Урок 1

Python — синтаксис языка, базовые структуры данных

Основные элементы языка Python. Функциональное программирование. Общие вопросы.

Выполняя курсовые проекты, вы приобрели знания и навыки, который сможете продемонстрировать при устройстве на работу. Вам предстоит пройти техническое интервью, которое всегда входит в собеседование.

В данном курсе мы повторим пройденный материал и разберем наиболее частые вопросы, с помощью которых работодатели оценивают кандидатов на должность Python-разработчика. Некоторые задачи будут проверять ваше умение адаптироваться в нестандартных ситуациях.

Начало собеседования

Вы решили устроиться на работу, и после поиска — продолжительного или не очень — вам звонит представитель компании, куда вы отправляли резюме. Поздравляем, собеседование уже началось! Пусть вас не смущает отсутствие вопросов, связанных непосредственно с вакансией. На данном этапе определяется, подходите ли вы компании на базовом уровне. Если этот этап успешно пройден, вам предстоит общение по техническим вопросам. Подготовимся к ним.

До технического собеседования вы можете задать рекрутеру вопросы. Рекомендуем узнать:

1. Местоположение офиса.
2. График работы.
3. Размер команды, свою позицию в ней.
4. Будущие обязанности в команде и зоны ответственности, цели разработки.
5. Размер компании, направления разработки (внутренние продукты или работа на заказ).
6. Стек технологий, стандарты и их контроль.
7. Перспективы развития в команде.
8. Методологии разработки.
9. Социальный пакет и другие привилегии для сотрудников.
10. Условия трудового контракта: есть ли материальная ответственность любого рода или другие риски.
11. Размер заработной платы и порядок ее выплаты.

На данном этапе не стоит пытаться узнать, как отмечается новогодний корпоратив или какая атмосфера в команде. При первом общении вы можете составить свое представление об организации и понять, интересно ли вам эта работа.

Если вы сумели заинтересовать рекрутера, вас пригласят в офис или предложат пройти собеседование удаленно — через Skype или другим способом. Особое внимание уделите выбору даты и времени собеседования: оно должно быть удобным для вас и полностью свободным от других дел. Вашим большим плюсом будет пунктуальность: желательно быть на месте и готовым интервью за 10–15 минут до начала собеседования.

На интервью приходите в опрятном виде и помните, что «встречают по одежке, а провожают по уму».

Резюме

Абсолютно все, что отражено в вашем резюме, может повлиять на ход собеседования. Рекрутер или интервьюер может задать вопрос по любому пункту и будет ожидать адекватного ответа. Поэтому не указывайте в резюме технологии, которыми не владеете. Всегда лучше чего-то не знать, чем быть

уличенным во лжи, если не сможете внятно ответить на поставленный вопрос. В этом случае вы сильно упадете в глазах интервьюера, и скорее всего вам будет отказано в работе.

Необходимые базовые знания

Компании нацелены на найм грамотного специалиста, и в начале второго этапа собеседования вам обязательно зададут вопросы, которые выявляют уровень знаний в основах языка. В этом курсе мы рассмотрим вопросы по Python и типовые задачи, встречающиеся на собеседовании.

Основные вопросы о языке, как правило, затрагивают эти темы:

1. Типизация языка.
2. Особенности версий.
3. Принципы работы языка в выбранной среде.
4. Циклы, массивы и другие фундаментальные конструкции.

На эти вопросы лучше ответить как можно более развернуто и правильно, иначе вы рискуете не пройти собеседование. Поэтому следует повторить материалы подготовительного курса Python.

Рассмотрим типовые задачи, которые часто включают в базовую часть интервью.

Что представляет собой язык программирования Python

- ***Расскажите про язык программирования Python. В чем его сильные и слабые стороны по сравнению с другими технологиями?***

О **Python** (произносится как «питон», некоторые говорят «пайтон») лучше всех говорит его создатель — голландец Гвидо ван Россум: «Python — интерпретируемый, объектно-ориентированный высокоуровневый язык программирования с динамической семантикой. Встроенные высокоуровневые структуры данных в сочетании с динамической типизацией и связыванием делают язык привлекательным для быстрой разработки приложений (Rapid Application Development).

К современному языку программирования предъявляются следующие основные критерии:

1. **Качество программного обеспечения.**
2. **Библиотеки поддержки.**
3. **Переносимость программ.**
4. **Скорость разработки.**

В Python, по сравнению с компилирующимися или строго типизированными языками, такими как Java, C++ или C, трудозатраты разработчика снижаются во много раз. Код, написанный на Python, втрое лаконичнее эквивалентного на Java или C++. А это значительно меньше ввода с клавиатуры, трудозатрат на поддержку и времени на тестирование и отладку. Еще один большой плюс в том, что программы на Python выполняются сразу, минуя этап компиляции, что экономит рабочее время программиста.

Задачи, которые решает Python

- ***Какие задачи решаются с помощью языка Python?***

Python — удачно спроектированный язык программирования, поэтому подходит для решения широкого спектра задач. Он используется как инструмент управления другими программными компонентами, применяется для реализации самостоятельных программ. Фактически, возможности Python как многоцелевого языка практически не ограничены: он может использоваться для реализации чего угодно — от создания веб-сайтов и игровых программ до управления роботами и космическими кораблями.

Сферы использования Python можно разбить на несколько широких категорий.

1. **Системное программирование.**

Встроенные в Python интерфейсы доступа к службам операционных систем делают его идеальным инструментом для создания переносимых программ и утилит системного администрирования (иногда они называются инструментами командной оболочки).

2. **Графический интерфейс.**

Простота Python и высокая скорость разработки делают его отличным средством для создания графического интерфейса. В состав Python входит стандартный объектно-ориентированный интерфейс к **Tk GUI API,** который называется **Tkinter**.

3. **Веб-сценарии.**

Интерпретатор Python поставляется вместе со стандартными интернет-модулями, которые позволяют программам выполнять сетевые операции как в режиме клиента, так и сервера. Сценарии могут производить взаимодействия через сокеты, извлекать информацию из форм, отправленных серверным **CGI**-сценариям; передавать файлы по протоколу **FTP**; обрабатывать файлы **XML**; передавать, принимать, создавать и производить разбор писем электронной почты; загружать веб-страницы с указанных адресов **URL** и многое другое.

4. **Интеграция компонентов.**

Возможность Python расширяться и встраиваться в системы на языке C делает его удобным и гибким для описания поведения других систем и компонентов. Например, интеграция с библиотекой на C позволяет Python проверять наличие библиотечных компонентов и запускать их. Встраиваясь в программные продукты, Python дает возможность настраивать их, не пересобирая.

5. **Приложения баз данных.**

В Python есть интерфейсы доступа ко всем основным реляционным базам данных: Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite и многим другим.

6. **Быстрое создание прототипов.**

В программах на Python компоненты, написанные на нем и C, выглядят одинаково. Благодаря этому можно сначала создавать прототипы систем на Python, а затем переносить выбранные компоненты на компилируемые языки. Это существенно экономит время разработки.

На каких проектах используется Python

- ***Где используется Python?***

- Google использует Python в своей поисковой системе. Компания оплачивала труд создателя Python — Гвидо ван Россума (сейчас он работает в DropBox Inc).
- Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm и IBM используют Python для тестирования аппаратного обеспечения.
- Служба коллективного использования видеоматериалов YouTube в значительной степени реализована на Python.
- NSA использует Python для шифрования и анализа разведданных.
- Компании JPMorgan Chase, UBS, Getco и Citadel применяют Python для прогнозирования финансового рынка.
- Популярная программа для обмена файлами в пиринговых сетях BitTorrent написана на Python.
- Веб-фреймворк App Engine от Google использует Python в качестве прикладного языка программирования.
- NASA, Los Alamos, JPL и Fermilab применяют Python для научных вычислений.

Типы данных в Python

- ***Какие бывают типы данных в Python?***

Практически все, с чем мы сталкиваемся, разрабатывая программу на Python, является объектом. Типы объектов могут быть встроенными или описанными разработчиком с помощью системы классов. Типы являются классами, а значения объектов — экземплярами классов.

В Python 2 разные понятия типа и класса, а в Python 3 это по сути одно и то же.

В Python есть динамическая поддержка типизации — во время исполнения интерпретатор сам определяет тип переменной. Правильно говорить «связывание значения с некоторым именем» вместо «присвоение значения переменной». Именно объект в памяти имеет тип, а переменная является просто указателем на него.

В Python есть хорошая коллекция объектных типов, которые встроены в сам язык. Не нужно создавать собственную реализацию объектов. Если в вашем проекте нет необходимости в специальных видах обработки, которые не обеспечиваются встроенными типами объектов, лучше не писать собственную реализацию, а использовать уже имеющиеся типы данных.

Рассмотрим встроенные типы данных.

Название типа	**Описание**
int	Целые числа. Пример: 2, 4, 8, -10, -2
float	Числа с десятичной точкой. Пример: 2.6, -5.2
str	Строки(заключены в кавычки): “Hello”, ‘Вася’
bool	Логический тип. Принимает два значения: True, False
list	Список. Пример: [2, 2.4, “Hello”]

tuple	Кортеж. Пример: (2, 2.4, “Hello”)
dict	Словарь. {“name”: “Вася”, “age”: 10}
set	Множества. set(['h','e','l','l','o'])
frozenset	Неизменяемые множества. frozenset('qwerty')
complex	Комплексные числа. complex(1, 2)
None	Неопределенный тип данных

Явное и неявное преобразование типов

- ***Как преобразовать один тип в другой?***

Имеется код:

```
def concat():
    name = 'Игорь '
    age = 18
    res = name + age
    print(res)

concat()
```

Если вы попытаетесь выполнить инструкцию, получите сообщение об ошибке:

```
TypeError: Can't convert 'int' object to str implicitly
```

То есть строку «Игорь 18» мы не получим.

Выполним такой код:

```
def sum():
    x = 1.9
    y = 2
    z = x + y
    print(z)

sum()
```

И получим:

```
3.9
```

Это вполне ожидаемый результат. Но на самом деле интерпретатор не сможет выполнить операции с разными типами. Чтобы программист собственноручно не преобразовывал их, Python в некоторых случаях сам делает это (неявно).

Почему в первом примере ошибка, и Python не преобразовал число к строке? Философия Python гласит о том, что явное лучше неявного. При неявных преобразованиях порождается целый пласт ошибок, которые сложно отследить.

Чтобы выполнить первый пример, нужно преобразовать число в строку.

```
def concat():
    name = 'Игорь '
    age = 18
    res = name + str(age)
    print(res)

concat()
```

В результате получим:

```
Игорь 18
```

В Python для явного преобразования типа используются одноименные функции. Для преобразования к типу **int** → применяйте функцию **Int()**, для преобразование к строке — **str()** и так далее.

Логические операции в Python

- ***Какие логические операторы вы знаете, что они выполняют?***

В жизни мы часто соглашаемся с фактами, утверждениями или отрицаем их. «Разность чисел 7 и 4 меньше 5» — истина. А «сумма чисел 8 и 3 меньше 2» — ложь. Если в результате вычисления мы получаем результат, который может быть или истинным, или ложным, такое выражение называется логическим. Оно имеет свой тип данных — **bool**.

Со всеми логическими операторами мы знакомы со школьного курса математики.

Логические операторы

Оператор	Описание
>	Больше
<	Меньше
==	Равно
!=	Не равно
>=	Больше или равно
<=	Меньше или равно

```
def check():
    a = 3 > 5
    print(a)

check()
```

Переменная будет иметь тип **bool** и значение **False** — то есть данное утверждение неверно. На экране увидим:

```
False
```

Запишем другое утверждение:

```
def check():
    a = (6 + 2) > 5
    print(a)

check()
```

В результате на экране получим:

```
True
```

Значение **True** — утверждение правдиво.

Любое значение может быть преобразовано к логическому типу функцией **bool()**.

bool(x) позволяет преобразовать структуру типа **bool**, которая использует стандартную проверку на истину. Если **x** является ложью, то возвращает значение **False**, в ином случае — **True**.

Внимание! Функция **bool()** возвращает **False** от нуля и любой пустой последовательности, а во всех остальных случаях будет **True**.

bool(0) → False

bool(“”) → False

bool(False) → False

bool(-1) → True

bool(“Hello”) → True

bool(“0”) → True

Помимо преобразования аргумента в булево значение возможна и обратная операция — перевод аргумента типа **bool** в целочисленный тип. Для этого применяется функция **int()**:

int(True) → 1

int(False) → 0

Логические выражения используются в инструкциях языков программирования.

Инструкции ветвления

- ***Что такое ветвление и зачем нужно? Какие его конструкции вы знаете?***

В курсе теории программирования доказано, что решение задачи любой сложности можно составить из трех структур: следование, ветвление и цикл.

Следованием называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или составных).

Ветвление задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия.

Цикл задает многократное выполнение оператора.

В следовании все достаточно просто: все инструкции (команды) выполняются последовательно, пока программа не завершится.

Рассмотрим подробнее ветвление:

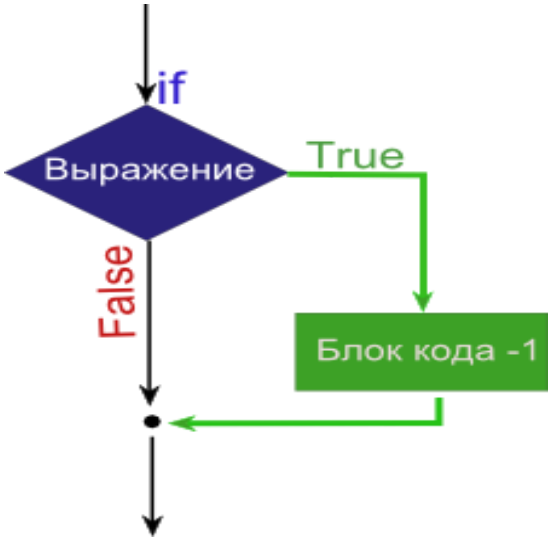


Схема ветвления if

Описание схемы

Оператор **if** является инструкцией. Он может применять любые выражения, которые в дальнейшем автоматически будут преобразовываться к логическому типу. Основная задача оператора **if** — выполнить определенный блок кода в случае верного утверждения.

Если выражение истинно — блок кода выполнится, если ложно — интерпретатор пропускает блок кода и программа будет выполняться дальше.

Рассмотрим на примере:

```
def psswrd_check():
    correct_password = 'ScT9#'      # правильный пароль, для сопоставления
                                    # с паролем, введенным пользователем

    password = input('Введите пароль:')
                                    # выводим на экран просьбу ввести пароль
                                    # и переключаемся в режим ввода информации

    if password == correct_password: # если пользователь ввел
                                     # правильный пароль
        print('Вы ввели правильный пароль, доступ разрешен')

    if password != correct_password:  # если пользователь ввел
                                      # неправильный пароль
        print('Вы ввели неправильный пароль, доступ запрещен')

psswrd_check()
```

В этой программе мы запрашиваем пароль у пользователя, и в случае верного ввода разрешаем доступ.

Конструкция **password == correct_password** сравнивает пароль, введенный пользователем, с тем, что хранится в системе. Если они равны — выводим сообщение, что доступ разрешен.

В конструкции **password != correct_password** сравниваем пароль с хранящимся у нас, и если он неверный — выводим сообщение о том, что доступ запрещен.

Пример ветвления с инструкцией else:

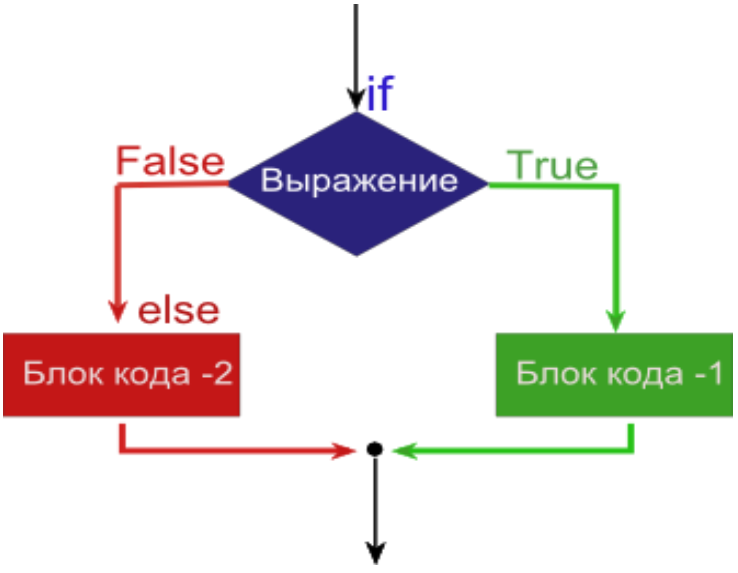


Схема ветвления if else

На этой схеме показано, что если выражение истинно (**True**), то выполнится зеленый блок кода. Если выражение ложно (**False**) — красный.

Модифицируем пример:

```
def psswrd_check():
    correct_password = 'ScT9#' # правильный пароль для сопоставления с
                               # паролем, введенным пользователем
    password = input('Введите пароль:')
                               # выводим на экран просьбу ввести пароль
                               # и переключаемся в режим ввода информации

    if password == correct_password: # если пользователь ввел
                                     # правильный пароль
        print('Вы ввели правильный пароль, доступ разрешен')

    else:    # если пользователь ввел неправильный пароль
        print('Вы ввели неправильный пароль, доступ запрещен')

psswrd_check()
```

В результате получили более лаконичный код, но функционал от этого не изменился.

Вложенные инструкции

Внутри блока условной инструкции могут находиться любые другие инструкции, в том числе условная. Их называют вложенными. Синтаксис вложенной условной инструкции:

```
if условие1:
    # блок кода
if условие2:
    # блок кода
else:
    # блок кода
else:
    # блок кода
```

Следует обратить особое внимание на отступ перед инструкциями: вложенная условная инструкция отделяется отступом с четырьмя пробелами. Ограничений на использование вложенных инструкций нет, но всегда надо помнить об отступе.

Условие 2 проверяется, только если верно условие 1.

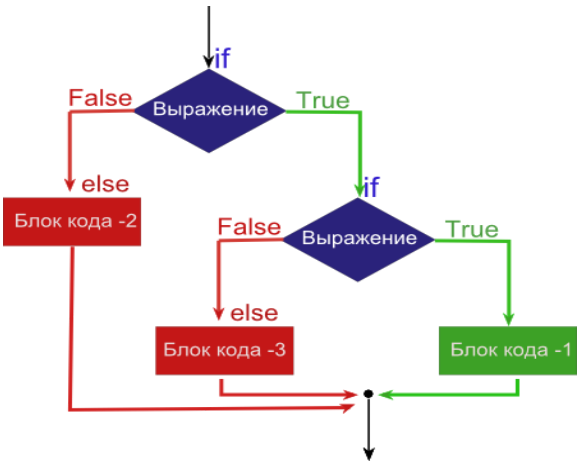
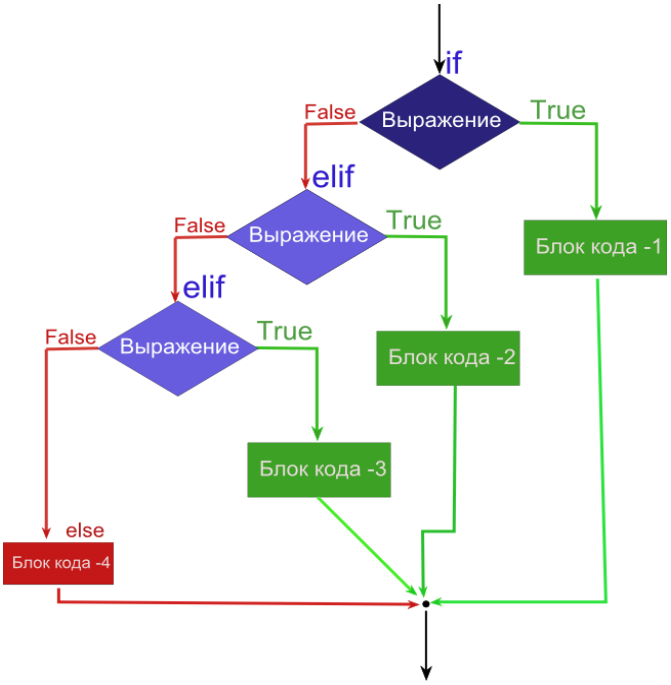


Схема вложенных инструкций ветвления

Пример ветвления с инструкцией elif:



Полная схема инструкции ветвления

Оператор **elif** переводится как «иначе если». Выражение, которое стоит после него, выполнится, только если все вышестоящие условия ложны (не выполняются).

В данной схеме может выполниться только один блок кода (или 1, или 2, или 3, или 4). Если одно из выражений истинно, то нижестоящие условия проверяться не будут.

Если нужно, чтобы проверялись все условия независимо от результата предыдущего, следует использовать несколько независимых операторов **if**.

Пример

```
def if_check():
    color = 4
    if color == 1:
        print('один')
    elif color == 2:
        print('два')
    elif color == 3:
        print('три')
    # else выполняется, только если все предыдущие проверки вернули False
    else:
        print('неизвестное число')

if_check()
```

Важно: в Python нет стандартного оператора **switch**. Для множественного условия применяют словари.

Пример

```
unit_to_multiplier = {
    'mm': 10**-3,
    'cm': 10**-2,
    'dm': 10**-1,
    'm': 1,
    'km': 10**3
}
```

Использование словаря:

```
def dict_check():
    unit = 'am'

    if unit in unit_to_multiplier:
        mult = unit_to_multiplier[unit]
        print(mult)
    else:
        # обработка отсутствия значения в словаре
        print("Такого ключа в словаре нет")

dict_check()
```

Операторные скобки

- ***Как в Python выделяются блоки кода?***

Во всех языках программирования, чтобы выделять блоки кода, используются специальные синтаксические конструкции, в которых показывается начало и конец блока с кодом. В C++ это фигурные скобки: **{ Блок кода }**. В Pascal — ключевые слова: **begin блок кода end**. В Python — это одинаковый отступ слева перед всеми инструкциями блока с кодом. Код на Python принято писать, применяя стандарт **PEP 8**. В соответствии с ним нужно использовать 4 пробела на один уровень отступа.

Нельзя смешивать символы табуляции и пробелы. Самый распространенный способ отступов — пробелы. Это удобно, делает код читабельнее и заставляет программиста более качественно писать код, используя правильную табуляцию.

Во многих языках символом конца страницы является точка с запятой. Зная это, весь код программы можно записать в одну строку. Но сможет ли кто-нибудь его понять?

Циклы в Python

- ***Что такое цикл? Зачем и как его использовать?***

Написанные нами ранее программы запускались, выполняли требуемые действия, выводили результат и заканчивали работу. Чтобы передать в программу другие данные, нужно перезапустить ее с новыми параметрами. Но большинство программ так не работают.

В основном они действуют непрерывно: выполнив одно действие, ожидают дальнейших инструкций. И так до тех пор, пока пользователь не завершит их. Это и есть работа программы в цикле.

Циклы — это инструкции, выполняющие одну последовательность действий, пока в силе заданное условие.

Цикл while

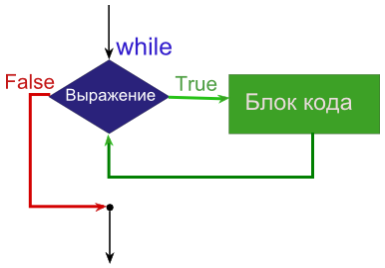


Схема цикла while

Описание схемы

Если выражение соответствует **True**, блок кода выполняется, и после этого программа снова возвращается к проверке логического выражения. Если выражение ложно (**False**), программа пропустит блок кода и продолжит работу.

Блок кода внутри цикла называется **телом цикла.**

Однократное выполнение тела цикла — это **итерация.**

Пример

```
def inp_check():
    number = int(input('Введите целое число от 0 до 9: '))
    if number < 10:
        while number < 10:
            print(number)
            number = number + 1
            print('программа завершена успешно')
    else:
        print('Вы ввели неверные данные')

inp_check()
```

Здесь мы просим пользователя ввести любое число от 0 до 9. Дальше проверяем, находится ли число в заданном диапазоне, и если да — выводим на экран все числа от введенного до 9. К примеру, если введем 5, то программа выдаст 5,6,7,8 и 9.

Сколько раз выполнится тело цикла, заранее неизвестно — это зависит от заданного значения переменной **number**.

Обратите внимание на четвертую строчку: при каждом ее выполнении в цикле ее значение будет увеличиваться на единицу — до тех пор, пока значение переменной **number** не станет больше или равно 10. При этом значении логическое выражение **number<10** станет ложным и цикл завершится.

Зацикливание

- ***Что такое зацикливание? Нужно ли его применять?***

Рассмотрим пример:

```
x = 5
while x > 0:
    print("!" + x)
    x = x + 1
```

Если запустим этот код, увидим, как в терминале выводится много восклицательных знаков. При заданном условии цикл будет выполняться бесконечно, ведь условие **x>0** всегда верно, так как изначально **x** равен 5. В написании программ лучше избегать бесконечных циклов, иначе операционная система посчитает такую программу зависшей и предложит «снять с нее задачу».

Инструкции break, continue, else

Вопрос:

- ***Для чего используются операторы break, continue?***

В теле цикла можно использовать вспомогательные инструкции **break** и **continue**, чтобы упростить ваш код и сделать его более читабельным.

Оператор **break** досрочно прерывает **цикл**.

```
def check_break():
    i = 0
    while True:
        if i >= 10:
            # инструкция break при выполнении немедленно
            # заканчивает выполнения цикла
            break
        print(i)
        i += 1

check_break()
```

Оператор **continue** начинает следующий проход цикла, минуя оставшееся тело цикла.

```
def check_continue():
    i = 0
    while i < 10:
        i += 1
        if i == 5:
            # переходим к очередному шагу цикла,
            # пропуская оставшиеся инструкции цикла
            continue
        print(i)

check_continue()
```

Оператор **else** в циклах действует, только если цикл выполнен успешно.

Функции

- ***Что такое функции?***

Функция — это, по сути, изолированный блок кода, к которому можно многократно обращаться в процессе выполнения программы.

- ***Зачем использовать функции?***

В первую очередь чтобы сократить объем исходного кода: рационально вынести часто повторяющиеся выражения в отдельный блок и затем по мере надобности обращаться к нему.

Определим простейшую функцию:

```
# определение функции
def summ(a, b):
    c = a + b
    return c
# вызов функции
res = summ(5, 10)
print(res)
```

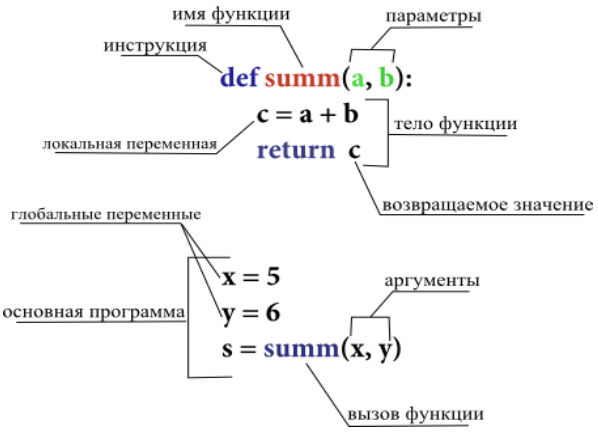


Схема и термины функции

- **def** — это команда Python, позволяющая создавать функцию.
- **summ** — это имя функции, которое может быть почти любым (но лучше осмысленным). К имени функции применяются те же требования, что и к имени переменной.

После имени функции в скобках можно перечислить ее параметры. Если их нет, скобки остаются пустыми. Затем следует двоеточие: оно обозначает конец заголовка функции.

Далее идет блок с кодом — тело функции. Он обязательно должен быть с отступом. В конце тела функции может присутствовать **return** (возвращает значение).

Блок кода тела функции выполнится, когда функция будет вызвана в основной программе. Если функция есть в коде, но нет ее вызова, то блок кода не выполнится.

Если функция написана правильно — нам не важно, что внутри нее.

Часто функции позволяют разбить программу на логические части, чтобы облегчить тестирование.

Документирование функций

- ***Как документируют функции? Зачем это нужно?***

Если мы перед началом использования функции изучаем, как она написана, то с большой долей вероятности это плохая функция. Создавая функцию, хорошие программисты пишут небольшой комментарий к ней, чтобы даже через год ее можно было использовать. Комментарии принято добавлять в виде многострочной строки после заголовка функции.

```
def summ(a, b):
    """
    Возвращает сумму аргументов
    """
    c = a + b
    return c
```

Аргументы функции

- ***Как передать аргументы функции?***

Иногда, чтобы функция выполнила заданный алгоритм, ей нужно получить аргументы. Они передаются в скобках при вызове функции. Чтобы функция могла взять эти данные, при ее создании необходимо их описать (в скобках после имени функции). Они представляют собой переменные.

При вызове функции заданные аргументы подставляются вместо параметров-переменных. Практически всегда количество параметров и аргументов должно совпадать, но можно сделать переменное количество принимаемых аргументов. Ими могут выступать как значения, так и переменные, ссылающиеся на них.

Пример:

```
def average(x, y, z, q):
    sum = x + y + z + q
    return sum/4

x = 5
y = 10
z = -8
result = average(x, y, z, 10)
print(result)
```

Эта функция вычисляет среднее арифметическое.

Глобальные и локальные переменные

- ***Что такое локальные и глобальные переменные?***

Все переменные, которые мы создаем внутри тела функции, являются **локальными** и существуют только во время ее выполнения.

Но есть переменные, которые объявлены в основной части программы, вне в функции. Они существуют до конца выполнения программы и называются **глобальными**.

В функциях не рекомендуется использовать глобальные переменные.

Пример:

```
a = 15
def local(b):
    # b, c — локальные переменные, доступны только во время выполнения функции
    # a — глобальная переменная, доступна во всем модуле (.py файле)
    c = 10
    print('b = {}, c = {}, a = {}'.format(b, c, a))
local(5)
```

Не рекомендуется использовать глобальные переменные, потому что нужно стремиться к переносимости функций — возможности вызывать их в любой другой программе. Если там не будет нужной глобальной переменной, функция не станет работать.

В Python есть оператор **global** — это из ключевых слов в языке. С его помощью переменная будет доступна для блока кода, следующего за оператором. Не рекомендуется создавать наименование перед тем, как объявлять его глобальным.

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

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

В Python 3 было добавлено новое ключевое слово **nonlocal**. С его помощью можно добавлять переопределение области во внутреннюю область. Оператор **nonlocal** отвечает за поиск переменной в объемлющей функции. Один из наиболее простых примеров его использования — это создание функции, которая может увеличиваться:

```
def adder():
    x = 10
    def internal():
        print(x)
        x += 1
    internal()
    print(x)

adder()
```

Если запустить этот код, возникнет ошибка **UnboundLocalError**. К переменной **x** осуществляется доступ (**print(x)**) до того, как с ней свяжется значение. Добавим **nonlocal**:

```
 def adder():
   x = 10
   def internal():
        nonlocal x
        print(x)
        x += 1
    internal()
    print(x)

adder()
```

Результат:

```
10
11
```

Благодаря оператору **nonlocal** мы можем получать доступ к переменным, определенным вне функции. Как правило, **nonlocal** используется для назначения переменных во внешней области, но не в глобальной.

Lambda-функции

- ***Что такое lambda-функции?***

Lambda-функции — это анонимные функции, содержащие всего одно выражение. Они выполняются гораздо быстрее обычных функций. Создать анонимную функцию можно с помощью инструкции **lambda**. Их не обязательно присваивать переменной:

```
# В переменную f будет сохранена ссылка на объект-функцию
f = lambda f: f * 2
# Суть в том, что инструкция lambda возвращает ссылку на функцию
# Тоже, что и
# def mult(f):
#    return f*2
# Только объявляется компактнее и работает быстрее
print(f(4))
# lambda-функции можно использовать и в выражении
print((lambda f: f * 2)(4))
```

Область видимости

- ***Для чего нужна область видимости, что под этим подразумевают?***

Область видимости — это пространство имен. Это место в коде, где имени было присвоено значение и где в программе к нему можно обратиться.

Почти все, что связано с именами, включая и область видимости, в Python имеет отношение к присваиванию. Прежде чем использовать переменную, ей необходимо присвоить значение; параметры с именами используются только после их названия. Имена не объявляются заранее — интерпретатор Python по местоположению операции присваивания связывает имя с конкретным пространством имен. То есть место, где переменной присваивается имя, определяет пространство имен, в котором будет находиться имя и область его видимости.

Изолируемые области видимости переменных в Python создают только функции.

В Python есть 4 области видимости:

1. Локальная.
2. Объемлющей функции.
3. Глобальная (модуля).
4. Встроенная (builtins) — предопределенные имена (например, имена встроенных функций).

Поиск переменной происходит поочередно с первой по четвертую.

```
x = 5   # глобальная переменная – доступна в любом месте данного модуля(файла)
def outside():
    y = 10      # доступна в теле данной функции + во всех вложенных

    def inside():
        z = 15 # доступна только в теле данной функции
        print('inside x: {}, y: {}, z: {}'.format(x, y, z))
    inside()
    print('outside x: {}, y: {}, z: {}'.format(x, y, 'z недоступна'))
outside()
print('inside x: {}, y: {}, z: {}'.format(x, 'y недоступна', 'z недоступна'))

x = 5
def wrapper():
    x = 1
    def test1():
        x = 10 # локальная переменная x перекрывает видимость глобальной x
        print('test1 x = ', x)
    def test2():
        print('test2 x = ', x) # Ищем переменную в объемлющей функции
    def test3():
        global x # инструкция global – поиск переменной в глобальной области
        print('test3 x = ', x)
        x = 25
    def test4():
        nonlocal x # Есть инструкция nonlocal –
                   # поиск переменной в объемлющей функции
        print('test4 x = ', x)

    test1()
    test2()
    test3()
    test4()

wrapper()
print('after wrapper x = ', x)
```

Произвольное количество аргументов

- ***Можно ли в функцию передать произвольное количество аргументов?***

Чтобы в функцию передать любое (неопределенное) количество аргументов, используют конструкцию ***args** в параметре функции, где **args** — это произвольное имя.

Используя эту особенность, можно написать универсальную функцию, вычисляющую среднее арифметическое, вместо той, которая работает только с 4 параметрами.

```
def average(*args):
    summ = 0
    for arg in args:
        summ += arg
    print(summ)
```

Вызов этой функции с аргументами:

```
average(1, 3, 5, 2)
```

Кортеж, элементами которого будут переданные значения аргументов, — это **args**. Если мы вызовем функцию **average(1, 3, 5, 2)**, то в теле функции будет создан кортеж: **args=(1, 3, 5, 2)**.

Именованные аргументы

- ***Для чего нужны именованные аргументы?***

Если нужно передать функции много аргументов, довольно сложно не запутаться в их порядке. Ведь если из перепутать, значение будет передано не в ту переменную. Этот риск в Python снимают именованные аргументы.

```
def print_info(**kwargs):
    print("You name is %s %s. You age is %s. And your address is: %s"%
    (kwargs["name"],kwargs["surname"],kwargs["age"],kwargs["adress"]))

print_info(name="Василий",surname="Иванов",age="12",adress="ул.Белана 22")
```

Чтобы принять именованные аргументы, используется ****kwargs** в качестве имени параметра функции. В теле функции **kwargs** — обычный словарь, в котором ключами являются имена аргументов.

Для данного примера в теле функции словарик выглядит так:

```
{'adress': 'ул.Белана 22', 'age': '12', 'surname': 'Иванов', 'name': 'Василий'}
```

Значения по умолчанию

- ***Можно ли создать функцию со значениями по умолчанию?***

Создадим свою функцию с параметром по умолчанию:

```
def welcome(name="Инкогнито"):
    print("Приветствую вас, %s"%(name))
```

Попробуем вызвать функцию с параметром и без:

```
welcome("User")
```

```
welcome()
```

Параметры по умолчанию делают функции удобнее.

Рассмотрим подробнее функцию **print()**. Параметры **sep**, **end** и **file** как раз и являются параметрами по умолчанию. Если вы не указываете их значения, в теле функции используются те, что заданы в объявлении. Но можно и указать значения явно.

Для начала рассмотрим, за что отвечают значения этих параметров:

- **sep** — строка, вставляемая между выводимыми значениями;
- **end** — строка, вставляемая в конец вывода функции **print()**;
- **file** — файл потока вывода.

Чтобы все стало понятно, наберите код:

```
print("Иван","Иванович","Иванов",sep="//",end="!!!")
```

Получите: **Иван//Иванович//Иванов!!!**

Поэкспериментируйте с другими значениями параметров по умолчанию. Теперь вы можете использовать значения по умолчанию функции **print()**.

Практическое задание

1. Написать функцию, реализующую вывод таблицы умножения размерностью **A**x**B**. Первый и второй множитель должны задаваться в виде аргументов функции. Значения каждой строки таблицы должны быть представлены списком, который формируется в цикле. После этого осуществляется вызов внешней lambda-функции, которая формирует на основе списка строку. Полученная строка выводится в главной функции. Элементы строки (элементы таблицы умножения) должны разделяться табуляцией.

2. Дополнить следующую функцию недостающим кодом:

```
def print_directory_contents(sPath):
    """
    Функция принимает имя каталога и распечатывает его содержимое
    в виде «путь и имя файла», а также любые другие
    файлы во вложенных каталогах.

    Эта функция подобна os.walk. Использовать функцию os.walk
    нельзя. Данная задача показывает ваше умение работать с
    вложенными структурами.
    """
    # заполните далее
```

3. Разработать генератор случайных чисел. В функцию передавать начальное и конечное число генерации (нуль необходимо исключить). Заполнить этими данными список и словарь. Ключи словаря должны создаваться по шаблону: “elem_<номер_элемента>”. Вывести содержимое созданных списка и словаря.

4. Написать программу «Банковский депозит». Она должна состоять из функции и ее вызова с аргументами. Клиент банка делает депозит на определенный срок. В зависимости от суммы и срока вклада определяется процентная ставка: 1000–10000 руб (6 месяцев — 5 % годовых,

год — 6 % годовых, 2 года — 5 % годовых). 10000–100000 руб (6 месяцев — 6 % годовых, год — 7 % годовых, 2 года – 6.5 % годовых). 100000–1000000 руб (6 месяцев — 7 % годовых, год — 8 % годовых, 2 года — 7.5 % годовых). Необходимо написать функцию, в которую будут передаваться параметры: сумма вклада и срок вклада. Каждый из трех банковских продуктов должен быть представлен в виде словаря с ключами (**begin_sum**, **end_sum**, **6**, **12**, **24**). Ключам соответствуют значения начала и конца диапазона суммы вклада и значения процентной ставки для каждого срока. В функции необходимо проверять принадлежность суммы вклада к одному из диапазонов и выполнять расчет по нужной процентной ставке. Функция возвращает сумму вклада на конец срока.

5. Усовершенствовать программу «Банковский депозит». Третьим аргументом в функцию должна передаваться фиксированная ежемесячная сумма пополнения вклада. Необходимо в главной функции реализовать вложенную функцию подсчета процентов для пополняемой суммы. Примем, что клиент вносит средства в последний день каждого месяца, кроме первого и последнего. Например, при сроке вклада в 6 месяцев пополнение происходит в течение 4 месяцев. Вложенная функция возвращает сумму дополнительно внесенных средств (с процентами), а главная функция — общую сумму по вкладу на конец периода.

Дополнительные материалы

1. https://habr.com/post/370831/.
2. https://habr.com/company/cit/blog/262887/.
3. http://www.quizful.net/category/python.
4. https://ru.stackoverflow.com/questions/460207/%D0%95%D1%81%D1%82%D1%8C-%D0%BB%D0%B8-%D0%B2-python-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80-switch-case.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Учим Python качественно (habr).
2. Самоучитель по Python.
3. Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание).
4. 15 основных вопросов для Python собеседования.
5. 20 вопросов и ответов из интервью на позицию Python-разработчика.