, и *Python* выдаёт ошибку *ZeroDivisionError*), то функция сработает и выдаст результат. Однако, скорее всего, вам понятно, что расстояние, как и скорость, не могут быть отрицательными. Иначе расчёт времени в пути противоречит здравому смыслу: что же означает отрицательное время в пути?

Чтобы случайно не передать в функцию get_time некорректные аргументы и не допустить ошибку, добавим их проверку. Если окажется, что аргументы меньше нуля, вызовем *ValueError* с помощью оператора raise.

После встречи с raise, как и в случае с return, интерпретатор прекращает исполнение кода функции.

```
def get_time(distance, speed):
    # Если расстояние или скорость отрицательные, то возвращаем ошибку
    if distance < 0 or speed < 0:
        # Оператор raise возвращает (raise — досл. англ. "поднимать")
        # объект-исключение. В данном случае ValueError (некорректное значение).
        # Дополнительно в скобках после слова ValueError пишем текст сообщения
        # об ошибке, чтобы сразу было понятно, чем вызвана ошибка.
        raise ValueError("Distance or speed cannot be below 0!")
    result = distance / speed
    return result
```

Теперь передадим в функцию заведомо некорректные аргументы и посмотрим, какая ошибка появится:

```
get_time(-25, 100)
# Будет напечатано:
# ValueError: Distance or speed cannot be below 0!

get_time(50, -100)
# Будет напечатано:
# ValueError: Distance or speed cannot be below 0!
```

Как видите, теперь при попытке передать некорректные аргументы вы сразу получаете сообщение об ошибке и понимаете, что входные данные необходимо изменить, чтобы они соответствовали требованиям функции.

ЗАДАНИЕ 3.1 (ВЫПОЛНЯЕТСЯ В CODEBOARD НИЖЕ)

?

Попробуйте добавить в новую функцию get_time из примера выше проверку скорости на равенство нулю. Если скорость равна нулю, верните *ValueError* с сообщением *"Speed cannot be equal to 0!"*.

Вы можете проверить своё решение, вызвав функцию get_time:

```
get_time(100, 0)
# Вы молодец, если на экране появилось:
# ValueError: Speed cannot be equal to 0!
```

При работе с заданием в *Codeboard* **не используйте кириллицу** — писать решение надо полностью **на английском языке**, иначе написанный вами код не запустится!

1. Начните работу традиционно с открытия файла main.py — там вы увидите условие задачи.
2. Впишите требуемый код вместо знаков ??? и нажмите **RUN.**
3. Нажмите **SUBMIT**, чтобы отправить ваш ответ на проверку и получить баллы в случае верного решения.

Задание 3.1 (Внешний источник)

(1.0 из 1.0 баллов)

6.3.1

Root
- src
  - __init__.py
  - **main.py**
- __init__.py

main.py

```
1  def get_time(distance, speed):
2      # Добавьте проверку скорости на равенство 0
3      # Верните ошибку ValueError с текстом
4      # "Speed cannot be equal to 0!"
5
6      if distance < 0 or speed < 0:
7          raise ValueError("Distance or speed cannot
8      result = distance / speed
9      return result
10
11
12 #попытка вызвать функцию с некорректными аргумента
13 try:
14     print(get_time(100, 0))
15 #должна завершиться с ошибкой ValueError, которую
16 except ValueError as e:
17     print(e)
```

```
This will display the output.
```

Input to your program (press Enter to send) >_ Send

User: #anonymous (sign in (/signin?

→ Отлично, теперь вы научились проверять входные данные на корректность и сообщать пользователю об ошибке!

АРГУМЕНТЫ ПО УМОЛЧАНИЮ

Представьте, что вы пришли в ресторан быстрого питания с золотой буквой *M* на логотипе и заказали колу среднего размера. Скорее всего, продавец не спросит вас, добавлять ли в напиток лёд, но при этом в газировке лёд будет.

Допустим, вы хотели напиток безо льда, но не сообщили об этом. К сожалению, в этом случае вы по умолчанию получили колу со льдом. А ведь вам всего-то нужно было сказать, что лёд не требуется!

Точно так же можно создавать функции, в которые можно передавать не все аргументы, а, например, задать часть аргументов (или даже все) **аргументами по умолчанию**, то есть сообщить функции, какие данные использовать, если они не были переданы в функцию извне.

Очень важный пример использования аргументов по умолчанию — **обработка даты**.

По долгу службы вам предстоит обрабатывать даты, записанные в различных форматах, например, чтобы оценить популярность какого-либо сервиса в зависимости от месяца. Чаще всего, находясь в России или Европе, вы будете сталкиваться с привычным форматом вроде 19.05.2021 (то есть *дд.мм.гггг*). Однако в США, например, дату записывают по-другому: 05.19.2021 (то есть *мм.дд.гггг*). Когда вы будете писать функцию, чтобы узнать номер месяца из даты, будет удобно по умолчанию использовать привычный формат, но при этом оставить возможность при необходимости передать в функцию необычный формат даты.

> Аргументы по умолчанию задаются в *Python* очень просто: достаточно после названия переменной сразу же прописать знак `=` (равно) и то значение, которое вы бы хотели присвоить по умолчанию.

Давайте напишем шуточную функцию `get_cola`, которая принимает на вход только один аргумент — `ice`. `ice` будет принимать значения *True* или *False*, причём по умолчанию будет задано *True*. В зависимости от *True* или *False* функция будет печатать *"Кола со льдом готова!"* или *"Кола безо льда готова!"*, соответственно.

Вот её исходный код:

```
# С помощью оператора '=' присвоим
# переменной ice значение True по умолчанию
def get_cola(ice=True):
    if ice == True:
        print("Cola with ice is ready!")
    else:
        print("Cola without ice is ready!")
```

Теперь запустим эту функцию с аргументами *True* и *False*:

```
get_cola(True)
# Будет напечатано:
# Cola with ice is ready!

get_cola(False)
# Будет напечатано:
# Cola without ice is ready!
```

Как видите, в поведении функции пока что нет ничего необычного, однако давайте запустим функцию без аргументов:

```
get_cola()
# Будет напечатано:
# Cola with ice is ready!
```

Функция сработала и без передачи аргументов, и мы по умолчанию получили колу со льдом.

Посмотрим теперь, что бы произошло, если бы мы не указали при создании функции значение по умолчанию для аргумента ice:

```
# Аргументу ice не присваиваем значение по умолчанию:
def get_cola(ice):
    if ice == True:
        print("Cola with ice is ready!")
    else:
        print("Cola without ice is ready!")

get_cola()
# Будет напечатано:
# TypeError: get_cola() missing 1 required positional argument: 'ice'
```

Вы заметили, что возникла ошибка. Она переводится так: *«В функции get_cola() не хватает одного обязательного* ***позиционного аргумента*** *'ice'»*.

> О том, что такое позиционные аргументы, вы узнаете уже в следующем юните.

Итак, использование аргументов по умолчанию позволяет передавать в функцию не все аргументы и не получать из-за этого сообщение об ошибке. Можно сказать, что если у аргумента есть значение по умолчанию, он становится **необязательным**.

Теперь напишем более серьёзную функцию: она будет извлекать корень *n*-ой степени из заданного числа. Назовём функцию root. Она будет принимать на вход value (число, из которого необходимо извлечь корень) и n (степень корня).

Давайте вспомним, что для возведения в степень в *Python* используется оператор **:

```
print(2 ** 4)
# Будет напечатано:
# 16
```

Также **важное утверждение из математики:** чтобы извлечь из числа *x* корень *n*-ой степени, то есть получить такое число, при возведении которого в степень *n* получится число *x*, достаточно возвести число *x* в степень *1/n*.

Давайте это проверим:

```
# Хотим посчитать корень 4-ой степени из 16:
print(16 ** (1/4))
# Будет напечатано:
# 2.0
```

В самом деле — корень четвёртой степени из 16 равен двум. Давайте наконец напишем функцию root:

```
# В функцию должны передаваться 2 значения:
# число и степень корня
def root(value, n):
    # Как мы уже выяснили, чтобы посчитать
    # корень степени n из числа, можно возвести это число
    # в степень 1/n
    result = value ** (1/n)
    return result

# Посчитаем корень 3-ей степени (кубический корень) из 27
print(root(27, 3))
# Будет напечатано:
# 3.0
```

Итак, теперь у нас есть функция, которая может считать корень любой заданной степени из числа. Однако на практике мы сталкиваемся с квадратными корнями гораздо чаще, чем с остальными. Даже когда мы говорим *«корень из 25 равен 5»*, мы имеем в виду квадратный корень, то есть корень степени 2.

ЗАДАНИЕ 3.2 (НА САМОПРОВЕРКУ)

Модифицируйте функцию root так, чтобы передавать степень корня было необязательно. Если степень корня не передана, функция должна возвращать значение квадратного корня.

Проверьте своё решение, вызвав функцию root:

```
print(root(81))
# Должно быть напечатано:
# 9.0
```

Важное замечание. Указывать аргументам значения по умолчанию можно только после того, как в функции перечислены все обязательные аргументы.

Поменяем в модифицированной функции root порядок аргументов:

```
def wrong_root(n=2, value):
    result = value ** (1/n)
    return result
# Возникнет ошибка:
# SyntaxError: non-default argument follows default argument
```

Эта ошибка переводится так: *«Синтаксическая ошибка: аргумент без значения по умолчанию следует после аргумента по умолчанию»*. Чтобы её избежать, необходимо указывать аргументы по умолчанию только после обязательных.

ОПАСНОСТЬ АРГУМЕНТОВ ПО УМОЛЧАНИЮ

Использование аргументов по умолчанию призвано упростить жизнь разработчику. Но, к сожалению, их некорректное использование может приводить к крайне неожиданным последствиям.

Создадим функцию, поведение которой будет неверным (и заодно повторим использование словарей).

Функция `add_mark` будет записывать оценку студента в словарь. Словарь не требуется обязательно передавать в функцию: будем использовать пустой словарь в качестве аргумента по умолчанию (⛔ **сейчас будет пример того, как делать не нужно!**).

Итак, получим функцию:

```
# Функция add_mark (от англ. add mark — "добавить оценку")
# принимает обязательные аргументы name (имя) и
# mark (оценка). Необязательным аргументом является journal
# (журнал оценок), который по умолчанию является пустым словарём.
def add_mark(name, mark, journal={}):
    # Присваиваем имени в журнале переданную оценку
    journal[name] = mark
    return journal
```

Теперь будем экспериментировать с новой функцией. Вначале передадим все три аргумента извне:

```
# Создадим пустой словарь, в который будем
# сохранять оценки группы 1
group1 = {}
# Добавим оценки двух студентов
group1 = add_mark('Ivanov', 5, group1)
group1 = add_mark('Tihonova', 4, group1)
print(group1)
# Будет напечатано:
# {'Ivanov': 5, 'Tihonova': 4}
```

Поведение функции выглядит именно так, как мы и задумывали: в новый журнал оценок (тип данных: словарь) добавляем оценки студентов, причём пустой журнал мы создали в основном скрипте, а не использовали его из значения по умолчанию.

А теперь не будем передавать пустой журнал изначально в качестве аргумента — он ведь и так создаётся по умолчанию:

```
# Добавим в журнал для новой группы оценку Смирнову
# Сам журнал не передаём в виде аргумента
# Пустой журнал будет использован как аргумент по умолчанию
group2 = add_mark('Smirnov', 3)
print(group2)
# Будет напечатано:
# {'Smirnov': 3}

# Аналогично для новой группы 3 добавим оценку Кузнецовой
group3 = add_mark('Kuznetsova', 5)
print(group3)
# Будет напечатано:
# {'Smirnov': 3, 'Kuznetsova': 5}
```

? Очень неожиданно, не так ли? Мы думали, что по умолчанию используется пустой словарь для группы 3, но в нём почему-то уже оказались оценки для второй группы. Как же так?

Дело в том, что пустой словарь был присвоен как новый объект аргументу journal только один раз: когда интерпретатор впервые прочитал объявление функции в коде. В аргументе journal по умолчанию теперь хранится указатель на созданный при прочтении функции словарь. В первый раз [group2 = add_mark('Смирнов', 3)] мы действительно воспользовались новым словарём. А вот при повторном использовании данного словаря в качестве аргумента по умолчанию [group3 = add_mark('Кузнецова', 5)], мы модифицировали уже созданный ранее словарь.

Звучит сложно, не правда ли?

Если вы не поняли предыдущее объяснение, попробуйте осознать следующую иллюстрацию и сопоставить её с описанной выше проблемой.

Допустим, в учебную часть пришел преподаватель и спросил, куда ставить оценки за экзамен, если в кабинете нет ведомости. Ему сказали, что новая ведомость лежит в третьем ящике стола у входа (то есть преподаватель получил указатель на пустую ведомость, узнал, где она должна храниться).

Преподаватель пришёл на экзамен, ведомости не было, поэтому он взял ведомость из третьего ящика стола (перешёл по указателю), вписал туда оценки (изменил данный объект) и вернул ведомость на место.

Этот же преподаватель пришёл принимать экзамен у ещё одной группы. Ведомость снова никто не принёс, но он помнит, что в таком случае необходимо использовать ведомость из третьего ящика стола (снова переходит по указателю). Преподаватель внёс в эту ведомость новые оценки и вернул её на место.

Что же теперь записано в ведомости? Оценки обеих групп! Преподаватель только один раз получил указания, где брать ведомость, если её нет, и записывал оценки только в неё.

То же самое происходит и с пустым словарём, который мы установили в виде аргумента по умолчанию: он был создан только 1 раз, а затем он только модифицировался функцией, а не создавался пустым заново.

Давайте сделаем так, чтобы каждый раз при запуске функции без словаря всё-таки создавался пустой словарь, а не использовался однажды созданный. Для этого изначально присвоим аргументу journal значение по умолчанию *None* и уже непосредственно в теле функции будем создавать пустой словарь, если journal является *None*:

```
def add_mark(name, mark, journal=None):
    # Если журнал является None
    # (напоминание: сравнивать объект с None
    # корректнее через оператор is),
    # запишем в journal пустой словарь
    if journal is None:
        journal = {}
    journal[name] = mark
    return journal
```

Сразу запустим фрагмент кода, который вызвал неожиданное поведение функции:

```
group2 = add_mark('Smirnov', 3)
print(group2)
# Будет напечатано:
# {'Smirnov': 3}

group3 = add_mark('Kuznetsova', 5)
print(group3)
# Будет напечатано:
# {'Kuznetsova': 5}
```

Как видите, теперь всё встало на свои места: оценки для новых групп теперь действительно записываются в новый словарь, а не добавляются в ранее созданный.

Теперь остаётся понять, какие объекты нельзя использовать в качестве аргументов по умолчанию.

К сожалению, полностью осознать ответ на этот вопрос можно будет только после изучения **объектно-ориентированного программирования**, то есть через несколько модулей.

Пока что вам стоит запомнить, что **в качестве аргументов по умолчанию некорректно использовать изменяемые типы данных: списки, словари, множества...** У всех из них есть собственные функции и методы, с помощью которых происходят

В качестве аргументов по умолчанию точно можно использовать «простые» типы данных, которые не содержат в себе дополнительные значения, такие как *int*, *float*, *str*, *bool*, *None*.

изменения этих объектов (например, append, add и т. д.) непосредственно внутри объекта.

Остальные типы данных, скорее всего, состоят из перечисленных «простых» и потому являются «сложными», например списки могут состоять из всех перечисленных выше «простых» элементов.

Ввиду особенностей работы со «сложными» типами данных, они являются изменяемыми, поэтому использование их в качестве аргументов по умолчанию нежелательно.

Чтобы избежать ошибок, связанных с изменяемыми типами данных, используйте *None* в качестве значения аргумента по умолчанию и создавайте новый объект уже в теле функции, как это было сделано в примере выше.

✍ В этом юните вы:

- узнали, что исключения — это полезный инструмент, с помощью которого можно указать пользователю, которым можете быть и вы сами, на допущенную ошибку в данных;
- научились возвращать исключение в случае некорректных аргументов;
- научились задавать аргументы по умолчанию и использовать их, чтобы не прописывать каждый раз часто повторяющиеся аргументы;
- поняли, что аргументы по умолчанию необходимо использовать на практике с осторожностью, чтобы не возникали ошибки; вместо изменяемых типов данных в качестве аргумента по умолчанию необходимо использовать *None*.

Закрепите полученные знания с помощью **заданий на повторение** ↓

Задание 3.3

1/1 point (graded)
Выберите тип ошибки, которую принято возвращать, если в функцию переданы некорректные аргументы:

- ○ ZeroDivisionError
- ○ KeyError
- ◉ ValueError ✔

NameError

Ответ

Верно: В данном юните мы возвращали эту ошибку, чтобы сообщить, что аргументы некорректны.

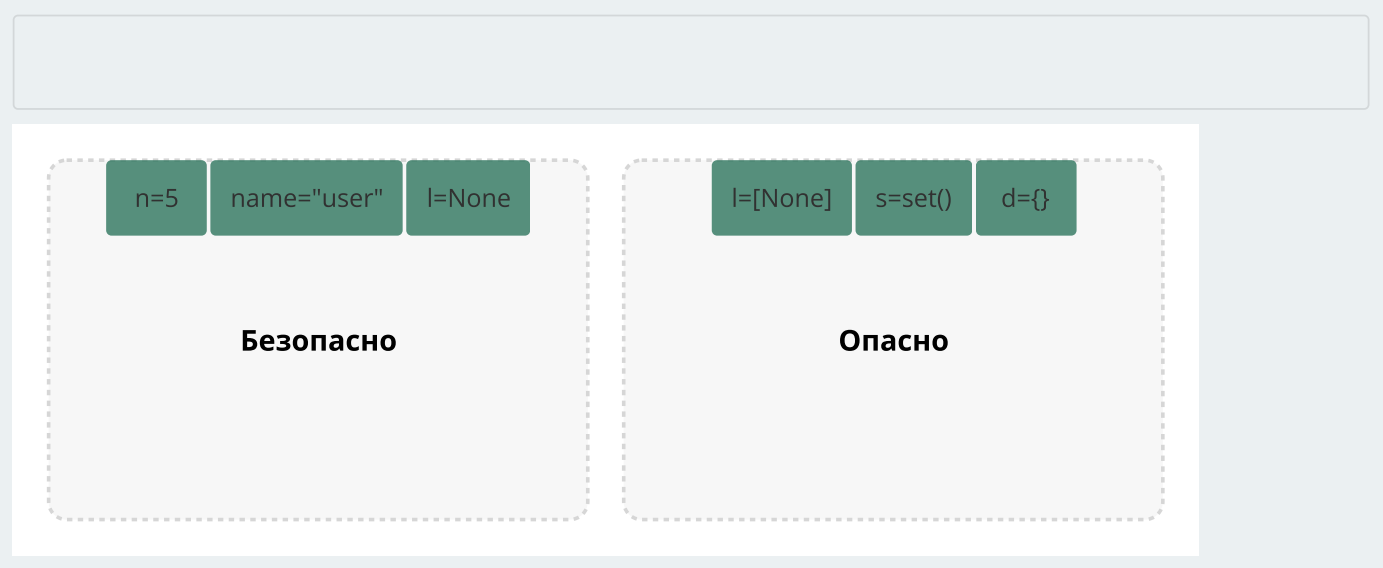
Отправить

Задание 3.4

1/1 балл (оценивается)

Справка по клавиатуре

Соотнесите аргументы с уровнем опасности их использования по умолчанию.



Сбросить

РЕЗУЛЬТАТ

Хорошо! Вы справились с этим заданием.

ЗАДАНИЕ 3.5 (ВЫПОЛНЯЕТСЯ В CODEBOARD НИЖЕ)

?

Дана функция add_mark:

```
def add_mark(name, mark, journal=None):
    # Добавьте здесь проверку аргумента mark
    if journal is None:
        journal = {}
    journal[name] = mark
    return journal
```

Дополните её код таким образом, чтобы возникала ошибка *ValueError* с текстом *"Invalid Mark!"* при попытке поставить оценку не из списка: 2, 3, 4 или 5.

Пример:

```
add_mark('Ivanov', 6)
# ValueError: Invalid Mark!
```

При работе с заданием в *Codeboard* **не используйте кириллицу** — писать решение надо полностью **на английском языке**, иначе написанный вами код не запустится!

1. Начните работу традиционно с открытия файла main.py — там вы увидите условие задачи.
2. Впишите требуемый код вместо знаков ??? и нажмите **RUN.**
3. Нажмите **SUBMIT**, чтобы отправить ваш ответ на проверку и получить баллы в случае верного решения.

Задание 3.5 (Внешний источник)

(1.0 из 1.0 баллов)

6.3.5

main.py

Root
- src
  - __init__.py
  - **main.py**
- __init__.py

```
1  def add_mark(name, mark, journal=None):
2     # Добавьте здесь проверку аргумента mark
3
4      if journal is None:
5          journal = {}
6      journal[name] = mark
7      return journal
8
9  #попытка вызвать функцию с некорректными аргументами
10 try:
11     print(add_mark('Ivanov', 6))
12 #должна завершиться с ошибкой ValueError, которую мы выведем в блоке except
13 except ValueError as e:
14     print(e)
```

```
This will display the output.
```

Input to your program (press Enter to send)

Send

User: #anonymous (sign in (/signin?