МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ КЫРГЫЗСКОЙ РЕСПУБЛИКИ

МЕЖДУНАРОДНЫЙ КУВЕЙТСКИЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ГУМАНИТАРНЫХ И ЕСТЕСТВЕННЫХ ДИСЦИПЛИН

Кафедра «ПЕДАГОГИКИ и ПРИКЛАДНОЙ ИНФОРМАТИКИ"

«СОГЛАСОВАНО»
Заведующий кафедрой
к.п.н. доцент каф. Машанова А.С.
Протокол№ _____
«____»_____20___г.

«УТВЕРЖДАЮ»
Председатель УМС
к.п.н., доцент. _____Ибраев А.Д.
Протокол №_____
«____» _____20___г

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС
по учебной дисциплине: «Алгоритмизация и методы программирование»

УМК составил:
(ст. преподаватель Тургунбаев Э.К).

БИШКЕК -2024

СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

1. **Пояснительная записка**
   - Краткая характеристика дисциплины;
   - Цель и задачи дисциплины, ее место в учебном процессе;
   - Результаты освоения программы.
2. **Тематический план изучения курса**
3. **Структура и содержание курса**
4. **Ресурсное обеспечение курса**

Пояснительная записка

Актуальность изучения учебной дисциплины «**Алгоритмизация и методы программирование**» в программе обучения специалистов технического профиля связана с современными задачами, решаемыми обществом. Данный курс относится к числу курсов, призванных заложить фундамент общей подготовки будущих специалистов в области информатики и информационных систем.

В курсе, призванном способствовать формированию достаточно четкого представления об основах высокоуровневого программирования, рассматриваются фундаментальные вопросы, связанные с современными технологиями программирования, эволюцией программного обеспечения, парадигмами процедурного, модульного и объектно-ориентированного программирования.

Основное внимание уделяется объектно-ориентированному программированию (ООП). Рассматриваются такие основополагающие понятия и конструкции ООП как классы и объекты, инкапсуляция; наследование и полиморфизм, события и компоненты, визуальные технологии проектирования.

Цель дисциплины: формирование представления об основных концепциях программирования;

- приобретение навыков визуального программирования;
- приобретение навыков отладки и тестирования программного обеспечения.

По окончании изучения курса "**Алгоритмизация и методы программирование**" студент должен иметь представление о:

- современных технологиях программирования;
- высокоуровневых средах и языках программирования и возможных областях их применения;
- применении методологии высокоуровневого программирования для решении широкого круга управленческих и иных задач.

По окончании изучения курса "Объектно-ориентированное программирование" студент должен знать:

- порядок работы с высокоуровневыми средами визуального программирования;
- возможности высокоуровневых сред визуального программирования.

Задачи курса применительно специальности:

- в результате изучения дисциплины слушатель должен иметь представление о тенденциях и перспективах развития современных информационных технологий;
- ориентироваться на смену поколений компьютерных систем и информационных технологий;
- работать в различных операционных системах и оболочках;
- работать с текстовым, графическим редакторами и электронными таблицами;
- уметь выбирать программное обеспечение для решения разного рода задач;
- осуществлять выбор архитектуры и комплексирования аппаратных средств информационных систем и поиск необходимой информации в Интернет, научной и периодической литературе.

Тематический план изучения курса

№ п/п	Название темы	Распределение видов работ						Литература
		Лек	Прак	Кол-во часов	СРС	Кол-во часов	График приема СРС	
1	Тема 1. Числа и строки как типы данных в Python. Переменные	2	2	4	Работа с учебниками, (анализ литературы).	4	До 2 модуля	Осн.1,5. Доп 8,9
2	Тема 2. Вывод данных. Функция print()	4	2	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	4	До 2 модуля	Осн.3,7. Доп.11,18
3	Тема 3. Ввод данных. Функция input()	2	2	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	6	До 2 модуля	Осн.1,6. Доп.19,29
4	Тема 4. Логические выражения и операторы в Python	2	2	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	6	До 2 модуля	Осн.10,13 Доп.21,23
5	Тема 5. Ветвление. Условный оператор if-else в Python	4	2	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	6	До 2 модуля	Осн.12,17. Доп.57,14
6	Тема 6. Исключения и их обработка в Python. Оператор try-except	4	2	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	4	До сессии	Осн.6,4 Доп.9,12
7	Тема 7. Множественное ветвление: if-elif-else. Оператор match в Python	2	4	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	6	До сессии	Осн.3, 14 Доп. 12,13
8	Тема 8. Оператор match-case в Python	2	2	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	6	До сессии	Осн.5,6 Доп.21,23
9	Тема 9. Циклы в программировании. Цикл while в Python	2	4	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	6	До сессии	Осн.4,9 Доп.57,60
10	Тема 10. Функции в программировании. Определение и вызов функций в Python	2	2	6	Работа с программой PyCharm 2024.3.1.1 (Professional Edition)	4	До сессии	Осн.6,15 Доп.22,19
11	Тема 11. Локальные и глобальные переменные в Python	2	4	4	Демонстрация программного продукта	4	До сессии	Осн. 11,20 Доп.17,28
12	Тема 12. Передача значений из функций в Python. Оператор return	4	4	2	Презентации Power point	4	До сессии	Осн.3,11 Доп.22,19
	Всего	**32**	**32**	**64**		**56**		

Структура и содержание курса

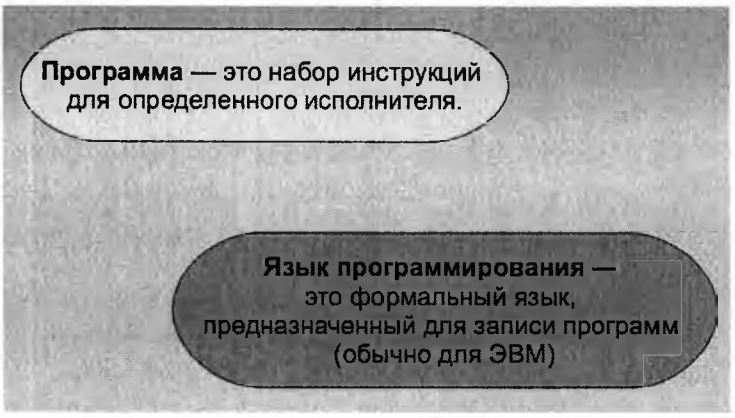
Понятие программы. Краткая история языков программирования. Трансляторы

Программа. Язык программирования

Программу можно представить как набор последовательных команд, то есть *алгоритм*, для объекта, то есть *исполнителя*, который должен их выполнить для достижения определенной цели.

Так можно условно запрограммировать человека, составив для него инструкцию "как приготовить оладьи", а он начнет ей следовать. При этом инструкция, она же программа, для человека будет написана на так называемом естественном языке, например, русском или английском.

Все же программируют не людей, а вычислительные машины, используя при этом специальные языки. Необходимость в особых языках связана с тем, что машины не в состоянии "понимать" наши, то есть человеческие, естественные для нас. Инструкции для машин пишут на *языках программирования*, которые *характеризуются формальностью*, то есть синтаксической однозначностью (например, в них нельзя менять местами определенные слова) и ограниченностью (имеют строго определенный набор слов и символов).



Основные этапы исторического развития языков программирования

Первые программы писались на **машинном языке**, так как для ЭВМ того времени еще не существовало развитого программного обеспечения, а машинный язык – это единственный способ взаимодействия с аппаратным обеспечением компьютера, так называемым "железом".

Каждую команду машинного языка непосредственно выполняет то или иное электронное устройство. Данные и команды записывали в цифровом виде, например, в шестнадцатеричной или двоичной системах счисления. Человеку воспринимать написанную таким образом программу сложно. Кроме того, даже небольшая программа состояла из множества строк кода. Ситуация осложнялась еще и тем, что каждая вычислительная машина понимает лишь свой машинный язык.

Людям, в отличие от машин, более понятны слова, чем наборы цифр. Стремление человека оперировать словами, а не цифрами привело к появлению **ассемблеров**. Это языки, в которых вместо численного обозначения команд и областей памяти используются словесно-буквенные.

Однако машина по-прежнему не может понимать слова. Необходим какой-нибудь переводчик на ее родной машинный язык. Поэтому, начиная со времен ассемблеров, под каждый язык программирования создаются *трансляторы* – специальные программы, преобразующие

программный код с языка программирования в машинный код. Ассемблеры на сегодняшний день продолжают использоваться. В системном программировании с их помощью создаются низкоуровневые интерфейсы операционных систем, компоненты драйверов.

После ассемблеров наступил расцвет языков так называемого **высокого уровня**. Для них потребовалось разрабатывать более сложные трансляторы, так как языки высокого уровня куда больше удобны для человека, чем для вычислительной машины.

В отличие от ассемблеров, которые остаются привязанными к своим типам машин, языки высоко уровня *обладают переносимостью*. Это значит, что, написав один раз программу, программист без последующего редактирования может выполнить ее на любом компьютере, если на нем установлен соответствующий транслятор. Программа-транслятор для данной ЭВМ при трансляции исходного кода сама адаптирует его под эту ЭВМ.

Следующим значимым шагом было появление **объектно-ориентированных языков**, что в первую очередь связано с усложнением разрабатываемых программ. С помощью таких языков программист как бы управляет виртуальными объектами. Мыслить в рамках объектов-сущностей, описывать их взаимодействие, обобщать объекты в классы и устанавливать между ними наследственные связи, – все это делает программу по-своему похожей на реальный мир, на то, как его воспринимает человек.

На сегодняшний день в большинстве случаев реализация крупных проектов осуществляется с помощью объектно-ориентированных возможностей языков. Хотя существуют и другие современные парадигмы программирования, поддерживаемые другими или теми же языками.

Разнообразие языков программирования

В настоящее время существует множество различающихся и похожих между собой языков программирования. Причина такого явления становится понятна, если представить то количество и разнообразие задач, которые на сегодняшний день решается с помощью вычислительной техники. Для решения разных задач требуются разные инструменты, то есть разные языки и подходы к программированию.

Разработка новых языков программирования, обладающих теми или иными преимуществами, велась как в прошлом, так и ведется сейчас. Эволюционируют, подстраиваясь под запросы нового времени, и старые языки программирования.

Все многообразие языков можно классифицировать по разным критериям. Например, по типу решаемых задач (языки системного или прикладного назначения, языки для web-разработки, организации баз данных, разработки мобильных приложений). Среди наиболее

популярных на сегодняшний день можно отметить Java, C, C++, C#, JavaScript, PHP, в том числе Python, изучению базовых основ которого посвящен данный курс.

Трансляция

Ранее было сказано, что для перевода кода с языка программирования высокого уровня на машинный язык требуется специальная программа – транслятор.

Заложенный в транслятор алгоритм такого перевода сложен. При этом существует два основных способа трансляции — **компиляция** программы или ее **интерпретация**.

При компиляции весь *исходный программный код* (тот, который пишет программист) сразу переводится в машинный. Создается так называемый отдельный *исполняемый файл*, который никак не связан с исходным кодом. Выполнение исполняемого файла обеспечивается операционной системой (ОС). После того как получен исполняемый файл, для его чтения транслятор уже не нужен.

При интерпретации выполнение кода происходит последовательно (условно можно сказать, строка за строкой). Грубо говоря, операционная система взаимодействует с интерпретатором, а не с файлом, содержащим программный код. Интерпретатор же, прочитав очередную часть исходного кода, переводит его в машинный (или не совсем машинный, но "понятный" для ОС) и "отдает" его ОС. ОС исполняет этот код и ждет следующей "подачки" от интерпретатора. Питон именно такой язык. Он интерпретируемый язык программирования.

Выполнение откомпилированной программы происходит быстрее, так как она представляет собой готовый машинный код. Однако на современных компьютерах снижение скорости выполнения при интерпретации обычно не заметно. Кроме того, интерпретируемые языки обладают рядом преимуществ, среди которых отсутствие подготовительных действий для исполнения программы, что может быть важным для тех, кто только начинает изучать программирование.

Знакомство с Python

Краткая историческая справка

Язык программирования Python был создан к 1991 году голландцем Гвидо ван Россумом. Свое имя – Пайтон (или Питон) – получил от названия телесериала, а не пресмыкающегося. После того как Россум разработал язык, он выложил его в Интернет, где сообщество программистов присоединилось к его улучшению.

Python активно развивается и сейчас. Часто выходят новые версии. Раньше поддерживались две отдельные ветки языка: Python 2.x и Python 3.x. Здесь английской буквой "x" обозначается конкретный релиз. Между вторым и третьим Питоном есть небольшая разница.

В настоящее время поддержка Python 2 прекращена.

Официальный сайт языка – https://www.python.org.

Особенности языка Python

Python – интерпретируемый язык программирования. Это значит, что исходный код частями преобразуется в машинный в процессе его чтения специальной программой – интерпретатором.

Python характеризуется ясным синтаксисом. Читать код на нем легче, чем на других языках программирования, так как в Питоне мало используются такие вспомогательные синтаксические элементы как скобки, точки с запятыми. С другой стороны, **правила языка заставляют программистов делать отступы** для обозначения вложенных конструкций. Понятно, что хорошо оформленный текст с малым количеством отвлекающих элементов читать и понимать легче.

Python – это полноценный во многом универсальный язык программирования, используемый в различных сферах. Основная, но не единственная, поддерживаемая им парадигма, – объектно-ориентированное программирование. Однако в данном курсе мы только упомянем об объектах, а будем изучать структурное программирование, так как оно является базой. Без знания основных типов данных, ветвлений, циклов, функций нет смысла изучать более сложные парадигмы, так как в них все это используется.

Интерпретаторы Python распространяется свободно на основании лицензии совместимой с GNU General Public License.

Как писать программы на Python

Интерактивный режим

Грубо говоря, интерпретатор выполняет команды построчно. Пишешь строку, нажимаешь Enter, интерпретатор выполняет ее, наблюдаешь результат.

Это удобно, когда изучаешь особенности языка или тестируешь какую-нибудь небольшую часть кода. Ведь если работать на компилируемом языке, пришлось бы сначала создать файл с кодом на исходном языке программирования, затем передать его компилятору, получить от него исполняемый файл, только потом выполнить программу и оценить результат. К счастью, даже в случае с компилируемыми языками все эти действия может взять на себя специально предназначенная для того или иного языка среда разработки, если вы используете ее.

В операционных системах на базе ядра Linux можно программировать на Python в интерактивном режиме с помощью приложения «Терминал», в котором работает командная оболочка Bash. Здесь, чтобы запустить интерпретатор, надо выполнить команду python3 (может быть просто python).

```
pl@desk:~$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

В данном случае запустилась версия 3.10.12. Первое число 3 указывает на то, что это интерпретатор для языка программирования Python 3. Последняя строка с тремя угловыми скобками (>>>) – это приглашение для ввода команд.

Для операционных систем семейства Windows надо скачать интерпретатор с официального сайта языка (https://www.python.org/downloads/windows/). После установки он будет запускаться по ярлыку. Использовать командную оболочку здесь не требуется.

Возможности Python позволяют использовать его как калькулятор. Поскольку команды языка мы не изучали, это хороший способ протестировать интерактивный ввод команд.

Бывает, что в процессе ввода была допущена ошибка или требуется повторить ранее используемую команду. Чтобы заново не вводить строку, в консоли можно прокручивать историю команд, используя для этого стрелки вверх и вниз на клавиатуре. В среде IDLE (в Windows) для этого используются сочетания клавиш (скорее всего Alt + N и Alt + P). Чтобы выйти из интерактивного режима, следует ввести команду exit().

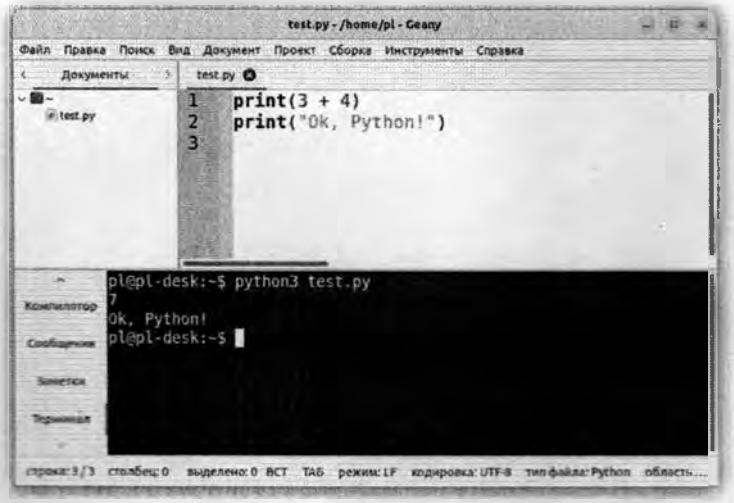
Создание скриптов

Несмотря на удобства интерактивного режима, чаще всего необходимо сохранить исходный программный код для последующего выполнения и использования. В таком случае подготавливаются файлы, которые передаются затем интерпретатору на исполнение. Файлы с кодом на Python обычно имеют расширение **.py*.

Существует множество сред разработки (IDE), в том числе созданные для программирования почти исключительно на Python. Примером такой среды является PyCharm. Однако подойдет и любой "легковесный" текстовый редактор с подсветкой синтаксиса, например, Geany или Sublime Text.

Здесь создается и сохраняется файл с кодом. Далее его можно запустить на выполнение через терминал. При этом сначала указывается интерпретатор (в данном случае python3), потом имя файла (если файл находится в другом каталоге, то указывается с адресом, или надо перейти в этот каталог с помощью команды cd оболочки Bash).

При этом в редакторе может быть установлен свой встроенный "Терминал", что упрощает работу.



Также в Geany можно просто нажать F5, что отправит файл на исполнение (терминал откроется сам, после выполнения программы и нажатия Enter закроется). Однако при этом должен быть правильно настроен вызываемый интерпретатор (пункт меню **Сборка → Установить команды сборки**).

В Windows подготовить файлы можно в той же среде IDLE. Для этого в меню следует выбрать команду **File → New Window** (Crtl + N), откроется чистое (без приглашения >>>) новое окно. Желательно сразу сохранить файл с расширением *.py*, чтобы появилась подсветка синтаксиса. После того как код будет подготовлен, снова сохраните файл. Запуск скрипта выполняется командой **Run → Run Module** (F5). После этого в окне интерактивного режима появится результат выполнения кода.

PyCharm Community. Основы работы

PyCharm – это одна из наиболее удобных сред разработки на языке Python. Существует в двух версиях:

- PyCharm Community – свободно-распространяемая версия с открытым исходным кодом.
- PyCharm Professional – проприетарная платная версия с триальным периодом.

В версии Community вы сможете программировать в основном на Python, в Professional – также на смежных языках (веб-программирование), использовать множество фреймворков.

В данном уроке мы рассмотрим создание проекта в PyCharm Community, первоначальную настройку среды и некоторые особенности работы в ней. Полную документацию смотрите на сайте разработчика данной IDE. Скачать саму среду можно по адресу https://www.jetbrains.com/ru-ru/pycharm

PyCharm не содержит самого интерпретатора Python, поэтому последний уже должен быть установлен в системе. В дистрибутивах Linux обычно это так и есть: пакет интерпретатора Python устанавливается вместе с операционной системой. Пользователи Windows, если еще не сделали этого, могут скачать интерпретатор Питона с официального сайта: https://www.python.org/downloads/

В Linux установку PyCharm Community лучше выполнить с помощью менеджера пакетов вашей операционной системы. Если такой возможности нет или это не ваш способ, то загрузив с официального сайта JetBrains и распаковав установочный пакет, найдите в нем файл *Install***.txt* или подобный, в котором описано, что надо сделать, чтобы установить и

запустить среду разработки.

В этом случае процесс может выглядеть следующим образом:

1. Перемещаем каталог с файлами среды разработки в директорию */opt* командой

sudo mv pycharm-community-2022.3.3/ /opt/

2. Переходим в директорию *bin* только что перемещенного каталога:

cd /opt/pycharm-community-2022.3.3/bin/

3. Выполняем файл *pycharm.sh* командой ./pycharm.sh

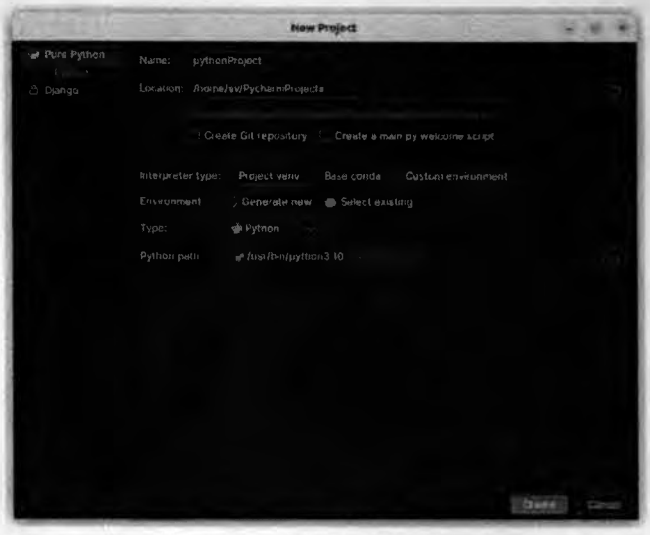
При первом запуске PyCharm будет предложено принять пользовательское соглашение, также появится окно с вопросом отправлять или нет анонимные данные о том, как вы используете продукт.

Далее появится приветственное окно, в котором среди прочего предлагается создать новый проект.

При создании проекта появляется диалоговое окно, в котором следует указать имя проекта и адрес его родительского каталога. Можно согласиться с заданными по умолчанию вариантами.

По-умолчанию предлагается использовать виртуальное окружение (выбран

вариант **Project venv**). Однако если вы только учитесь языку Питона и никаких дополнительных библиотек устанавливать не планируете, то во избежание большого количества непонятных файлов в каталоге проекта, может быть целесообразно отказаться от создания виртуальной среды Python. Для этого надо переключиться на **Custom environment**, там в качестве окружения выбрать **Select existing**, тип – **Python** и указать путь до установленного в операционной системе интерпретатора.



Далее запустится среда разработки, в ней будет открыт только что созданный проект.

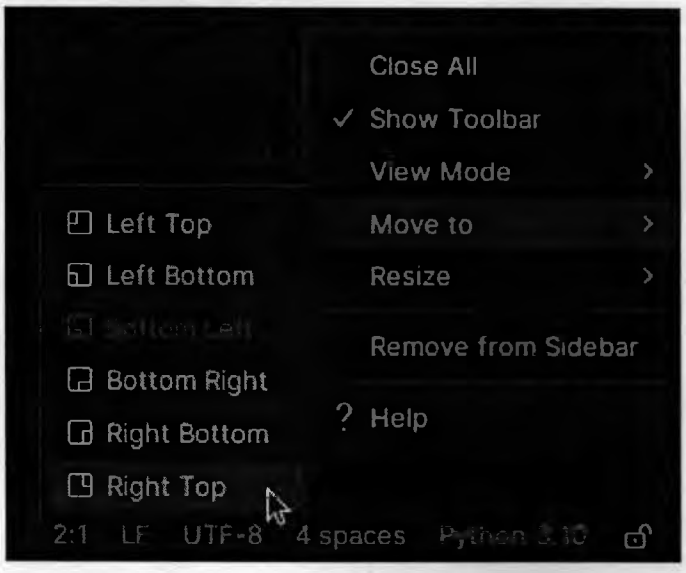
Слева на панели **Project** управляют файлами проекта. На скрине выше в папке *pythonProject* нет ни одного файла (у вас там уже может быть каталог с виртуальным окружением). Чтобы создать файл, в котором будет написана программа на Python, кликнем по этой папке правой кнопкой мыши. В контекстном меню выбираем **New → Python File**.

После этого в центральной части среды разработки появится небольшое окно, в которое вписываем имя файла.

Нажав Enter, вы увидите файл на панели **Project**. Также он будет открыт в центральной части окна PyCharm.

После того как исходный код написан, чтобы запустить программу, надо нажать Shift + F10 (возможно для первого запуска понадобится нажать Ctrl + Shift + F10). Внизу раскроется вкладка **Run**, в которой отобразиться результат выполнения.

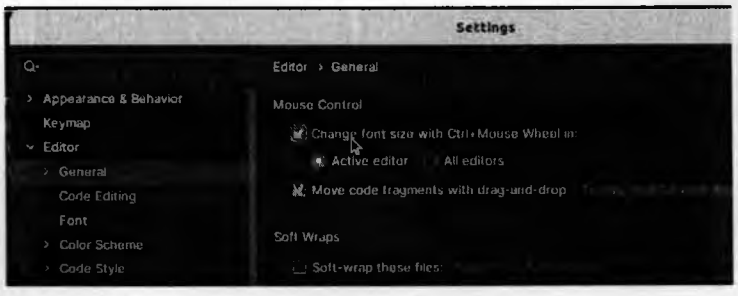
Иногда удобнее, чтобы панель выполнения программы открывалась не снизу, а, например, справа. В этом случае в настройках панели (справа значок с тремя вертикальными точками) следует выбрать **Move to → Right Top**.



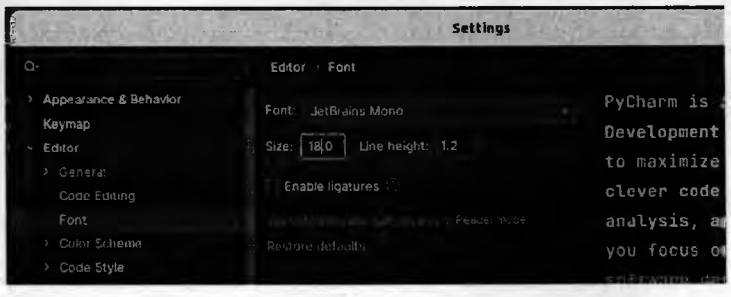
После этого интерфейс среды разработки примет такой вид:

Внешний вид среды и множество других ее свойств, поведение настраиваются в окне **Settings** (меню **File → Settings**). На скрине ниже показано, как изменить темную тему оформления PyCharm на светлую.

Бывает удобно менять размер шрифта в редакторе кода, зажав Ctrl и прокручивая колесо мыши. Чтобы воспользоваться этой возможностью в PyCharm, надо установить соответствующий флажок в разделе **Editor → General** окна настроек.

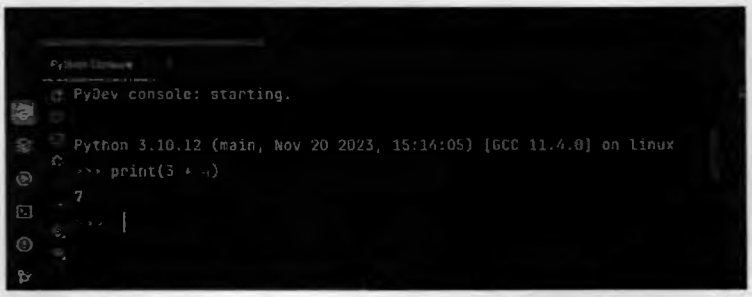


Изменить по-умолчанию заданный размер шрифта можно в разделе **Editor → Font**.



В PyCharm встроена интерактивная консоль, в которой выполняют небольшие фрагменты

кода без создания файлов.



Если в дистрибутивах Linux вы устанавливали среду разработки самостоятельно, а не через менеджер пакетов, то значок PyCharm скорее всего не появится в системном меню. И для последующего запуска среды вам снова надо будет обращаться к файлу *pycharm.sh*. Чтобы создать ярлык на приложение, в меню **Tools** поищите пункт **Create Desktop Entry…**.

Теперь рассмотрим некоторые особенности работы в PyCharm, точнее в его редакторе кода. Многие из них универсальны, характерны для других сред разработки. Так нажатие Ctrl + D дублирует строку, в которой находится курсор.

Ctrl + C копирует строку, в которой находится курсор, выделять строку при этом не надо. Потом копию можно вставить в любое место программы командой Ctrl + V. Если надо скопировать или продублировать участок в несколько строк, его следует выделить.

Выделенный участок можно сдвинуть вправо (сделать вложенным), нажав Tab. Смещение влево (на внешний уровень) выполняется комбинацией Shift + Tab.

Поднять/опустить (поменять местами с предшествующей/нижестоящей) строку или выделенный участок можно с помощью сочетаний Shift + Ctrl + стрелка вверх или стрелка вниз клавиатуры.

Ресурсное обеспечение курса

Основная литература:

1. Python для детей. Самоучитель по программированию / Джейсон Бриггс ; пер. с англ. Станислава Ломакина ; [науч. ред. Д. Абрамова]. — М. : Манн, Иванов и Фербер, 2017. — 320 с
2. Бессмертный, И. А. Системы искусственного интеллекта : учеб. пособие для СПО / И. А. Бессмертный. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2018. — 130 с.
3. Гниденко, И. Г. Технология разработки программного обеспечения : учеб. пособие для СПО / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — М. : Издательство Юрайт, 2017. — 235 с.
4. Гордеев, С. И. Организация баз данных в 2 ч. Часть 2 : учебник для вузов / С. И. Гордеев, В. Н. Волошина. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2019. — 501 с.
5. Жмудь, В. А. Моделирование замкнутых систем автоматического управления : учеб. пособие для академического бакалавриата / В. А. Жмудь. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2019. — 128 с.
6. Жорняк А.Г., Морозова Т.А., Наумченко В.П. Специфика применения языка программирования python при работе с базами данных // Научно-технический вестник Поволжья. 2022. № 5. С. 90-92
7. Завьялова О.А., Маркелов В.К. Возможности онлайн-сред программирования при обучении языку python в школе // Информатика в школе. 2022. № 3 (176). С. 75-82.
8. Зыков, С. В. Программирование. Объектно-ориентированный подход : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2019. — 155 с.

Дополнительная:

1. Косенко А.А. Противоречия объектно-ориенированного и процедурного стиля программирования в java // Научный Лидер. 2022. № 28 (73). С. 7-10.
2. Кубенский, А. А. Функциональное программирование : учебник и практикум для академического бакалавриата / А. А. Кубенский. — М. : Издательство Юрайт, 2019. — 348 с.
3. Кудрина, Е. В. Основы алгоритмизации и программирования на языке c# : учеб. пособие для СПО / Е. В. Кудрина, М. В. Огнева. — М. : Издательство Юрайт, 2019. — 322 с.
4. Кудрина, Е. В. Основы алгоритмизации и программирования на языке c# : учеб. пособие для бакалавриата и специалитета / Е. В. Кудрина, М. В. Огнева. — М. : Издательство Юрайт, 2019. — 322 с.
5. Кудрявцев, К. Я. Методы оптимизации : учеб. пособие для вузов / К. Я. Кудрявцев, А. М. Прудников. — 2-е изд. — М. : Издательство Юрайт, 2019. — 140 с.
6. Лаврищева, Е. М. Программная инженерия и технологии программирования сложных систем : учебник для вузов / Е. М. Лаврищева. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2019. — 432 с.
7. Лебедев, В. М. Программирование на vba в ms excel : учеб. пособие для академического бакалавриата / В. М. Лебедев. — М. : Издательство Юрайт, 2019. — 272 с.
8. Малявко, А. А. Формальные языки и компиляторы : учеб. пособие для вузов / А. А. Малявко. — М. : Издательство Юрайт, 2018. — 429 с
9. Мамонова, Т. Е. Информационные технологии. Лабораторный практикум : учеб. пособие для СПО / Т. Е. Мамонова. — М. : Издательство Юрайт, 2019. — 178 с.
10. Маркин, А. В. Программирование на sql в 2 ч. Часть 2 : учебник и практикум для бакалавриата и магистратуры / А. В. Маркин. — М. : Издательство Юрайт, 2019. — 292 с.
11. Меленова М.М. Особенности языка программирования python, которые необходимо учитывать при подготовке к олимпиадам по информатике // Молодой ученый. 2022. № 22 (417). С. 496-498.
12. Носарев П.Ю., Бужинская Н.В. Возможности языка программирования Python для решения задачи резюмирования текста // Тенденции развития науки и образования. 2022. № 86-2. С. 31-35.

Лекционные занятия за 1 семестр
Наименование и краткое содержание

1 модуль

Тема 1. Числа и строки как типы данных в Python. Переменные

Данные и их типы

В реальной жизни мы совершаем различные действия над окружающими нас предметами, или объектами. Мы меняем их свойства, наделяем новыми функциями. По аналогии с этим компьютерные программы также управляют объектами, только виртуальными, цифровыми. Пока не дойдем до уровня объектно-ориентированного программирования, будем называть такие объекты **данными**.

Очевидно, данные бывают разными. Часто компьютерной программе приходится работать с числами и строками. Так мы уже имели дело с числами, выполняя над ними арифметические действия. Операция сложения выполняла изменение первого числа на величину второго, а умножение увеличивало одно число в количество раз, соответствующее второму.

Числа в свою очередь также бывают разными: целыми, вещественными, могут иметь огромную величину или очень длинную дробную часть.

При знакомстве с языком программирования Python мы столкнемся с тремя типами данных:

- **целые числа** (тип int) – положительные и отрицательные целые числа, а также 0 (например, 4, 687, -45, 0).
- **числа с плавающей точкой** (тип float) – дробные, они же вещественные, числа (например, 1.45, -3.789654, 0.00453). Числа в свою очередь также бывают разными: целыми, вещественными, могут иметь огромную величину или очень длинную дробную часть.
- **строки** (тип str) — набор символов, заключенных в кавычки (например, "ball", "What is your name?", 'dkfjUUv', '6589'). Кавычки в Python могут быть одинарными или двойными; одиночный символ в кавычках также является строкой, отдельного символьного типа в Питоне нет.

Операции в программировании

Операция – это выполнение каких-либо действий над данными, которые в данном случае именуют **операндами**. Само действие выполняет **оператор** – специальный инструмент. Если бы вы выполняли операцию постройки стола, то вашими операндами были бы доска и гвоздь, а оператором – молоток.

Так в математике и программировании символ плюса является оператором операции сложения по отношению к числам. В случае строк этот же оператор выполняет операцию *конкатенации*, то есть соединения.

```
>>> 10.25 + 98.36
108.61
>>> 'Hello' + 'World'
'HelloWorld'
```

Здесь следует для себя отметить, что то, что делает оператор в операции, зависит не только от него, но и от типов данных, которыми он оперирует. Молоток в случае нападения на вас крокодила перестанет играть роль строительного инструмента. Однако в большинстве случаев операторы не универсальны. Например, знак плюса неприменим, если операндами являются, с одной стороны, число, а с другой – строка.

```
>>> 1 + 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Здесь в строке TypeError: unsupported operand type(s) for +: 'int' and 'str' интерпретатор сообщает, что произошла ошибка типа – неподдерживаемый операнд для типов int и str.

Изменение типов данных

Приведенную выше операцию все-таки можно выполнить, если превратить число 1 в строку "1". Для изменения одних типов данных в другие в языке Python предусмотрен ряд встроенных в него функций (что такое функция в принципе, вы узнаете в других уроках). Поскольку мы пока работаем только с тремя типами (int, float и str), рассмотрим вызовы соответствующих им функций – int(), float(), str().

```
>>> str(1) + 'a'
'1a'
>>> int('3') + 4
7
>>> float('3.2') + int('2')
5.2
>>> str(4) + str(1.2)
'41.2'
```

Эти функции преобразуют то, что помещается в их скобки соответственно в целое число, вещественное число или строку. Однако преобразовать можно не все:

```
>>> int('hi')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hi'
```

Здесь возникла ошибка значения (ValueError), так как передан литерал (в данном случае строка с буквенными символами), который нельзя преобразовать к числу с основанием 10. Однако функция int не такая простая:

```
>>> int('101', 2)
5
>>> int('F', 16)
15
```

Если вы знаете о различных системах счисления, то поймете, что здесь произошло.

Обратим внимание еще на одно. Данные могут называться **значениями**, а также **литералами**. Эти три понятия ("данные", "значение", "литерал") не обозначают одно и то же, но близки и нередко употребляются как синонимы. Чтобы понять различие между ними, места их употребления, надо изучить программирование глубже.

Переменные

Данные хранятся в ячейках памяти компьютера. Когда мы вводим число, оно помещается в какую-то область памяти. Но как потом узнать, куда именно? Как впоследствии обращаться к этим данными? Нужно как-то установить связь с местом хранения наших данных, запомнить его и пометить.

Раньше, при написании программ на машинном языке, обращение к ячейкам памяти осуществляли с помощью указания их регистров, то есть конкретно сообщали, куда положить данные и откуда их взять. Однако с появлением ассемблеров при обращении к данным стали использовать словесные **переменные**, что куда удобней для человека.

Механизм связи между переменными и данными может различаться в зависимости от языка программирования и типов данных. Пока достаточно запомнить, что в программе данные связываются с каким-либо именем и в дальнейшем обращение к ним происходит через это имя-переменную.

Слово "переменная" обозначает, что сущность может меняться, в ней что-то непостоянно. Действительно, вы увидите это в дальнейшем, одна и та же переменная может быть связана сначала с одними данными, а потом – с другими. То есть ее значение может меняться, она переменчива.

В программе на языке Python, как и на большинстве других языков, связь между данными и переменными устанавливается с помощью знака =. Такая операция называется **присваивание** (также говорят "присвоение"). Например, выражение sq = 4 означает, что на объект, представляющий собой число 4, находящееся в определенной области памяти, теперь ссылается переменная *sq*, и обращаться к этому объекту следует по имени *sq*.

Имена переменных могут быть любыми. Однако есть несколько общих правил их написания:

1. Желательно давать переменным осмысленные имена, говорящие о назначении данных, на которые они ссылаются.
2. Имя переменной не должно совпадать с командами языка (зарезервированными ключевыми словами).
3. Имя переменной должно начинаться с буквы или символа подчеркивания (_), но не с цифры.
4. Имя переменной не должно содержать пробелы.

Чтобы узнать значение, на которое ссылается переменная, находясь в режиме интерпретатора, достаточно ее вызвать, то есть написать имя и нажать Enter.

```
>>> sq = 4
>>> sq
4
```

Вот более сложный пример работы с переменными в интерактивном режиме:

```
>>> apples = 100
>>> eat_day = 2
>>> days = 7
>>> apples = apples - eat_day * days
>>> apples
86
```

Здесь используются три переменные: *apples*, *eat_day* и *days*. Каждой из них присваивается свое значение. Выражение apples = apples - eat_day * days сложное. Сначала выполняется подвыражение, стоящее справа от знака равенства. После этого его результат присваивается переменной *apples*, в результате чего ее старое значение (100) теряется. В подвыражении apples - eat_day * days вместо имен переменных на самом деле используются их значения, то есть числа 100, 2 и 7.

Ввод и вывод данных

Мы уже встречались с функцией print(). Она отвечает за вывод данных, по-умолчанию на экран. Если код содержится в файле, то без нее не обойтись. В интерактивном режиме в ряде случаев можно обойтись без нее.

Ввод данных в программу и их вывод важны в программировании. Без ввода программы делали бы одно и то же, исключая случаи, когда в них самих генерируются случайные значения. Вывод позволяет увидеть, использовать, куда-нибудь передать результат работы программы.

Обычно требуется, чтобы программа обрабатывала какой-то диапазон различных входных данных, которые поступают в нее из внешних источников. В качестве последних могут выступать файлы, клавиатура, сеть, выходные данные из другой программы. В свою очередь вывод данных, например, возможен в файл, по сети, в базу данных, на принтер. Однако нередко

информацию просто выводят на экран монитора.

Можно сказать, что программа – это открытая система, которая обменивается чем-либо с внешней для нее средой. Если живой организм в основном обменивается веществом и энергией, то программа – данными, информацией.

Тема 2. Вывод данных. Функция print()

Что такое функция в программировании, узнаем позже. Пока будем считать, что print() – это такая команда языка Python, которая выводит то, что в ее скобках на экран.

```
>>> print(1032)
1032
>>> print(2.34)
2.34
>>> print("Hello")
Hello
```

В скобках могут быть любые типы данных. Кроме того, количество данных может быть различным:

```
>>> print("a:", 1)
a: 1
>>> one = 1
>>> two = 2
>>> three = 3
>>> print(one, two, three)
1 2 3
```

Можно передавать в функцию print() как непосредственно литералы (в данном случае "a:" и 1), так и переменные, вместо которых будут выведены их значения. Аргументы функции (то, что в скобках), разделяются между собой запятыми. В выводе вместо запятых значения разделены пробелом.

Если в скобках стоит выражение, то сначала оно выполняется, после чего print() уже выводит результат данного выражения:

```
>>> print("hello" + " " + "world")
hello world
>>> print(10 - 2.5/2)
8.75
```

В print() предусмотрены дополнительные параметры. Например, через параметр sep можно указать отличный от пробела разделитель строк:

```
>>> print("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun", sep="-")
```

```
Mon-Tue-Wed-Thu-Fri-Sat-Sun
>>> print(1, 2, 3, sep="//")
1//2//3
```

Параметр end позволяет указывать, что делать, после вывода строки. По-умолчанию происходит переход на новую строку. Однако это действие можно отменить, указав любой другой символ или строку:

```
>>> print(10, end="")
10>>>
```

Обычно end используется не в интерактивном режиме, а в скриптах, когда несколько выводов подряд надо разделить не переходом на новую строку, а, скажем, запятыми. Сам переход на новую строку обозначается символом '\n'. Если присвоить это значение параметру end, то никаких изменений в работе функции print вы не увидите, так как это значение и так присвоено по-умолчанию:

```
>>> print(10, end='\n')
10
>>>
```

Однако, если надо отступить на одну дополнительную строку после вывода, то можно сделать так:

```
>>> print(10, end='\n\n')
10
>>>
```

В функцию print нередко передаются так называемые форматированные строки, хотя по смыслу их правильнее называть строки-шаблоны. Никакого отношения к самому print они не имеют. Когда такая строка находится в скобках print(), интерпретатор сначала согласно заданному в ней формату преобразует ее к обычной строке, после чего передает результат в print().

Форматирование может выполняться в так называемом старом стиле или с помощью строкового метода format. Старый стиль также называют Си-стилем, так как он схож с тем, как происходит вывод на экран в языке С. Рассмотрим пример:

```
>>> pupil = "Ben"
>>> old = 16
>>> grade = 9.2
>>> print("It's %s, %d. Level: %f" % (pupil, old, grade))
It's Ben, 16. Level: 9.200000
```

Здесь вместо трех комбинаций символов %s, %d, %f подставляются значения переменных *pupil*, *old*, *grade*. Буквы s, d, f обозначают типы данных – строку, целое число, вещественное число. Если бы требовалось подставить три строки, то во всех случаях

использовалось бы сочетание %s.

Хотя в качестве значения переменной *grade* было указано число 9.2, на экран оно вывелось с дополнительными нулями. Чтобы указать, сколько требуется знаков после запятой, надо перед f поставить точку, после нее указать желаемое количество знаков в дробной части:

```
>>> print("It's %s, %d. Level: %.1f" % (pupil, old, grade))
```

It's Ben, 16. Level: 9.2

Теперь посмотрим на метод format():

```
>>> print("This is a {0}. It's {1}.".format("ball", "red"))
```

This is a ball. It's red.

```
>>>
>>> print("This is a {1}. It's {0}.".format("white", "cat"))
```

This is a cat. It's white.

```
>>>
>>> print("This is a {2}. It's {0} {1}.".format("a", "number", 1))
```

This is a 1. It's a number.

В строке в фигурных скобках указаны номера данных, которые будут сюда подставлены. Далее к строке применяется метод format(). В его скобках указываются сами данные (можно использовать переменные). На нулевое место подставится первый аргумент метода format(), на место с номером 1 – второй и т. д.

На самом деле возможности метода format существенно шире, и для их изучения понадобился бы отдельный урок. Нам пока будет достаточно этого.

В новых релизах Питона появился третий способ создания форматированных строк – f-строки. Перед их открывающей кавычкой прописывается буква f. В самой строке внутри фигурных скобок записываются выражения на Python, которые исполняются, когда интерпретатор преобразует строку-шаблон в обычную.

```
>>> a = 10
>>> b = 1.33
>>> c = 'Box'
>>> print(f'qty - {a:5}, goods - {c}')
```

qty -   10, goods - Box

```
>>> print(f'price - {b + 0.2:.1f}')
```

price - 1.5

В примере число 5 после переменной *a* обозначает количество знакомест, отводимых под вывод значения переменной. В выражении b + 0.2:.1f сначала выполняется сложение, после этого значение округляется до одного знака после запятой.

Тема 3. Ввод данных. Функция input()

За ввод в программу данных с клавиатуры в Python отвечает функция input. Когда вызывается эта функция, программа останавливает свое выполнение и ждет, когда пользователь введет текст. После этого, когда он нажмет Enter, функция input() заберет введенный текст и передаст его программе, которая уже будет обрабатывать его согласно своим алгоритмам.

Если в интерактивном режиме ввести команду input(), то ничего интересного вы не увидите. Компьютер будет ждать, когда вы что-нибудь введете и нажмете Enter или просто нажмете Enter. Если вы что-то ввели, это сразу же отобразиться на экране:

```
>>> input()
Yes!
'Yes!'
```

Функция input() передает введенные данные в программу. Их можно присвоить переменной. В этом случае интерпретатор не выводит строку сразу же:

```
>>> answer = input()
No, it is not.
```

В данном случае строка сохраняется в переменной *answer*, и при желании мы можем вывести ее значение на экран:

```
>>> answer
'No, it is not.'
```

При использовании функции print() кавычки в выводе опускаются:

```
>>> print(answer)
No, it is not.
```

Куда интересней использовать функцию input() в скриптах – файлах с кодом. Рассмотрим такую программу:

```
name_user = input()
city_user = input()
print(f'Вас зовут {name_user}. Ваш город {city_user}')
```

При запуске программы, компьютер ждет, когда будет введена сначала одна строк потом вторая. Они будут присвоены переменным *name_user* и *city_user*. После этого значени этих переменных выводятся на экран с помощью форматированного вывода. Прим выполнения скрипта:

```
Арнольд
Питонск
Вас зовут Арнольд. Ваш город Питонск
```

Эта программа далека от совершенства. Откуда пользователю знать, что от него хот Чтобы не вводить человека в замешательство, для функции input предусмотрен специальн параметр-приглашение. Это приглашение выводится на экран при вызове inpu Усовершенствованная программа может выглядеть так (сразу под ней пример ее выполнени

```
name_user = input('Ваше имя: ')
city_user = input('Ваш город: ')
print(f'Вас зовут {name_user}. Ваш город {city_user}')
```

Ваше имя: Серый

Ваш город: Белый

Вас зовут Серый. Ваш город Белый

Обратите внимание, что в программу поступает строка. Даже если ввести число, функция input() все равно вернет его строковое представление. Но что делать, если надо получить число? Ответ: использовать функции преобразования типов.

```
qty = input("Сколько апельсинов? ")
price = input("Цена одного? ")
qty = int(qty)
price = float(price)
summa = qty * price
print("Заплатите", summa, "сом.")
```

Сколько апельсинов? 5

Цена одного? 35.80

Заплатите 179.0 сом.

В данном случае с помощью функций int() и float() строковые значения переменных *qty* и *price* преобразуются соответственно в целое число и вещественное число. После этого новые численные значения присваиваются тем же переменным.

Программный код можно сократить, если преобразование типов выполнить в тех же строках кода, где вызывается функция input():

```
qty = int(input("Сколько апельсинов? "))
price = float(input("Цена одного апельсина? "))
summa = qty * price
print("Заплатите", summa, "сом.")
```

Сначала выполняется функция input(). Она возвращает строку, которую функция int() или float() сразу преобразует в число. Только после этого происходит присваивание переменной, то есть она сразу получает численное значение.

Тема 4. Логические выражения и операторы в Python

Часто в реальной жизни мы соглашаемся с каким-либо утверждением или отрицаем его. Например, если вам скажут, что сумма чисел 3 и 5 больше 7, вы согласитесь, скажете: "Да, это правда". Если же кто-то будет утверждать, что сумма трех и пяти меньше семи, то вы расцените такое утверждение как ложное.

Подобные фразы предполагают только два возможных ответа – либо "да", когда выражение оценивается как правда/истина, либо "нет", когда утверждение оценивается как ошибочное/ложное. В программировании и математике **если результатом вычисления выражения может быть лишь истина или ложь, то такое выражение называется логическим**.

Например, выражение 4 > 5 является логическим, так как его результатом является либо правда, либо ложь. Выражение 4 + 5 не является логическим, так как результатом его выполнения является число.

На позапрошлом уроке мы познакомились с тремя типами данных – целыми и вещественными числами, а также строками. Сегодня введем четвертый – **логический тип данных** (тип bool). Его также называют булевым. У этого типа всего два возможных значения: **True** (правда) и **False** (ложь).

```
>>> a = True
>>> type(a)
<class 'bool'>
>>> b = False
>>> type(b)
<class 'bool'>
```

Здесь переменной *a* было присвоено значение True, после чего с помощью встроенной в Python функции type() проверен ее тип. Интерпретатор сообщил, что это переменная класса bool. Понятия "класс" и "тип данных" в данном случае одно и то же. Переменная *b* также связана с булевым значением.

В программировании False обычно приравнивают к нулю, а True – к единице. Чтобы в этом убедиться, можно преобразовать булево значение к целочисленному типу:

```
>>> int(True)
1
>>> int(False)
0
```

Возможно и обратное. Можно преобразовать какое-либо значение к булевому типу:

```
>>> bool(3.4)
True
>>> bool(-150)
True
>>> bool(0)
False
>>> bool(' ')
```

True

>>> bool('')

False

И здесь работает правило: всё, что не 0 и не пустота, является правдой.

Логические операторы

В естественном языке (например, русском), чтобы сравнивать одно с другим, мы используем слова "равно", "больше", "меньше". В языках программирования для этого есть специальные знаки, подобные тем, которые используются в математике: > (больше), < (меньше), >= (больше или равно), <= (меньше или равно), == (равно), != (не равно).

Не путайте операцию присваивания значения переменной, обозначаемую в языке Python одиночным знаком "равно", и операцию сравнения (два знака "равно"). Присваивание и сравнение – разные операции.

>>> a = **10**

>>> b = **5**

>>> a + b > **14**

True

>>> a < **14** - b

False

>>> a <= b + **5**

True

>>> a != b

True

>>> a == b

False

>>> c = a == b

>>> a, b, c

(10, 5, False)

В данном примере выражение c = a == b состоит из двух подвыражений. Сначала происходит сравнение (==) переменных *a* и *b*. После этого результат логической операции присваивается переменной *c*. Выражение a, b, c просто выводит значения переменных на экран.

Сложные логические выражения

Логические выражения типа kbyte >= 1023 являются простыми, так как в них выполняется только одна логическая операция. Однако, на практике нередко возникает необходимость в более сложных выражениях. Может понадобиться получить ответа "Да" или "Нет" в зависимости от результата выполнения двух простых выражений. Например, "на улице идет снег или дождь",

"переменная *news* больше 12 и меньше 20".

В таких случаях используются специальные операторы, объединяющие два и более простых логических выражения. Широко используются два оператора – так называемые логические И (**and**) и ИЛИ (**or**).

Чтобы получить True при использовании оператора and, необходимо, чтобы результаты обоих простых выражений, которые связывает данный оператор, были истинными. Если хотя бы в одном случае результатом будет False, то и все сложное выражение будет ложным.

Чтобы получить True при использовании оператора or, необходимо, чтобы результат хотя бы одного простого выражения, входящего в состав сложного, был истинным. В случае оператора or сложное выражение становится ложным лишь тогда, когда ложны оба составляющие его простые выражения.

Допустим, переменной *x* было присвоено значение 8 (x = 8), переменной *y* присвоили 13 (y = 13). Логическое выражение y < 15 and x > 8 будет выполняться следующим образом. Сначала выполнится выражение y < 15. Его результатом будет True. Затем выполнится выражение x > 8. Его результатом будет False. Далее выражение сведется к True and False, что вернет False.

```
>>> x = 8
>>> y = 13
>>> y < 15 and x > 8
False
```

Если бы мы записали выражение так: x > 8 and y < 15, то оно также вернуло бы False. Однако сравнение y < 15 не выполнялось бы интерпретатором, так как его незачем выполнять. Ведь первое простое логическое выражение (x > 8) уже вернуло ложь, которая, в случае оператора and, превращает все выражение в ложь.

В случае с оператором or второе простое выражение проверяется, если первое вернуло ложь, и не проверяется, если уже первое вернуло истину. Так как для истинности всего выражения достаточно единственного True, неважно по какую сторону от or оно стоит.

```
>>> y < 15 or x > 8
True
```

В языке Python есть еще унарный логический оператор not, то есть отрицание. Он превращает правду в ложь, а ложь в правду. Унарный он потому, что применяется к одному выражению, стоящему после него, а не справа и слева от него как в случае бинарных and и or.

```
>>> not y < 15
False
```

Здесь y < 15 возвращает True. Отрицая это, мы получаем False.

```
>>> a = 5
>>> b = 0
>>> not a
False
```

```
>>> not b
True
```

Число 5 трактуется как истина, отрицание истины дает ложь. Ноль приравнивается к False. Отрицание False дает True.

Тема 5. Ветвление. Условный оператор if-else в Python

Ход выполнения программы может быть *линейным*, то есть таким, когда выражения выполняются друг за другом, начиная с первого и заканчивая последним. Ни одна строка кода программы не пропускается.

Однако чаще в программах бывает не так. При выполнении кода, в зависимости от тех или иных условий, некоторые его участки могут быть опущены, в то время как другие – выполнены. Иными словами, в программе может присутствовать *ветвление*, которое реализуется **условным оператором – особой конструкцией языка программирования**.

Проведем аналогию с реальностью. Человек живет по расписанию. Можно сказать, расписание – это алгоритм для человека, его программный код, подлежащий выполнению. В расписании на 18.00 стоит поход в бассейн. Однако экземпляр биоробота класса Homo sapiens через свои рецепторы-сенсоры получает информацию, что воду из бассейна слили. Разумно было бы отменить занятие по плаванию, то есть изменить ход выполнения программы-расписания. Одним из условий посещения бассейна должно быть его функционирование, иначе должны выполняться другие действия.

Подобная нелинейность действий может быть реализована в компьютерной программе. Например, часть кода будет выполняться лишь при определенном значении конкретной переменной. В языках программирования используется приблизительно такая конструкция условного оператора:

```
if логическое_выражение {
    выражение 1;
    выражение 2;
    ...
}
```

Перевести на человеческий язык можно так: **если логическое выражение возвращает истину, то выполняются выражения внутри фигурных скобок**; если логическое выражение возвращает ложь, то код внутри фигурных скобок не выполняется. С английского "if" переводится как "если".

Конструкция if логическое_выражение называется **заголовком условного оператора**. Выражения внутри фигурных скобок – **телом условного оператора**. Тело может содержать как множество выражений, так и всего одно.

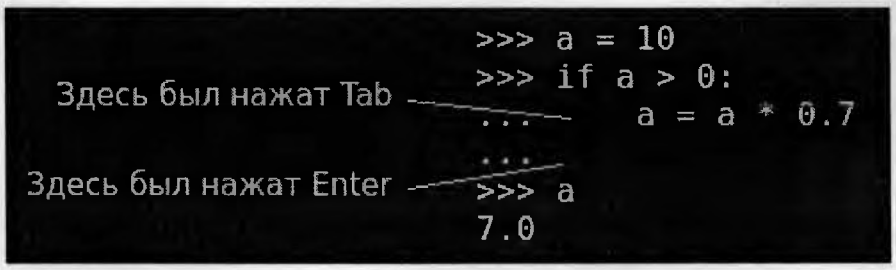
Пример использования условного оператора в языке программирования Python:

```
if n < 100:
    a = a + b
```

В Питоне вместо фигурных скобок используется двоеточие. Обособление вложенного

кода, то есть тела оператора, достигается за счет отступов. В программировании принято делать отступ равным четырем пробелам. Можно использовать клавишу табуляции (Tab) на клавиатуре.

Большинство сред программирования автоматически создают отступ, как только вы поставите двоеточие и перейдете на новую строку. Однако при работе в интерактивном режиме отступы надо добавлять вручную.



Нахождение в теле условного оператора здесь обозначается тремя точками. При создании файла со скриптом таких точек быть не должно, как и приглашения >>>.

Python считается языком с ясным синтаксисом и легко читаемым кодом. Это достигается сведением к минимуму таких вспомогательных элементов как различные скобки и точка с запятой. Для разделения выражений используется переход на новую строку, а для обозначения вложенных выражений – отступы от начала строки. В других языках данный стиль программирования также используется, но лишь для удобочитаемости кода человеком. В Питоне же такой стиль возведен в ранг синтаксического правила.

В примере выше логическим выражением является n < 100. Если оно возвращает истину, то выполнится строчка кода a = a + b. Если логическое выражение ложно, то выражение a = a + b не выполнится.

Данный пример вырван из контекста и сам по-себе не является рабочим. Полная версия программы могла бы выглядеть так:

```
a = 50
b = 10
n = 98
if n < 100:
    a = a + b
print(a)
```

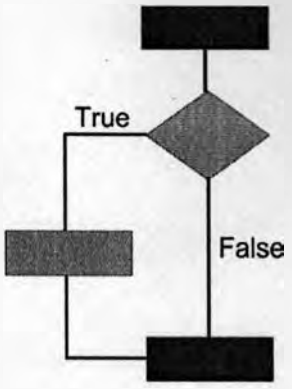
Последняя строчка кода print(a) уже не относится к условному оператору, что обозначено отсутствием перед ней отступа. Она не является вложенной в условный оператор, значит, не принадлежит ему.

Поскольку переменная *n* равна 98, а это меньше 100, то *a* станет равной 60. Это значение будет выведено на экран. Если переменная *n* изначально была бы связана, например, со значением 101, то на экран было бы выведено 50. Потому что при *n*, равной 101, логическое выражение в заголовке условного оператора вернуло бы ложь. Значит, тело не было бы выполнено, и переменная *a* не изменилась бы.

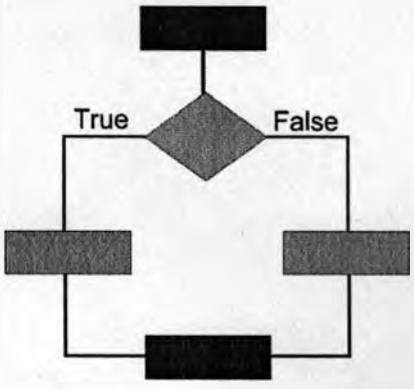
Структуру программы можно изобразить следующим образом:

Основная ветка программы выполняется всегда, а вложенный код лишь тогда, когда в темно-зеленой строчке, обозначающей заголовок условного оператора, случается истина.

Для небольших программ иногда чертят так называемые блок-схемы, отражающие алгоритм выполнения. В языке блок-схем различные части кода обозначаются своими фигурами. Так блоку последовательно выполняемых действий соответствует прямоугольник, ветвлению – ромб. Для кода выше блок-схема может выглядеть так:



Условный оператор может включать не одну ветку, а две, реализуя тем самым полноценное ветвление.



В случае возврата логическим выражением False поток выполнения программы не возвращается сразу в основную ветку. На случай False существует другой вложенный код, отличный от случая True. Другими словами, встретившись с расширенной версией условного оператора, поток выполнения программы не вернется в основную ветку, не выполнив хоть какой-нибудь вложенный код.

В языках программирования разделение на две ветви достигается с помощью добавления блока **else**, получается так называемое **if–else** (если-иначе). Синтаксис выглядит примерно так:

```
if логическое_выражение {
```

```
    выражение 1;
    выражение 2;
    ...
}
else {
    выражение 3;
    ...
}
```

Если условие при инструкции if оказывается ложным, то выполняется блок кода при инструкции else. **Ситуация, при которой бы выполнились обе ветви, невозможна.** Либо код, принадлежащий if, либо код, принадлежащий else. Никак иначе. **В заголовке else никогда не бывает логического выражения.**

Пример программы с веткой else на языке Python (под ним показаны варианты выполнения):

```
your_money = int(input('Сколько у вас монет? '))
sword = 50
helmet = 32
if sword + helmet > your_money:
    print('Вы не можете купить меч и шлем')
else:
    print('Удачный апгрейд!')
```

Сколько у вас монет? 85

Удачный апгрейд!

Сколько у вас монет? 63

Вы не можете купить меч и шлем

Следует иметь в виду, что логическое выражение при if может выглядеть нестандартно, то есть не так как a > b и тому подобное. Там может стоять просто одна переменная, число, слово True или False, а также сложное логическое выражение, когда два простых соединяются через логически and или or.

```
a = ?
if a:
    a = 1
```

Если вместо знака вопроса будет стоять 0, то с логической точки зрения это False, значит выражение в if не будет выполнено. Если *a* будет связано с любым другим числом, то оно будет расцениваться как True, и тело условного оператора выполнится. Другой пример:

```
a = 5 > 0
if a:
    print(a)
```

Здесь *a* уже связана с булевым значением. В данном случае это True. Отметим, что в выражении a = 5 > 0 присваивание выполняется после оператора сравнения, так что подвыражение 5 > 0 выполнится первым, после чего его результат будет присвоен переменной *a*. На будущее, если вы сомневаетесь в последовательности выполнения операторов, используйте скобки, например так: a = (5 > 0).

Третий пример:

```
if a > 0 and a < b:
    print(b - a)
```

Тут, чтобы вложенный код выполнился, *a* должно быть больше нуля и одновременно меньше *b*. Также в Питоне, в отличие от других языков программирования, позволительна такая сокращенная запись сложного логического выражения:

```
if 0 < a < b:
    print(b - a)
```

Тема 6. Исключения и их обработка в Python. Оператор try-except
Ошибки и исключения

В любой, особенно большой, программе могут возникать ошибки, приводящие к ее неработоспособности или к тому, что программа делает не то что должна. Причин возникновения ошибок много.

Программист может сделать ошибку в употреблении самого языка программирования. Другими словами, выразиться так, как выражаться не положено. Например, начать имя переменной с цифры или забыть поставить двоеточие в заголовке сложной инструкции. Подобные ошибки называют синтаксическими, они нарушают синтаксис и пунктуацию языка. Интерпретатор Питона, встретив ошибочное выражение, не знает как его интерпретировать. Поэтому останавливает выполнение программы и выводит соответствующее сообщение, указав на место возникновения ошибки:

```
>>> 1a = 10
  File "<stdin>", line 1
    1a = 10
     ^
SyntaxError: invalid syntax
```

В терминологии языка Python здесь возникло исключение, принадлежащее классу SyntaxError. Согласно документации Python синтаксические ошибки все-таки принято относить к ошибкам, а все остальные – к исключениям. В некоторых языках программирования не используется слово "исключение", а ошибки делят на синтаксические и семантические. Нарушение семантики обычно означает, что, хотя выражения написаны верно с точки зрения синтаксиса языка, программа не работает так, как от нее ожидалось. Для сравнения. Вы можете грамотным русским языком сказать несколько предложений, но по смыслу это будет белиберда, или вас поймут не так, как вы думали.

В Python не говорят о семантических ошибках, говорят об исключениях. Их множество. В этом уроке мы рассмотрим некоторые из них, в последующих встретимся еще с несколькими.

Если вы попытаетесь обратиться к переменной, которой не было присвоено значение, что в случае Python означает, что переменная вообще не была объявлена, она не существует, то

возникнет исключение NameError.

```
>>> a = 0
>>> print(a + b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Последнюю строку сообщения можно перевести как "Ошибка имени: имя 'b' не определено".

Если исключение возникает при выполнении кода из файла, то вместо line 1 будет указана строка, в которой оно возникло, например, line 24. Вместо <stdin> будет указано имя файла, например, test.py. В данном же случае stdin обозначает стандартный поток ввода. По-умолчанию это поток ввода с клавиатуры. Строка 1 – потому что в интерактивном режиме каждое выражение интерпретируется отдельно, как обособленная программка. Если написать выражение, состоящее из нескольких строк, то линия возникновения ошибки может быть другой:

```
>>> a = 0
>>> if a == 0:
...     print(a)
...     print(a + b)
...
0
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
NameError: name 'b' is not defined
```

Следующие два исключения, о которых следует упомянуть, и с которыми вы уже могли встретиться в предыдущих уроках, это ValueError и TypeError – ошибка значения и ошибка типа.

```
>>> int("Hi")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hi'
>>>
>>> 8 + "3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

В примере строку "Hi" нельзя преобразовать к целому числу. Возникает исключение ValueError, потому что функция int() не может преобразовать такое значение.

Число 8 и строка "3" принадлежат разным типам, операнд сложения между которыми не поддерживается. При попытке их сложить возникает исключение TypeError.

Деление на ноль вызывает исключение ZeroDivisionError:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Обработка исключений. Оператор try-except

Когда ошибки фиксируются в процессе написания программы, то программист вынужден исправить код так, чтобы их не было. Однако исключительные ситуации могут возникать уже при использовании программы. Например, ожидается ввод числа, но человек вводит букву. Попытка преобразовать ее к числу приведет к возбуждению исключения ValueError, и программа аварийно завершится.

На этот случай в языках программирования, в том числе Python, существует

специальный оператор, позволяющий перехватывать возникающие исключения и обрабатывать их так, чтобы программа продолжала работать или корректно завершала свою работу.

В Питоне такой перехват выполняет оператор try-except. "Try" переводится как "попытаться", "except" – как исключение. Словами описать его работу можно так: "Попытаться сделать вот это. Если при этом возникло исключение, то сделать что-то другое." Его конструкция (но не логика работы) похожа на условный оператор с веткой else. Рассмотрим пример:

```
n = input("Введите целое число: ")
try:
    n = int(n)
    print("Удачно")
except:
    print("Что-то пошло не так")
```

Исключительная ситуация может возникнуть в третьей строчке кода, когда значение переменной n преобразуется к целому числу. Если это невозможно, то дальнейшее выполнение выражений в теле try прекращается. В данном случае выражение print("Удачно") выполнено не будет. При этом поток выполнения программы перейдет на ветку except и выполнит ее тело.

Если в теле try исключения не возникает, то тело ветки except не выполняется.

Вот пример вывода программы, когда пользователь вводит целое число:

```
Введите целое число: 100
Удачно
```

А здесь – когда вводит не то, что ожидалось:

```
Введите целое число: AA
Что-то пошло не так
```

Есть одна проблема. Код выше обработает любое исключение. Однако в теле try могут возникать разные исключения, и у каждого из них должен быть свой обработчик. Поэтому более правильным является указание типа исключения после ключевого слова except.

```
try:
    n = input("Введите целое число: ")
    n = int(n)
    print("Все нормально. Вы ввели число", n)
except ValueError:
    print("Вы ввели не целое число")
```

Теперь если сработает тело except мы точно знаем, из-за чего возникла ошибка. Но если в теле try возникнет еще какое-нибудь исключение, то оно не будет обработано. Для него надо написать отдельную ветку except. Рассмотрим программу:

```
try:
    a = float(input("Введите делимое: "))
    b = float(input("Введите делитель: "))
    c = a / b
    print("Частное: %.2f" % c)
except ValueError:
    print("Нельзя вводить строки")
except ZeroDivisionError:
    print("Нельзя делить на ноль")
```

При ее выполнении исключения могут возникнуть в трех строчках кода: где происходит преобразование введенных значений к вещественным числам и в месте, где происходит деление. В первом случае может возникнуть ValueError, во втором – ZeroDivisionError. Каждый тип исключения обрабатывается своей веткой except.

Несколько исключений можно сгруппировать в одну ветку и обработать совместно:

```
try:
    a = float(input("Введите делимое: "))
```

```
    b = float(input("Введите делитель: "))
    c = a / b
    print("Частное: %.2f" % c)
  except (ValueError, ZeroDivisionError):
    print("Нельзя вводить строки или делить на ноль")
```

У оператора обработки исключений, кроме except, могут быть еще ветки finally и else (не обязательно обе сразу). Тело finally выполняется всегда, независимо от того, выполнялись ли блоки except в ответ на возникшие исключения или нет.
Тело else сработает, если исключений в try не было, то есть не было переходов на блоки except.

```
  try:
    n = input("Введите целое число: ")
    n = int(n)
  except ValueError:
    print("Неверный ввод")
  else:  # выполняется, когда в блоке try не возникло исключений
    print("Все нормально. Вы ввели число", n)
  finally:  # выполняется в любом случае
    print("Конец программы")
```

Примечание. В данном коде используются комментарии. В языке Python перед ними ставится знак решетки #. Комментарии в программном коде пишутся исключительно для человека и игнорируются интерпретатором или компилятором.

Посмотрите, как выполняется программа в случае возникновения исключения и без этого:

```
  Введите целое число: 4.3
  Неверный ввод
  Конец программы
  Введите целое число: 4
  Все нормально. Вы ввели число 4
  Конец программы
```

В данном уроке изложены не все особенности обработки исключений. Так в более крупных программах, содержащих несколько уровней вложенности кода, функции, модули и классы, исключения могут обрабатываться не по месту их возникновения, а передаваться дальше по иерархии вызовов.

Также исключение может возникнуть в блоке except, else или finally, и тогда им нужен собственный обработчик. Модифицируем немного предыдущую программу и специально сгенерируем исключение в теле except:

```
  try:
    n = input("Введите целое число: ")
    n = int(n)
  except ValueError:
    print("Неверный ввод")
    3 / 0
  except ZeroDivisionError:
    print("Деление на ноль")
  else:
    print("Все нормально. Вы ввели число", n)
  finally:
    print("Конец программы")
```

Поначалу может показаться, что все нормально. Исключение, выбрасываемое выражением 3 / 0 будет обработано веткой except ZeroDivisionError. Однако это не так. Эта ветка обрабатывает только исключения, возникающие в блоке try, к которому она сама относится. Вот вывод программы, если ввести не целое число:

```
Введите целое число: a
Неверный ввод
Конец программы
Traceback (most recent call last):
  File "test.py", line 15, in <module>
    n = int(n)
ValueError: invalid literal for int() with base 10: 'a'
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    3 / 0
ZeroDivisionError: division by zero
```

Мало того, что не было обработано деление на ноль, поскольку тело except ValueError неудачно завершилось, само исключение ValueError посчиталось необработанным. Решение проблемы может быть, например, таким:

```
...
except ValueError:
    print("Неверный ввод")
    try:
        3 / 0
    except ZeroDivisionError:
        print("Деление на ноль")
...
```

Здесь в тело except вложен свой внутренний обработчик исключений.

В конце отметим, что в программах, имеющих практическое значение, обязанность предвидеть возможность возникновения исключительных ситуаций и создавать код их обработки лежит на программисте. Тестировщик также должен понимать, в каких случаях могут возникать исключения. Его задача – написать код, который проверяет работу программы в разных ситуациях, в том числе крайних.

Тема 7. Множественное ветвление: if-elif-else. Оператор match в Python

Ранее мы рассмотрели работу условного оператора if. С помощью его расширенной версии **if-else** можно реализовать две отдельные ветви выполнения. Однако алгоритм программы может предполагать выбор больше, чем из двух путей, например, из трех, четырех или даже пяти. В данном случае следует говорить о необходимости множественного ветвления.

Рассмотрим конкретный пример. Допустим, в зависимости от возраста пользователя, ему рекомендуется определенный видеоконтент. При этом выделяют группы от 3 до 6 лет, от 6 до 12, от 12 до 16, 16+. Итого 4 диапазона. Как бы мы стали реализовывать задачу, имея в наборе инструментов только конструкцию **if-else**?

Самый простой ответ – последовательно проверять вхождение введенного числа-возраста в определенный диапазон с помощью следующих друг за другом условных операторов:

```
old = int(input('Ваш возраст: '))
print('Рекомендовано:', end=' ')
if 3 <= old < 6:
    print("'Заяц в лабиринте"')
if 6 <= old < 12:
```

```
    print('"Марсианин"')
if 12 <= old < 16:
    print('"Загадочный остров"')
if 16 <= old:
    print('"Поток сознания"')
```

Примечание. Названия фильмов выводятся на экран в двойных кавычках. Поэтому в программе для определения строк используются одинарные.

Предложенный код прекрасно работает, но есть одно существенное "но". Он не эффективен, так как каждый if в нем – это отдельно взятый оператор, никак не связанный с другими if. Процессор тратит время и "нервы" на обработку каждого из них, даже если в этом уже нет необходимости. Например, введено число 10. В первом if логическое выражение возвращает ложь, и поток выполнения переходит ко второму if. Логическое выражение в его заголовке возвращает истину, и его тело выполняется. Всё, на этом программа должна была остановиться.

Однако следующий if никак не связан с предыдущим, поэтому далее будет проверяться вхождение значения переменной *old* в диапазон от 12 до 16, в чем необходимости нет. И далее будет обрабатываться логическое выражение в последнем if, хотя уже понятно, что и там будет False.

Решить проблему избыточности проверок можно, вкладывая условные операторы друг в друга:

```
old = int(input('Ваш возраст: '))
print('Рекомендовано:', end=' ')
if 3 <= old < 6:
    print('"Заяц в лабиринте"')
else:
    if 6 <= old < 12:
        print('"Марсианин"')
    else:
        if 12 <= old < 16:
            print('"Загадочный остров"')
        else:
            if 16 <= old:
                print('"Поток сознания"')
```

Рассмотрим поток выполнения этого варианта кода. Сначала проверяется условие в первом if (он же самый внешний). Если здесь было получено True, то тело этого if выполняется, а в ветку else мы даже не заходим, так как она срабатывает только тогда, когда в условии if возникает ложь.

Если внешний if вернул False, поток выполнения программы заходит в соответствующий ему внешний else. В его теле находится другой if со своим else. Если введенное число попадает в диапазон от 6 до 12, то выполнится тело вложенного if, после чего программа завершается. Если же число не попадает в диапазон от 6 до 12, то произойдет переход к ветке else. В ее теле находится свой условный оператор, имеющий уже третий уровень вложенности.

Таким образом до последней проверки (16 <= old) интерпретатор доходит только тогда, когда все предыдущие возвращают False. Если же по ходу выполнения программы возникает True, то все последующие проверки опускаются, что экономит ресурсы процессора. Кроме того, такая логика выполнения программы более правильная.

Теперь зададимся следующим вопросом. Можно ли как-то оптимизировать код множественного ветвления и не строить лестницу из вложенных друг в друга условных операторов? Во многих языках программирования, где отступы используются только для удобства чтения программистом, но не имеют никакого синтаксического значения, часто используется подобный стиль:

```
if логическое_выражение {
    … ;
}
else if логическое_выражение {
    … ;
}
else if логическое_выражение {
    … ;
}
else {
    … ;
}
```

Может показаться, что имеется только один уровень вложенности, и появляется новое расширение для if, выглядящее как else if. Но это только кажется. На самом деле if, стоящее сразу после else, является вложенным в это else. Выше приведенная схема – то же самое, что

```
if логическое_выражение {
    … ;
}
else
    if логическое_выражение {
        … ;
    }
```

```
else
    if логическое_выражение {
        … ;
    }
    else {
        … ;
    }
```

Именно так ее "понимает" интерпретатор или компилятор. Однако считается, что человеку проще воспринимать первый вариант.

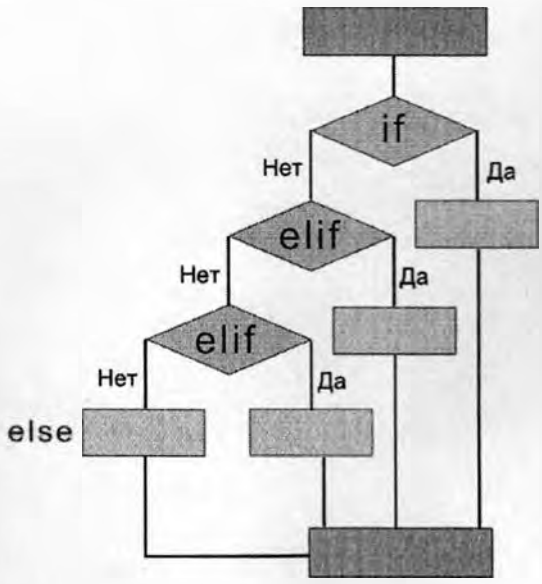
В Питоне такое поднятие вложенного if к внешнему else невозможно, потому что здесь отступы и переходы на новую строку имеют синтаксическое значение. Поэтому в язык Python встроена возможность настоящего **множественного ветвления на одном уровне вложенности, которое реализуется с помощью веток elif**.

Слово "elif" образовано от двух первых букв слова "else", к которым присоединено слово "if". Это можно перевести как "иначе если".

В отличие от else, **в заголовке elif обязательно должно быть логическое выражение** также, как в заголовке if. Перепишем нашу программу, используя конструкцию множественного ветвления:

```
old = int(input('Ваш возраст: '))
print('Рекомендовано:', end=' ')
if 3 <= old < 6:
    print('"Заяц в лабиринте"')
elif 6 <= old < 12:
    print('"Марсианин"')
elif 12 <= old < 16:
    print('"Загадочный остров"')
elif 16 <= old:
    print('"Поток сознания"')
```

Обратите внимание, в конце, после всех elif, может использоваться одна ветка else для обработки случаев, не попавших в условия ветки if и всех elif. Блок-схему полной конструкции **if-elif-…-elif-else** можно изобразить так:



Как только тело if или какого-нибудь elif выполняется, программа сразу же возвращается в основную ветку (нижний ярко-голубой прямоугольник), а все нижеследующие elif, а также else пропускаются.

Тема 8. Оператор match-case в Python

Начиная с версии 3.10 в Питоне появился оператор match, который можно использовать как аналог оператора switch, который есть в других языках. На самом деле возможности match немного шире.

В match множественное ветвление организуется с помощью веток case:

```
match имя_переменной:
   case значение_1:
      действия
   case значение_2:
      действия
   ...
```

Слова match-case можно перевести как "соответствовать случаю". То есть, если значение переменной или выражения при match соответствует значению при каком-либо case, то выполнятся действия, вложенные в этот case.

В отличие от if-elif здесь нельзя использовать логические выражения. После case должен находится литерал, конкретное значение, выражение, возвращающее однозначный результат.

Рассмотрим программу, в которой реализовать множественное ветвление с помощью match-case удобнее, чем через if-elif-else:

```
sign = input('Знак операции: ')
a = int(input('Число 1: '))
b = int(input('Число 2: '))
match sign:
   case '+':
      print(a + b)
   case '-':
      print(a - b)
   case '/':
      if b != 0:
         print(round(a / b, 2))
   case '*':
      print(a * b)
```

```
    case _:
        print('Неверный знак операции')
```

Здесь значение переменной *sign* проверяется не на вхождение в какой-либо диапазон, а на точное соответствие заданным строковым литералам. При этом в ветках case уже не надо писать sign == '+' или sign == '-', как это пришлось бы делать в программе с if-elif:

```
if sign == '+':
    print(a + b)
elif sign == '-':
    print(a - b)
elif sign == '/':
    if b != 0:
        print(round(a / b, 2))
elif sign == '*':
    print(a * b)
else:
    print('Неверный знак операции')
```

Код с match выглядит более ясным.

Оператор match языка Python не имеет ветки else. Вместо нее используется ветка case _. При одном case через оператор | можно перечислять несколько значений. Если значение переменной соответствует хотя бы одному из них, тело этого case выполнится.

```
sign = input('Знак операции: ')

match sign:
    case '+' | '-' | '*':
        a = int(input('Число 1: '))
        b = int(input('Число 2: '))
        print(eval(f'{a} {sign} {b}'))
    case _:
        print('Неверный знак операции')
```

В коде выше с помощью функции eval() переданная ей строка выполняется как выражение. Например, если были введены числа 3, 5 и знак *, то получится строка "3 * 5". Вызов eval("3 * 5") возвращает число 15.

Тема 9. Циклы в программировании. Цикл while в Python

Циклы являются такой же важной частью структурного программирования, как условные операторы. С помощью циклов можно организовать повторение выполнения участков кода. Потребность в этом возникает довольно часто. Например, пользователь последовательно вводит числа, и каждое из них требуется добавлять к общей сумме. Или нужно вывести на экран квадраты ряда натуральных чисел и тому подобные задачи.

"While" переводится с английского как "пока". Но не в смысле "до свидания", а в смысле "пока имеем это, делаем то".

Можно сказать, **while** является универсальным циклом. Он присутствует во всех языках, поддерживающих структурное программирование, в том числе в Python. Его синтаксис обобщенно для всех языков можно выразить так:

```
while логическое_выражение {
    выражение 1;
    ...
    выражение n;
}
```

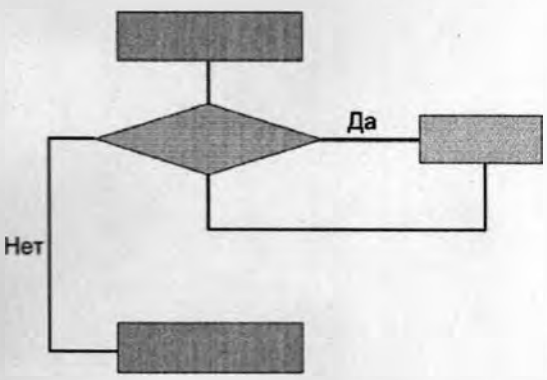
Это похоже на условный оператор if. Однако в случае циклических операторов их тела могут выполняться далеко не один раз. В случае if, если логическое выражение в заголовке возвращает истину, то тело выполняется единожды. После этого поток выполнения программы

возвращается в основную ветку и выполняет следующие выражения, расположенные ниже всей конструкции условного оператора.

В случае while, после того как его тело выполнено, поток возвращается к заголовку цикла и снова проверяет условие. Если логическое выражение возвращает истину, то тело снова выполняется. Потом снова возвращаемся к заголовку и так далее.

Цикл завершает свою работу только тогда, когда логическое выражение в заголовке возвращает ложь, то есть условие выполнения цикла больше не соблюдается. После этого поток выполнения перемещается к выражениям, расположенным ниже всего цикла. Говорят, "происходит выход из цикла".

Рассмотрите блок-схему цикла while.



На ней ярко-голубыми прямоугольниками обозначена основная ветка программы, ромбом – заголовок цикла с логическим выражением, бирюзовым прямоугольником – тело цикла.

С циклом **while** возможны две исключительные ситуации:

- Если при первом заходе в цикл логическое выражение возвращает False, то тело цикла не выполняется ни разу. Эту ситуацию можно считать нормальной, так как при определенных условиях логика программы может предполагать отсутствие необходимости в выполнении выражений тела цикла.

- Если логическое выражение в заголовке while никогда не возвращает False, а всегда остается равным True, то цикл никогда не завершится, если только в его теле нет оператора принудительного выхода из цикла (break) или вызовов функций выхода из программы – quit(), exit() в случае Python. Если цикл повторяется и повторяется бесконечное количество раз, то в программе происходит **зацикливание**. В это время она зависает и самостоятельно завершиться не может.

Вспомним наш пример из урока про исключения. Пользователь должен ввести целое число. Поскольку функция input() возвращает строку, то программный код должен преобразовать введенное к целочисленному типу с помощью функции int(). Однако, если были введены символы, не являющиеся цифрами, то возникает исключение ValueError, которое обрабатывается веткой except. На этом программа завершается.

Другими словами, если бы программа предполагала дальнейшие действия с числом (например, проверку на четность), а она его не получила, то единственное, что программа могла сделать, это закончить свою работу досрочно.

Но ведь можно просить и просить пользователя корректно вести число, пока он его не введет. Вот как может выглядеть реализующий это код:

```
n = input("Введите целое число: ")

while type(n) != int:
    try:
        n = int(n)
```

```
    except ValueError:
        print("Неправильно ввели!")
        n = input("Введите целое число: ")

if n % 2 == 0:
    print("Четное")
else:
    print("Нечетное")
```

Примечание 1. Не забываем, в языке программирования Python в конце заголовков сложных инструкций ставится двоеточие.

Примечание 2. В выражении type(n) != int с помощью функции type() проверяется тип переменной *n*. Если он не равен int, то есть значение *n* не является целым числом, а является в данном случае строкой, то выражение возвращает истину. Если же тип *n* равен int, то данное логическое выражение возвращает ложь.

Примечание 3. Оператор % в языке Python используется для нахождения остатка от деления. Так, если число четное, то оно без остатка делится на 2, то есть остаток будет равен нулю. Если число нечетное, то остаток будет равен единице.

Проследим алгоритм выполнения этого кода. Пользователь вводит данные, они имеют строковый тип и присваиваются переменной *n*. В заголовке while проверяется тип *n*. При первом входе в цикл тип *n* всегда строковый, то есть он не равен int. Следовательно, логическое выражение возвращает истину, что позволяет зайти в тело цикла.

Здесь в ветке try совершается попытка преобразования строки к целочисленному типу. Если она была удачной, то ветка except пропускается, и поток выполнения снова возвращается к заголовку while.

Теперь *n* связана с целым числом, следовательно, ее тип int, который не может быть не равен int. Он ему равен. Таким образом логическое выражение type(n) != int возвращает False, и весь цикл завершает свою работу. Далее поток выполнения переходит к оператору **if-else**, находящемуся в основной ветке программы. Здесь могло бы находиться что угодно, не обязательно условный оператор.

Вернемся назад. Если в теле try попытка преобразования к числу была неудачной, и было выброшено исключение ValueError, то поток выполнения программы отправляется в ветку except и выполняет находящиеся здесь выражения, последнее из которых просит пользователя снова ввести данные. Переменная *n* теперь имеет новое значение.

После завершения except снова проверяется логическое выражение в заголовке цикла. Оно даст True, так как значение *n* по-прежнему строка.

Выход из цикла возможен только тогда, когда значение *n* будет успешно конвертировано в число.

Рассмотрим следующий пример:

```
total = 100
i = 0
while i < 5:
    n = int(input())
    total = total - n
    i = i + 1

print("Осталось", total)
```

Сколько раз "прокрутится" цикл в этой программе, то есть сколько итераций он сделает? Ответ: 5.

1. Сначала переменная *i* равна 0. В заголовке цикла проверяется условие i < 5, и оно истинно. Тело цикла выполняется. В нем меняется значение *i*, путем добавления к нему единицы.

2. Теперь переменная *i* равна 1. Это меньше пяти, и тело цикла выполняется второй раз. В нем *i* меняется, ее новое значение 2.

3. Два меньше пяти. Тело цикла выполняется третий раз. Значение *i* становится равным трем.

4. Три меньше пяти. На этой итерации *i* присваивается 4.

5. Четыре по-прежнему меньше пяти. К *i* добавляется единица, и теперь ее значение равно пяти.

Далее начинается шестая итерация цикла. Происходит проверка условия i < 5. Но поскольку теперь оно возвращает ложь, то выполнение цикла прерывается, и его тело не выполняется.

"Смысловая нагрузка" данного цикла – это последовательное вычитание из переменной *total* вводимых чисел. Переменная *i* в данном случае играет только роль счетчика итераций цикла. В других языках программирования для таких случаев предусмотрен цикл for, который так и называется: "цикл со счетчиком". Его преимущество заключается в том, что в теле цикла не надо изменять переменную-счетчик, ее значение меняется автоматически в заголовке for.

В языке Python тоже есть цикл for. Но это не цикл со счетчиком. В Питоне он предназначен для перебора элементов последовательностей и других сложных объектов. Данный цикл и последовательности будут изучены в последующих уроках.

Для while наличие счетчика не обязательно. Представим, что надо вводить числа, пока переменная *total* больше нуля. Тогда код будет выглядеть так:

```
total = 100
while total > 0:
    n = int(input())
    total = total - n
print("Ресурс исчерпан")
```

Сколько раз здесь выполнится цикл? Неизвестно, все зависит от вводимых значений. Поэтому у цикла со счетчиком известно количество итераций, а у цикла без счетчика – нет.

Самое главное для цикла while – чтобы в его теле происходили изменения значений переменных, которые проверяются в его заголовке, и чтобы хоть когда-нибудь наступил случай, когда логическое выражение в заголовке возвращает False. Иначе произойдет зацикливание.

Примечание 1. Не обязательно в выражениях total = total - n и i = i + 1 повторять одну и ту же переменную. В Python допустим сокращенный способ записи подобных выражений: total -= n и i += 1.

Примечание 2. При использовании счетчика он не обязательно должен увеличиваться на единицу, а может изменяться в любую сторону на любое значение. Например, если надо вывести числа кратные пяти от 100 до 0, то изменение счетчика будет таким i = i - 5, или i -= 5.

Примечание 3. Для счетчика не обязательно использовать переменную с идентификатором *i*. Можно назвать переменную-счетчик как угодно. Однако так принято в программировании, что счетчики обозначают именами *i* и *j* (иногда одновременно требуются два счетчика).

Тема 10. Функции в программировании. Определение и вызов функций в Python

Функция в программировании представляет собой обособленный участок кода, который можно вызывать. Для этого к нему обращаются по имени, которым он был назван. При вызове происходит выполнение команд тела функции.

Функции можно сравнить с небольшими программками, которые сами по себе, то есть автономно, не исполняются, а встраиваются в обычную программу. Нередко их так и называют – подпрограммы. Других ключевых отличий функций от программ нет. Функции также при необходимости могут получать и возвращать данные. Только обычно они их получают не с ввода (клавиатуры, файла и др.), а из вызывающей программы. Сюда же они возвращают результат своей работы.

Существует множество встроенных в язык программирования функций. С некоторыми такими в Python мы уже сталкивались. Это print(), input(), int(), float(), str(), type(). Код их тела нам не виден, он где-то "спрятан внутри языка". Нам же предоставляется только интерфейс – имя функции, что она возвращает, информация об особенностях передачи в нее данных-аргументов.

С другой стороны, программист всегда может определять свои функции. Их называют пользовательскими. В данном случае под "пользователем" понимают программиста, а не того, кто использует программу. Разберемся, зачем нам эти функции, и как их создавать.

Предположим, надо три раза подряд запрашивать на ввод пару чисел и складывать их. С этой целью можно использовать цикл:

```
i = 0
while i < 3:
    a = int(input())
    b = int(input())
    print(a + b)
    i += 1
```

Однако, что если перед каждым запросом чисел, надо выводить надпись, зачем они нужны, и каждый раз эта надпись разная. Мы не можем прервать цикл, а затем вернуться к тому же циклу обратно. Придется отказаться от него, и тогда получится длинный код, содержащий в разных местах одинаковые участки:

```
print("Сколько бананов и ананасов для обезьян?")
a = int(input())
b = int(input())
print("Всего", a + b, "шт.")

print("Сколько жуков и червей для ежей?")
a = int(input())
b = int(input())
print("Всего", a + b, "шт.")

print("Сколько рыб и моллюсков для выдр?")
a = int(input())
b = int(input())
print("Всего", a + b, "шт.")
```

Пример исполнения программы:

Сколько бананов и ананасов для обезьян?
15
5
Всего 20 шт.
Сколько жуков и червей для ежей?
50
12
Всего 62 шт.
Сколько рыб и моллюсков для выдр?
16
8
Всего 24 шт.

Определение функций позволяет решить проблему дублирования кода в разных местах программы. Благодаря функциям можно исполнять один и тот же участок кода не сразу, а только тогда, когда он понадобится.

Определение функции. Оператор def

В языке программирования Python функции определяются с помощью оператора def.

Рассмотрим код:

```
def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a + b, "шт.")
```

Это пример определения функции. Как и другие сложные инструкции вроде условного оператора и цикла функция состоит из заголовка и тела. Заголовок оканчивается двоеточием и переходом на новую строку. Тело имеет отступ.

Ключевое слово def сообщает интерпретатору, что перед ним определение функции. За def следует имя функции. Оно может быть любым, также как и всякий идентификатор, например, переменная. В программировании весьма желательно давать всему осмысленные имена. Так в данном случае функция названа "посчитать_еду" в переводе на русский.

После имени функции ставятся скобки. В приведенном примере они пустые. Это значит, что функция не принимает никакие данные из вызывающей ее программы. Однако она могла бы их принимать, и тогда в скобках были бы указаны так называемые параметры.

После двоеточия следует тело, содержащее инструкции, которые выполняются при вызове функции. Следует различать определение функции и ее вызов. В программном коде они не рядом и не вместе. Можно определить функцию, но ни разу ее не вызвать. Нельзя вызвать функцию, которая не была определена. Определив функцию, но ни разу не вызвав ее, вы никогда не выполните ее тела.

Вызов функции

Рассмотрим полную версию программы с функцией:

```
def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")
print("Сколько бананов и ананасов для обезьян?")
count_food()
print("Сколько жуков и червей для ежей?")
count_food()
print("Сколько рыб и моллюсков для выдр?")
count_food()
```

После вывода на экран каждого информационного сообщения осуществляется вызов функции, который выглядит просто как упоминание ее имени со скобками. Поскольку в функцию мы ничего не передаем скобки опять же пустые. В приведенном коде функция вызывается три раза.

Когда функция вызывается, поток выполнения программы переходит к ее определению и начинает исполнять ее тело. После того, как тело функции исполнено, поток выполнения возвращается в основной код в то место, где функция вызывалась. Далее исполняется следующее за вызовом выражение.

В языке Python определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строка за строкой и о том, что находится ниже по течению, ему еще неизвестно. Поэтому если вызов функции предшествует ее определению, то возникает ошибка (выбрасывается исключение NameError):

```
print("Сколько бананов и ананасов для обезьян?")
count_food()
print("Сколько жуков и червей для ежей?")
count_food()
print("Сколько рыб и моллюсков для выдр?")
count_food()
def count_food():
    a = int(input())
```

```
    b = int(input())
    print("Всего", a + b, "шт.")
```
Результат:
```
Сколько бананов и ананасов для обезьян?
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    count_food()
NameError: name 'count_food' is not defined
```
Для многих компилируемых языков это не обязательное условие. Там можно определять и вызывать функцию в произвольных местах программы.

Функции придают программе структуру

Польза функций не только в возможности многократного вызова одного и того же кода из разных мест программы. Не менее важно, что благодаря им программа обретает истинную структуру. Функции как бы разделяют ее на обособленные части, каждая из которых выполняет свою конкретную задачу.

Пусть надо написать программу, вычисляющую площади разных фигур. Пользователь указывает, площадь какой фигуры он хочет вычислить. После этого вводит исходные данные. Например, длину и ширину в случае прямоугольника. Чтобы разделить поток выполнения на несколько ветвей, будем использовать оператор **if-elif-else**:
```
figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")
if figure == '1':
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a * b))
elif figure == '2':
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))
elif figure == '3':
    r = float(input("Радиус: "))
    print("Площадь: %.2f" % (3.14 * r ** 2))
else:
    print("Ошибка ввода")
```
Здесь нет никаких функций, и все прекрасно. Но напишем вариант с функциями:
```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a * b))


def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))
def circle():
    r = float(input("Радиус: "))
    print("Площадь: %.2f" % (3.14 * r ** 2))
figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")
if figure == '1':
    rectangle()
elif figure == '2':
```

```
    triangle()
elif figure == '3':
    circle()
else:
    print("Ошибка ввода")
```

Он кажется сложнее, а каждая из трех функций вызывается всего один раз. Однако из общей логики программы как бы убраны и обособлены инструкции для нахождения площадей. Программа теперь состоит из отдельных "кирпичиков Лего". В основной ветке мы можем комбинировать их как угодно. Она играет роль управляющего механизма.

Если нам когда-нибудь захочется вычислять площадь треугольника по формуле Герона, а не через высоту, то не придется искать код во всей программе (представьте, что она состоит из тысяч строк кода как реальные программы). Мы пойдем к месту определения функций и изменим тело одной из них.

Если понадобиться использовать эти функции в какой-нибудь другой программе, то мы сможем импортировать их туда, сославшись на данный файл с кодом (как это делается в Python, будет рассмотрено позже).

Тема 11. Локальные и глобальные переменные в Python

В программировании особое внимание уделяется концепции о локальных и глобальных переменных, а также связанное с ними представление об областях видимости. Соответственно, локальные переменные видны только в локальной области видимости, которой может выступать отдельно взятая функция. Глобальные переменные видны во всей программе. "Видны" – значит, известны, доступны. К ним можно обратиться по имени и получить связанное с ними значение.

К глобальной переменной можно обратиться из локальной области видимости. К локальной переменной нельзя обратиться из глобальной области видимости, потому что локальная переменная существует только в момент выполнения тела функции. При выходе из нее, локальные переменные исчезают. Компьютерная память, которая под них отводилась, освобождается. Когда функция будет снова вызвана, локальные переменные будут созданы заново.

Вернемся к нашей программе из прошлого урока, немного упростив ее для удобства:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a * b))
def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))


figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
```

Сколько здесь переменных? Какие из них являются глобальными, а какие – локальными?

Здесь пять переменных. Глобальной является только *figure*. Переменные *a* и *b* из функции rectangle, а также *a* и *h* из triangle – локальные. При этом локальные переменные с одним и тем же идентификатором *a*, но объявленные в разных функциях, – разные переменные.

Следует отметить, что идентификаторы rectangle и triangle, хотя и не являются именами переменных, а представляют собой имена функций, также имеют область видимости. В данном

случае она глобальная, так как функции объявлены непосредственно в основной ветке программы.

В приведенной программе к глобальной области видимости относятся заголовки объявлений функций, объявление и присваивание переменной *figure*, конструкция условного оператора.

К локальной области относятся тела функций. Если, находясь в глобальной области видимости, мы попытаемся обратиться к локальной переменной, то возникнет ошибка:

```
...
elif figure == '2':
    triangle()

print(a)
```

Пример выполнения:

1-прямоугольник, 2-треугольник: 2
Основание: 4
Высота: 5
Площадь: 10.00

```
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    print(a)
NameError: name 'a' is not defined
```

Однако мы можем обращаться из функций к глобальным переменным:

```
def rectangle():
    a = float(input("Ширина %s: " % figure))
    b = float(input("Высота %s: " % figure))
    print("Площадь: %.2f" % (a * b))
def triangle():
    a = float(input("Основание %s: " % figure))
    h = float(input("Высота %s: " % figure))
    print("Площадь: %.2f" % (0.5 * a * h))
figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
```

Пример выполнения:

1-прямоугольник, 2-треугольник: 1
Ширина 1: 6.35
Высота 1: 2.75
Площадь: 17.46

В данном случае из тел функций происходит обращение к имени *figure*, которое, из-за того, что было объявлено в глобальной области видимости, видимо во всей программе.

Наши функции не совсем идеальны. Они должны вычислять площади фигур, но выводить результат на экран им не следовало бы. Вполне вероятна ситуация, когда результат нужен в программе для каких-то дальнейших вычислений, а выводить ли его на экран – вопрос второстепенный.

Если функции не будут выводить, а только вычислять результат, то его надо где-то сохранить для дальнейшего использования. Для этого подошли бы глобальные переменные. В них можно записать результат. Напишем программу вот так:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
```

```
    result = a * b
def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    result = 0.5 * a * h
result = 0
figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
print("Площадь: %.2f" % result)
```

Итак, мы ввели в программу глобальную переменную *result* и инициировали ее нулем. В функциях ей присваивается результат вычислений. В конце программы ее значение выводится на экран. Мы ожидаем, что программа будет прекрасно работать. Однако…

```
1-прямоугольник, 2-треугольник: 2
Основание: 6
Высота: 4.5
Площадь: 0.00
```

… что-то пошло не так.

Дело в том, что в Python присвоение значения переменной совмещено с ее объявлением. (Во многих других языках это не так.) Поэтому, когда имя *result* впервые упоминается в локальной области видимости, и при этом происходит присваивание ей значения, то создается локальная переменная *result*. Это другая переменная, никак не связанная с глобальной *result*.

Когда функция завершает свою работу, то значение локальной *result* теряется, а глобальная не была изменена.

Когда мы вызывали внутри функции переменную *figure*, то ничего ей не присваивали. Наоборот, мы запрашивали ее значение. Интерпретатор Питона искал такую переменную сначала в локальной области видимости и не находил. После этого шел в глобальную и находил.

В случае с *result* он ничего не ищет. Он выполняет вычисления справа от знака присваивания, создает локальную переменную *result*, связывает ее с полученным значением.

На самом деле можно принудительно обратиться к глобальной переменной. Для этого существует команда global:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    global result
    result = a * b


def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    global result
    result = 0.5 * a * h
result = 0
figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
print("Площадь: %.2f" % result)
```

В таком варианте программа будет работать правильно.

Однако менять значения глобальных переменных в теле функции – плохая практика. В больших программах трудно отследить, где, какая функция и почему изменила их значение. Программист смотрит на исходное значение глобальной переменной и может подумать, что оно остается таким же. Сложно заметить, что какая-то функция поменяла его. Подобное может привести к логическим ошибкам.

Чтобы избавиться от необходимости использовать глобальные переменные, для функций существует возможность возврата результата своей работы в основную ветку программы. И уже это полученное из функции значение можно присвоить глобальной переменной в глобальной области видимости. То есть программист видит изменение значения переменной в основной ветке программы. Это делает программу более понятной.

Как функция принимает и возвращает данные, будет рассмотрено в следующих уроках.

Тема 12. Передача значений из функций в Python. Оператор return

Функции могут передавать какие-либо данные из своих тел в основную ветку программы. Говорят, что функция возвращает значение. В большинстве языков программирования, в том числе Python, выход из функции и передача данных в то место, откуда она была вызвана, выполняется оператором return.

Если интерпретатор Питона, выполняя тело функции, встречает return, то он "забирает" значение, указанное после этой команды, и "уходит" из функции.

```
def cylinder():
    r = float(input())
    h = float(input())
    # площадь боковой поверхности цилиндра:
    side = 2 * 3.14 * r * h
    # площадь одного основания цилиндра:
    circle = 3.14 * r**2
    # полная площадь цилиндра:
    full = side + 2 * circle
    return full
square = cylinder()
print(square)
```

Пример выполнения:

3
7
188.4

В данной программе в основную ветку из функции возвращается значение локальной переменной *full*. Не сама переменная, а ее значение, в данном случае – какое-либо число, полученное в результате вычисления площади цилиндра.

В основной ветке программы это значение присваивается глобальной переменной *square*. То есть выражение square = cylinder() выполняется так:

1. Вызывается функция cylinder().
2. Из нее возвращается значение.
3. Это значение присваивается переменной *square*.

Не обязательно присваивать результат переменной, его можно сразу вывести на экран:

```
...
print(cylinder())
```

Здесь число, полученное из cylinder(), непосредственно передается функции print(). Если мы в программе просто напишем cylinder(), не присвоив полученные данные переменной или не передав их куда-либо дальше, то эти данные будут потеряны. Но синтаксической ошибки не

будет.

В функции может быть несколько операторов return. Однако всегда выполняется только один из них. Тот, которого первым достигнет поток выполнения. Допустим, мы решили обработать исключение, возникающее на некорректный ввод. Пусть тогда в ветке except обработчика исключений происходит выход из функции без всяких вычислений и передачи значения:

```
def cylinder():
  try:
    r = float(input())
    h = float(input())
  except ValueError:
    return
  side = 2 * 3.14 * r * h
  circle = 3.14 * r**2
  full = side + 2 * circle
  return full
print(cylinder())
```

Если попытаться вместо цифр ввести буквы, то сработает return, вложенный в except. Он завершит выполнение функции, так что все нижеследующие вычисления, в том числе return full, будут опущены. Пример выполнения:

```
r
None
```

Но постойте! Что это за слово None, которое нам вернул "пустой" return? Это ничего, такой объект – "ничто". Он принадлежит классу NoneType. До этого мы знали четыре типа данных, они же классы: int, float, str, bool. Пришло время пятого.

Когда после return ничего не указывается, то по умолчанию считается, что там стоит объект None. При желании мы можете явно писать return None.

Более того. Ранее мы рассматривали функции, которые вроде бы не возвращали никакого значения, потому что в них не было оператора return. На самом деле возвращали, просто мы не обращали на него внимание, не присваивали никакой переменной и не выводили на экран. В Python всякая функция что-либо возвращает. Если в ней нет оператора return, то она возвращает None. То же самое, как если в ней имеется "пустой" return.

Возврат нескольких значений

В Питоне позволительно возвращать из функции несколько объектов, перечислив их через запятую после команды return:

```
def cylinder():
  r = float(input())
  h = float(input())
  side = 2 * 3.14 * r * h
  circle = 3.14 * r ** 2
  full = side + 2 * circle
  return side, full
s_cyl, f_cyl = cylinder()
print("Площадь боковой поверхности %.2f" % s_cyl)
print("Полная площадь %.2f" % f_cyl)
```

Из функции cylinder() возвращаются два значения. Первое из них присваивается переменной *s_cyl*, второе – *f_cyl*. Возможность такого группового присвоения – особенность Python, обычно не характерная для других языков:

```
>>> a, b, c = 10, 15, 19
>>> a
10
```

>>> b
15
>>> c
19

Фокус здесь в том, что перечисление значений через запятую (например, 10, 15, 19) создает объект типа tuple. На русский переводится как "кортеж". Это разновидность структур данных, которые будут изучены позже.

Когда же кортеж присваивается сразу нескольким переменным, то происходит сопоставление его элементов соответствующим в очереди переменным. Это называется распаковкой.

Таким образом, когда из функции возвращается несколько значений, на самом деле из нее возвращается один объект класса tuple. Перед возвратом эти несколько значений упаковываются в кортеж. Если же после оператора return стоит только одна переменная или объект, то ее/его тип сохраняется как есть.

Распаковка не является обязательной. Будет работать и так:

...
print(cylinder())

Пример выполнения:

4
3
(75.36, 175.84)

На экран выводится кортеж, о чем говорят круглые скобки. Его также можно присвоить одной переменной, а потом вывести ее значение на экран.

Дидактические материалы для контроля (самоконтроля) усвоенного материала

Примеры тестовых заданий

Тестовое задание содержит 50 вопросов. К каждому заданию даны несколько ответов, из которых только один верный. Выполнив задание, выберите верный ответ.

Вопрос 1

Что выведет код:

python
print(3 + 5 * 2)

a) 16
b) 13
c) 10

d) 11
Ответ: b) 13

Вопрос 2
Какой тип данных у переменной x после выполнения x = 3.14?
a) int
b) float
c) str
d) bool
Ответ: b) float

Вопрос 3
Что делает оператор continue в цикле?
a) Прерывает цикл полностью
b) Пропускает текущую итерацию
c) Ничего не делает
d) Вызывает ошибку
Ответ: b) Пропускает текущую итерацию

Вопрос 4
Какой будет результат:
python
x = [1, 2, 3]
print(x[1])
a) 1
b) 2
c) 3
d) Ошибка
Ответ: b) 2

Вопрос 5
Какой метод добавляет элемент в конец списка?
a) append()
b) insert()
c) extend()
d) pop()
Ответ: a) append()

Вопрос 6
Что выведет код:
python
x = "Python"
print(x[0])
a) P
b) y
c) t
d) Ошибка
Ответ: a) P

Вопрос 7
Какой оператор используется для проверки равенства?
a) =

b) ==
c) !=
d) ===
Ответ: b) ==

Вопрос 8
Что выведет код:
python
```
for i in range(3):
    print(i)
```
a) 1 2 3
b) 0 1 2
c) 0 1 2 3
d) Ошибка
Ответ: b) 0 1 2

Вопрос 9
Какой тип данных у x после x = True?
a) int
b) bool
c) str
d) float
Ответ: b) bool

Вопрос 10
Что делает функция len()?
a) Добавляет элемент
b) Удаляет элемент
c) Возвращает длину объекта
d) Изменяет объект
Ответ: c) Возвращает длину объекта

Вопрос 11
Что выведет код:
python
```
x = [1, 2, 3]
x.pop()
print(x)
```
a) [1, 2]
b) [1, 2, 3]
c) [2, 3]
d) Ошибка
Ответ: a) [1, 2]

Вопрос 12
Какой символ используется для комментариев в Python?
a) //
b) /*
c) #
d) --
Ответ: c) #

Вопрос 13
Что выведет код:
python
print("Hello" + "World")
a) Hello World
b) HelloWorld
c) Hello+World
d) Ошибка
Ответ: b) HelloWorld

Вопрос 14
Какой метод удаляет первый элемент списка?
a) remove()
b) pop(0)
c) del
d) clear()
Ответ: b) pop(0)

Вопрос 15
Что выведет код:
python
x = 5
x += 2
print(x)
a) 5
b) 7
c) 2
d) Ошибка
Ответ: b) 7

Вопрос 16
Какой тип данных у x после x = [1, 2, 3]?
a) tuple
b) list
c) set
d) dict
Ответ: b) list

Вопрос 17
Что делает функция print()?
a) Считывает ввод
b) Выводит данные на экран
c) Создает файл
d) Удаляет данные
Ответ: b) Выводит данные на экран

Вопрос 18
Что выведет код:
python
x = {1, 2, 3}

print(len(x))
a) 3
b) 2
c) 1
d) Ошибка
Ответ: a) 3

Вопрос 19
Какой оператор проверяет неравенство?
a) ==
b) !=
c) <>
d) =!
Ответ: b) !=

Вопрос 20
Что выведет код:
python
x = "abc"
print(x.upper())
a) ABC
b) abc
c) Abc
d) Ошибка
Ответ: a) ABC

Вопрос 21
Какой метод сортирует список по возрастанию?
a) sort()
b) sorted()
c) reverse()
d) shuffle()
Ответ: a) sort()

Вопрос 22
Что выведет код:
python
def f(x):
  return x * 2
print(f(3))
a) 3
b) 6
c) 9
d) Ошибка
Ответ: b) 6

Вопрос 23
Какой тип данных у x после x = (1, 2, 3)?
a) list
b) tuple
c) set

d) dict
Ответ: b) tuple

Вопрос 24
Что выведет код:
python
x = 10
if x > 5:
  print("Yes")
else:
  print("No")
a) Yes
b) No
c) Yes No
d) Ошибка
Ответ: a) Yes

Вопрос 25
Какой метод убирает все элементы из списка?
a) remove()
b) pop()
c) clear()
d) delete()
Ответ: c) clear()

Вопрос 26
Что выведет код:
python
x = "123"
print(int(x))
a) 123
b) "123"
c) 1 2 3
d) Ошибка
Ответ: a) 123

Вопрос 27
Что делает оператор %?
a) Деление
b) Возведение в степень
c) Остаток от деления
d) Умножение
Ответ: c) Остаток от деления

Вопрос 28
Что выведет код:
python
x = [1, 2, 3]
print(x[-1])
a) 1
b) 2
c) 3

d) Ошибка
Ответ: c) 3

Вопрос 29
Какой метод возвращает количество вхождений элемента в список?
a) count()
b) index()
c) find()
d) len()
Ответ: a) count()

Вопрос 30
Что выведет код:
python
x = {1: "a", 2: "b"}
print(x[1])
a) 1
b) "a"
c) "b"
d) Ошибка
Ответ: b) "a"

Вопрос 31
Какой тип данных у x после x = {1: "a"}?
a) list
b) tuple
c) dict
d) set
Ответ: c) dict

Вопрос 32
Что выведет код:
python
for i in "abc":
  print(i)
a) a b c
b) abc
c) 1 2 3
d) Ошибка
Ответ: a) a b c

Вопрос 33
Что делает функция input()?
a) Выводит текст
b) Считывает ввод пользователя
c) Создает список
d) Удаляет данные
Ответ: b) Считывает ввод пользователя

Вопрос 34
Что выведет код:
python

x = 5
print(x == 5)
a) True
b) False
c) 5
d) Ошибка
Ответ: a) True

Вопрос 35
Какой метод добавляет несколько элементов в список?
a) append()
b) extend()
c) insert()
d) add()
Ответ: b) extend()

Вопрос 36
Что выведет код:
python
x = [1, 2, 3]
del x[0]
print(x)
a) [1, 2]
b) [2, 3]
c) [1, 3]
d) Ошибка
Ответ: b) [2, 3]

Вопрос 37
Что делает оператор in?
a) Проверяет наличие элемента
b) Удаляет элемент
c) Добавляет элемент
d) Сортирует данные
Ответ: a) Проверяет наличие элемента

Вопрос 38
Что выведет код:
python
x = 2 ** 3
print(x)
a) 6
b) 8
c) 9
d) Ошибка
Ответ: b) 8

Вопрос 39
Какой метод возвращает индекс элемента в списке?
a) count()
b) index()

c) find()
d) len()
Ответ: b) index()

Вопрос 40
Что выведет код:
python
```
class Test:
    def __init__(self):
        self.x = 1
t = Test()
print(t.x)
```
a) 1
b) 0
c) None
d) Ошибка
Ответ: a) 1

Вопрос 41
Какой тип данных неизменяемый?
a) list
b) tuple
c) dict
d) set
Ответ: b) tuple

Вопрос 42
Что выведет код:
python
```
x = [1, 2, 3]
y = x
y[0] = 0
print(x)
```
a) [1, 2, 3]
b) [0, 2, 3]
c) [0, 1, 2]
d) Ошибка
Ответ: b) [0, 2, 3]

Вопрос 43
Что делает метод split()?
a) Объединяет строки
b) Разделяет строку на список
c) Удаляет символы
d) Заменяет символы
Ответ: b) Разделяет строку на список

Вопрос 44
Что выведет код:
python

x = lambda a: a + 1
print(x(5))
a) 5
b) 6
c) 4
d) Ошибка
Ответ: b) 6

Вопрос 45
Какой метод возвращает ключи словаря?
a) keys()
b) values()
c) items()
d) get()
Ответ: a) keys()

Вопрос 46
Что выведет код:
python
try:
  print(1 / 0)
except:
  print("Error")
a) 1
b) 0
c) Error
d) Ошибка
Ответ: c) Error

Вопрос 47
Что делает метод join()?
a) Разделяет строку
b) Объединяет элементы в строку
c) Удаляет элементы
d) Сортирует элементы
Ответ: b) Объединяет элементы в строку

Вопрос 48
Что выведет код:
python
x = {1, 2, 2, 3}
print(len(x))
a) 4
b) 3
c) 2
d) Ошибка
Ответ: b) 3

Вопрос 49
Какой оператор используется для логического "И"?
a) or
b) and

c) not
d) nor
Ответ: b) and

Вопрос 50
Что выведет код:
python
x = slice(1, 3)
print("abcde"[x])
a) ab
b) bc
c) cd
d) Ошибка
Ответ: b) bc

Задания для самостоятельной работы студентов

Задания для самостоятельной работы по программированию на Python (40 заданий)

Задание 1

Приветствие
Напиши программу, которая запрашивает имя пользователя и выводит приветствие: "Привет, [имя]!".

Задание 2

Площадь прямоугольника
Создай программу, которая запрашивает длину и ширину прямоугольника и выводит его площадь.

Задание 3
Числа от 1 до 10
Выведи все числа от 1 до 10 с помощью цикла while.

Задание 4
Таблица умножения
Напиши программу, которая выводит таблицу умножения для числа, введенного пользователем (от 1 до 10).

Задание 5
Сравнение чисел
Создай программу, которая запрашивает два числа и выводит, какое из них больше, или сообщение о том, что они равны.

Задание 6
Сумма списка
Напиши программу, которая создает список из 5 чисел и выводит их сумму.

Задание 7
Обратный отсчет
Создай программу, которая запрашивает число и выводит обратный отсчет от этого числа до 0.

Задание 8
Проверка четности
Напиши программу, которая запрашивает число и сообщает, четное оно или нечетное.

Задание 9
Строка в верхнем регистре
Создай программу, которая запрашивает строку и выводит её в верхнем регистре.

Задание 10
Количество символов
Напиши программу, которая запрашивает строку и выводит количество символов в ней.

Задание 11

Максимум в списке
Создай программу, которая находит максимальное число в списке из 5 элементов, введенных пользователем.

Задание 12
Гласные буквы
Напиши программу, которая считает количество гласных букв (а, е, и, о, у, ы, э, ю, я) в введенной пользователем строке.

Задание 13
Фильтр четных чисел
Создай программу, которая принимает список чисел и выводит только четные из них.

Задание 14
Словарь перевода
Создай словарь с 5 словами и их переводом на другой язык. Напиши программу, которая запрашивает слово и выводит его перевод или сообщение "Слово не найдено".

Задание 15
Квадрат числа
Напиши функцию, которая принимает число и возвращает его квадрат.

Задание 16
Сумма диапазона
Создай программу, которая считает сумму всех чисел от 1 до введенного пользователем числа.

Задание 17
Палиндром
Напиши программу, которая проверяет, является ли введенная строка палиндромом (читается одинаково с обеих сторон, например, "радар").

Задание 18
Сортировка списка
Создай программу, которая запрашивает 5 чисел, добавляет их в список и выводит отсортированный список.

Задание 19
Генератор паролей
Напиши программу, которая генерирует случайный пароль длиной 8 символов из букв и цифр. Используй модуль random.

Задание 20
Среднее значение
Создай программу, которая принимает список чисел и вычисляет их среднее арифметическое.

Задание 21
Простые числа
Напиши программу, которая проверяет, является ли введенное число простым (делится только на 1 и само себя).

Задание 22
Удаление дубликатов
Создай программу, которая принимает список и возвращает новый список без повторяющихся элементов.

Задание 23
Калькулятор с функциями
Напиши программу с функциями для сложения, вычитания, умножения и деления. Пользователь выбирает операцию и вводит два числа.

Задание 24
Обратный список
Создай программу, которая принимает список и выводит его элементы в обратном порядке.

Задание 25
Словарь частот
Напиши программу, которая принимает строку и создает словарь, показывающий, сколько раз каждый символ встречается в строке.

Задание 26
Числа Фибоначчи
Создай программу, которая выводит первые n чисел последовательности Фибоначчи (где каждое число — сумма двух предыдущих).

Задание 27
Запись в файл
Напиши программу, которая запрашивает текст у пользователя и сохраняет его в файл output.txt.

Задание 28
Чтение файла
Создай программу, которая читает содержимое файла output.txt и выводит его на экран.

Задание 29
Игра "Камень, ножницы, бумага"
Напиши игру, где пользователь вводит свой выбор (камень, ножницы или бумага), а компьютер выбирает случайный вариант. Определи победителя.

Задание 30
Список квадратов
Создай программу, которая генерирует список квадратов чисел от 1 до 10.

Задание 31
Поиск слова
Напиши программу, которая запрашивает предложение и слово, а затем сообщает, сколько раз это слово встречается в предложении.

Задание 32
Лямбда-функция
Создай лямбда-функцию, которая принимает два числа и возвращает их произведение. Продемонстрируй её работу.

Задание 33
Случайный выбор
Напиши программу, которая создает список из 10 имен и выбирает из них одно случайное имя с помощью модуля random.

Задание 34
Обработка исключений
Создай программу, которая запрашивает два числа и делит одно на другое, обрабатывая исключение деления на ноль.

Задание 35
Подсчет слов
Напиши программу, которая запрашивает предложение и выводит количество слов в нем.

Задание 36
Класс "Студент"
Создай класс Student с атрибутами name и grade. Добавь метод, который выводит информацию о студенте.

Задание 37
Слияние списков
Напиши программу, которая объединяет два списка в один и удаляет все дубликаты.

Задание 38
Генератор чисел
Создай программу, которая генерирует список из 10 случайных чисел от 1 до 100 и выводит их сумму.

Задание 39
Шифр Цезаря
Напиши программу, которая шифрует введенную строку, сдвигая каждую букву на 3 позиции вперед в алфавите (например, "a" становится "d").

Задание 40
Игра "Угадай число"
Создай игру, в которой компьютер загадывает число от 1 до 100, а пользователь пытается угадать его. После каждой попытки давай подсказку "больше" или "меньше".

Темы рефератов по программированию на Python

1. История создания Python: от Гвидо ван Россума до наших дней
   Исследование происхождения языка и его эволюции.

2. Основные особенности синтаксиса Python и их влияние на читаемость кода
   Почему Python считается простым и удобным языком?

3. Сравнение Python с другими языками программирования (C++, Java, JavaScript)
   Преимущества и недостатки Python в разных контекстах.

4. Роль Python в разработке искусственного интеллекта и машинного обучения
   Как Python стал стандартом для AI/ML?

5. Работа со списками, кортежами и словарями в Python: возможности и ограничения
   Подробный разбор структур данных.

6. Функции в Python: от простых до лямбда-функций
   Применение и примеры использования.

7. Объектно-ориентированное программирование в Python: классы и наследование
   Как Python реализует ООП?

8. Обработка исключений в Python: зачем и как это работает
   Практическое значение try/except.

9. Работа с файлами в Python: чтение, запись и управление
   Основы работы с файловой системой.

10. Модули и библиотеки Python: создание и использование
    Как расширить функциональность программы?

11. Применение Python в веб-разработке: фреймворки Django и Flask
    Обзор инструментов и их возможностей.

12. Python в анализе данных: библиотеки Pandas и NumPy
    Как Python помогает работать с большими данными?

13. Автоматизация задач с помощью Python: примеры из реальной жизни
    Скрипты для упрощения рутинной работы.

14. Разработка игр на Python с использованием Pygame
    Возможности Python в геймдеве.

15. Сравнение версий Python 2 и Python 3: что изменилось и почему
    Причины перехода и ключевые различия.

16. Алгоритмы и структуры данных в Python: реализация и оптимизация
    Примеры классических алгоритмов на Python.

17. Работа с базами данных в Python: SQLite и другие инструменты
    Как Python взаимодействует с БД?

18. Python в научных исследованиях: библиотека SciPy и её применение
    Роль языка в науке и вычислениях.

19. Интерпретатор Python: как он работает под капотом
    Технические аспекты выполнения кода.

20. Будущее Python: тренды и перспективы развития языка
    Куда движется Python в программировании будущего?

Вопросы к экзамену

1. Что такое Python и чем он отличается от других языков программирования?
2. Как объявить переменную в Python?
3. Какие основные типы данных есть в Python?
4. В чем разница между изменяемыми и неизменяемыми типами данных?
5. Как работает оператор = в Python?
6. Что такое индексация и как она применяется к строкам и спискам?
7. Как создать список и добавить в него элемент?
8. Чем отличается метод append() от extend() для списков?
9. Что такое кортеж и в чем его отличие от списка?
10. Как создать словарь и получить значение по ключу?
11. Что делает оператор in в Python?
12. Как работает цикл for в Python?
13. Чем отличается цикл while от цикла for?

14. Как написать условный оператор if с несколькими условиями?
15. Что такое функция и как её определить в Python?
16. Что такое аргументы функции и как их передать?
17. Как работает ключевое слово return в функциях?
18. Что такое лямбда-функция и где она используется?
19. Как импортировать модуль в Python?
20. Что такое PEP 8 и зачем он нужен?
21. Как обработать исключение в Python с помощью try/except?
22. Что такое файл в Python и как его открыть для чтения?
23. Как записать данные в файл?
24. Что такое генератор списка (list comprehension)?
25. Как найти длину строки или списка с помощью функции len()?
26. Что делает метод split() для строк?
27. Как объединить список строк в одну строку с помощью join()?
28. Что такое множества и как их создать?
29. Как проверить, является ли число четным?
30. Что такое рекурсия и как её реализовать в Python?
31. Как отсортировать список с помощью метода sort()?
32. Чем отличается sort() от функции sorted()?
33. Как найти максимальное и минимальное значение в списке?
34. Что такое модуль random и как с его помощью сгенерировать случайное число?
35. Как работает оператор slice для извлечения части списка?
36. Что такое объектно-ориентированное программирование в Python?
37. Как создать класс и объект в Python?
38. Что такое метод __init__ в классах?
39. Как реализовать наследование классов в Python?
40. Что такое GIL (Global Interpreter Lock) и как он влияет на многопоточность?

Глоссарий

2to3

Инструмент, который пытается преобразовать код Python 2.x в код Python 3.x, обрабатывая большинство несовместимостей, которые могут быть обнаружены при разборе исходного текста и обходе дерева разбора.

абстрактный базовый класс

Абстрактные базовые классы дополняют duck-typing, предоставляя способ определения интерфейсов, когда другие методы, такие как **hasattr()**, были бы неуклюжими или тонко ошибочными (например, с magic methods). ABC вводят виртуальные подклассы - классы, которые не наследуются от класса, но по-прежнему распознаются **isinstance()** и **issubclass()**; см. документацию модуля **abc**. Python поставляется с множеством встроенных ABC для структур

данных (в модуле **collections.abc**), чисел (в модуле **numbers**), потоков (в модуле **io**), поиска и загрузки импорта (в модуле **importlib.abc**). С помощью модуля **abc** вы можете создавать свои собственные азбуки.

аннотация

Метка, связанная с переменной, атрибутом класса или параметром функции или возвращаемым значением, используемая по соглашению как type hint.

Аннотации локальных переменных не могут быть доступны во время выполнения, но аннотации глобальных переменных, атрибутов классов и функций хранятся в специальном атрибуте **__annotations__** соответственно модулей, классов и функций.

аргумент

Аргументы присваиваются именованным локальным переменным в теле функции. Правила, регулирующие это присвоение, см. в разделе Звонки. Синтаксически, любое выражение может быть использовано для представления аргумента; оцененное значение присваивается локальной переменной.

асинхронный менеджер контекста

Объект, который управляет окружением, видимым в операторе **async with**, определяя методы **__aenter__**() и **__aexit__**(). Вводится в **PEP 492**.

асинхронный генератор

Функция, которая возвращает значение asynchronous generator iterator. Она похожа на функцию coroutine, определенную с помощью **async def**, за исключением того, что она содержит **yield** выражения для получения серии значений, используемых в цикле **async for**.

асинхронный итератор генератора

Объект, созданный функцией asynchronous generator.

Это asynchronous iterator, который при вызове методом **__anext__**() возвращает объект awaitable, который будет выполнять тело функции асинхронного генератора до следующего выражения **yield**.

асинхронная итерабельность

Объект, который может быть использован в операторе **async for**. Должен возвращать asynchronous iterator из своего метода **__aiter__**(). Введено **PEP 492**.

асинхронный итератор

Объект, реализующий методы **__aiter__**() и **__anext__**(). **__anext__** должен возвращать объект awaitable. **async for** разрешает awaitables, возвращаемые методом **__anext__**() асинхронного итератора, пока не вызовет исключение **StopAsyncIteration**. Представлен **PEP 492**.

атрибут

Значение, связанное с объектом, на которое ссылаются по имени с помощью точечных выражений. Например, если объект *o* имеет атрибут *a*, то на него ссылаются как *o.a*.

ожидаемый

Объект, который может быть использован в выражении **await**. Может быть coroutine или объектом с методом **__await__()**. См. также **PEP 492**.

двоичный файл

file object, способный читать и записывать bytes-like objects. Примерами двоичных файлов являются файлы, открытые в двоичном режиме
(**'rb'**, **'wb'** или **'rb+'**), **sys.stdin.buffer**, **sys.stdout.buffer** и экземпляры **io.BytesIO** и **gzip.GzipFile**.

заимствованная ссылка

В C API Python заимствованная ссылка - это ссылка на объект. Она не изменяет счетчик ссылок на объект. Она становится висячим указателем, если объект уничтожается. Например, сборка мусора может удалить последнюю ссылку strong reference на объект и таким образом уничтожить его.

байтоподобный объект

Объект, который поддерживает Буферный протокол и может экспортировать буфер C-contiguous. Сюда входят все объекты **bytes**, **bytearray** и **array.array**, а также многие общие объекты **memoryview**. Байтоподобные объекты можно использовать для различных операций, работающих с двоичными данными; к ним относятся сжатие, сохранение в двоичный файл и передача через сокет.

байткод

Исходный код Python компилируется в байткод, внутреннее представление программы Python в интерпретаторе CPython. Байткод также кэшируется в файлах **.pyc**, так что выполнение одного и того же файла во второй раз происходит быстрее (можно избежать перекомпиляции исходного текста в байткод). Считается, что этот «промежуточный язык» выполняется на virtual machine, который исполняет машинный код, соответствующий каждому байткоду.

обратный звонок

Функция подпрограммы, которая передается в качестве аргумента для выполнения в какой-то момент в будущем.

класс

Шаблон для создания определяемых пользователем объектов. Определения классов обычно содержат определения методов, которые работают с экземплярами класса.

переменная класса

Переменная, определенная в классе и предназначенная для изменения только на уровне класса (т.е. не в экземпляре класса).

принуждение

Неявное преобразование экземпляра одного типа в другой во время операции, в которой участвуют два аргумента одного типа. Например, **int(3.15)** преобразует число с плавающей точкой в целое **3**, но в **3+4.5** каждый аргумент имеет разный тип (один int, другой float), и оба должны быть преобразованы к одному типу, прежде чем их можно будет добавить, иначе возникнет ошибка **TypeError**.

комплексное число

Расширение привычной системы действительных чисел, в которой все числа выражаются в виде суммы действительной и мнимой частей. Мнимые числа - это действительные кратные мнимой единицы (квадратного корня из **-1**), часто записываемой **i** в математике или **j** в технике.

менеджер контекста

Объект, который управляет окружением, видимым в операторе **with**, определяя методы **__enter__()** и **__exit__()**. См. **PEP 343**.

контекстная переменная

Переменная, которая может иметь различные значения в зависимости от контекста. Это похоже на Thread-Local Storage, когда каждый поток выполнения может иметь разное значение для переменной. Однако в случае контекстных переменных в одном потоке выполнения может быть несколько контекстов, и основное использование контекстных переменных - это отслеживание переменных в параллельных асинхронных задачах.

смежные

Буфер считается непрерывным, если он либо *C-непрерывный*, либо *Fortran-непрерывный*. Нульмерные буферы являются смежными для C и Fortran. В одномерных массивах элементы должны располагаться в памяти рядом друг с другом в порядке возрастания индексов, начиная с нуля. В многомерных C-континуальных массивах последний индекс изменяется быстрее всего, если обращаться к элементам в порядке возрастания адреса в памяти. Однако в непрерывных массивах Fortran быстрее всего изменяется первый индекс.

coroutine

Короутины - это более обобщенная форма подпрограмм. Подпрограммы вводятся в одной точке и завершаются в другой. Подпрограммы можно вводить, завершать и возобновлять в разных точках. Они могут быть реализованы с помощью оператора **async def**. См. также **PEP 492**.

корутинная функция

Функция, которая возвращает объект coroutine. Функция coroutine может быть определена с помощью оператора **async def** и может содержать ключевые слова **await**, **async for** и **async with**. Они были введены оператором **PEP 492**.

CPython

Каноническая реализация языка программирования Python, распространяемая на python.org. Термин «CPython» используется, когда это необходимо, чтобы отличить эту реализацию от других, таких как Jython или IronPython.

декоратор

Функция, возвращающая другую функцию, обычно применяется как преобразование функции с использованием синтаксиса **@wrapper**. Общими примерами декораторов являются **classmethod()** и **staticmethod()**.

дескриптор

Любой объект, определяющий методы **__get__()**, **__set__()** или **__delete__()**. Когда атрибут класса является дескриптором, при поиске атрибута срабатывает специальное поведение связывания.

словарь

Ассоциативный массив, в котором произвольные ключи сопоставлены со значениями. Ключами могут быть любые объекты с методами **__hash__()** и **__eq__()**. В Perl называется хэшем.

понимание словаря

Компактный способ обработки всех или части элементов итерабельной таблицы и возврата словаря с результатами. **results = {n: n ** 2 for n in range(10)}** генерирует словарь, содержащий ключ **n**, сопоставленный со значением **n ** 2**.

просмотр словаря

Объекты, возвращаемые из **dict.keys()**, **dict.values()** и **dict.items()**, называются представлениями словаря. Они обеспечивают динамическое представление записей словаря, что означает, что когда словарь изменяется, представление отражает эти изменения. Чтобы заставить представление словаря стать полным списком, используйте **list(dictview)**.

docstring

Строковый литерал, который появляется в качестве первого выражения в классе, функции или модуле. Хотя при выполнении набора он игнорируется, он распознается компилятором и помещается в атрибут **__doc__** окружающего класса, функции или модуля. Поскольку он доступен через интроспекцию, он является каноническим местом для документации объекта.

duck-typing

Стиль программирования, который не смотрит на тип объекта, чтобы определить, имеет ли он правильный интерфейс; вместо этого метод или атрибут просто вызывается или используется («Если он выглядит как утка и крякает как утка, то это, должно быть, утка»). Подчеркивая интерфейсы, а не конкретные типы, хорошо спроектированный код улучшает свою гибкость, позволяя полиморфную замену. Утиная типизация позволяет избежать тестов, использующих **type()** или **isinstance()**.

EAFP

Проще попросить прощения, чем разрешения. Этот распространенный стиль кодирования Python предполагает существование допустимых ключей или атрибутов и отлавливает исключения, если предположение оказывается ложным. Этот чистый и быстрый стиль характеризуется наличием большого количества утверждений **try** и **except**. Эта техника контрастирует со стилем LBYL, характерным для многих других языков, таких как C.

выражение

Часть синтаксиса, которая может быть оценена до некоторого значения. Другими словами, выражение - это совокупность элементов выражения, таких как литералы, имена, доступ к атрибутам, операторы или вызовы функций, которые возвращают значение. В отличие от многих других языков, не все языковые конструкции являются выражениями. Существуют также statement, которые не могут быть использованы в качестве выражений, например **while**. Присвоения также являются утверждениями, а не выражениями.

модуль расширения

Модуль, написанный на C или C++, использующий C API Python для взаимодействия с ядром и пользовательским кодом.

f-строка

Строковые литералы с префиксом **'f'** или **'F'** обычно называют «f-строками», что является сокращением от formatted string literals.

файловый объект

Объект, предоставляющий файлово-ориентированный API (с такими методами, как **read()** или **write()**) к базовому ресурсу. В зависимости от того, как он был создан, файловый объект может опосредовать доступ к реальному файлу на диске или к другому типу устройств хранения или связи (например, стандартный ввод/вывод, буферы в памяти, сокеты, трубы и т.д.). Файловые объекты также называются *file-like objects* или *streams*.

кодирование файловой системы и обработчик ошибок

Обработчик кодирования и ошибок, используемый Python для декодирования байтов из операционной системы и кодирования Unicode в операционной системе.

Кодировка файловой системы должна гарантировать успешное декодирование всех байтов меньше 128. Если кодировка файловой системы не может обеспечить такую гарантию, функции API могут выдать ошибку **UnicodeError**.

оисковик

Объект, который пытается найти loader для импортируемого модуля.

деление пола

Математическое деление с округлением до ближайшего целого числа. Оператором деления на пол является //. Например, выражение **11 // 4** оценивается в **2** в отличие от **2.75**, возвращаемого истинным делением float. Обратите внимание, что **(-11) // 4** является **-3**, поскольку это **-2.75** округляется *вниз*.

функция

Серия утверждений, которая возвращает некоторое значение вызывающей стороне. Ему также может быть передано ноль или более arguments, которые могут быть использованы при выполнении тела. См. также parameter, method и раздел Определения функций.

аннотация функции

annotation параметра функции или возвращаемого значения.

Аннотации функций обычно используются для type hints: например, ожидается, что эта функция будет принимать два **int** аргумента, а также будет иметь **int** возвращаемое значение:

__будущее__

Модуль future statement, **from __future__ import <feature>** направляет компилятор на компиляцию текущего модуля с использованием синтаксиса или семантики, которые станут стандартными в одном из будущих выпусков Python. Модуль **__future__** документирует возможные значения *feature*. Импортировав этот модуль и оценив его переменные, вы можете узнать, когда новая функция была впервые добавлена в язык и когда она станет (или уже стала) стандартной:

генератор

Функция, которая возвращает значение generator iterator. Она выглядит как обычная функция, за исключением того, что содержит выражения **yield** для получения серии значений, которые

можно использовать в цикле for или извлекать по одному с помощью функции **next()**.

выражение генератора

Выражение, возвращающее итератор. Оно выглядит как обычное выражение, за которым следует предложение **for**, определяющее переменную цикла, диапазон и необязательное предложение **if**.

общая функция

Функция, состоящая из нескольких функций, реализующих одну и ту же операцию для разных типов. Какая реализация должна быть использована при вызове, определяется алгоритмом диспетчеризации.

глобальная блокировка интерпретатора

Механизм, используемый интерпретатором CPython для обеспечения того, что только один поток одновременно выполняет Python bytecode. Это упрощает реализацию CPython, делая объектную модель (включая критические встроенные типы, такие как **dict**) неявно защищенной от одновременного доступа. Блокировка всего интерпретатора упрощает многопоточность интерпретатора, но при этом теряется большая часть параллелизма, обеспечиваемого многопроцессорными машинами.

хэш-ориентированный pyc

Файл кэша байткода, который для определения своей валидности использует хэш, а не время последнего изменения соответствующего исходного файла. См. Аннулирование кэшированного байткода.

hashable

Объект является *хэшируемым*, если он имеет хэш-значение, которое никогда не меняется в течение его жизни (ему нужен метод **__hash__**()), и может быть сравнен с другими объектами (ему нужен метод **__eq__**()). Хешируемые объекты, которые сравниваются на равных, должны иметь одинаковое хеш-значение.

Хэшируемость делает объект пригодным для использования в качестве ключа словаря и члена набора, поскольку эти структуры данных используют хэш-значение внутри.

IDLE

Интегрированная среда разработки и обучения для Python. IDLE - это базовая среда редактора и интерпретатора, поставляемая со стандартным дистрибутивом Python.

неизменяемый

Объект с фиксированным значением. К неизменяемым объектам относятся числа, строки и кортежи. Такой объект не может быть изменен. Для хранения другого значения необходимо создать новый объект. Они играют важную роль в тех случаях, когда требуется постоянное хэш-значение, например, в качестве ключа в словаре.

путь импорта

Список мест (или path entries), в которых path based finder ищет модули для импорта. Во время импорта этот список местоположений обычно поступает из **sys.path**, но для подпакетов он может также поступать из атрибута **__path__** родительского пакета.

импортирование

Процесс, в ходе которого код Python в одном модуле становится доступным для кода Python в другом модуле.

импортер

Объект, который одновременно находит и загружает модуль; одновременно объект finder и loader.

интерактивный

Python имеет интерактивный интерпретатор, что означает, что вы можете вводить утверждения и выражения в подсказку интерпретатора, немедленно выполнять их и видеть их результаты. Просто запустите программу **python** без аргументов (возможно, выбрав ее из главного меню вашего компьютера). Это очень мощный способ тестирования новых идей или проверки модулей и пакетов (вспомните **help(x)**).

интерпретировано

Python является интерпретируемым языком, в отличие от компилируемого, хотя это различие может быть нечетким из-за наличия компилятора байткода. Это означает, что исходные файлы могут быть запущены напрямую, без явного создания исполняемого файла, который затем запускается. Интерпретируемые языки обычно имеют более короткий цикл разработки/отладки, чем компилируемые, хотя их программы обычно также выполняются медленнее. См. также interactive.

отключение интерпретатора

Когда интерпретатору Python предлагается завершить работу, он вступает в особую фазу, в которой постепенно освобождает все выделенные ресурсы, такие как модули и различные критические внутренние структуры. Он также делает несколько обращений к garbage collector. Это может вызвать выполнение кода в пользовательских деструкторах или обратных вызовах weakref. Код, выполняемый во время фазы выключения, может столкнуться с различными исключениями, поскольку ресурсы, на которые он полагается, могут больше не функционировать (распространенными примерами являются библиотечные модули или механизм предупреждений).

итерабельный

Объект, способный возвращать свои члены по одному за раз. Примерами итерабельных объектов являются все последовательные типы (такие как **list**, **str** и **tuple**) и некоторые непоследовательные типы, такие как **dict**, file objects, а также объекты любых классов, которые вы определяете методом **__iter__()** или методом **__getitem__()**, реализующим семантику Sequence.

итератор

Объект, представляющий поток данных. Повторные вызовы метода итератора **__next__()** (или передача его встроенной функции **next()**) возвращают последовательные элементы в потоке. Когда данные больше не доступны, вместо них возникает исключение **StopIteration**. В этот момент объект итератора исчерпан, и любые дальнейшие вызовы его метода **__next__()** просто снова вызывают исключение **StopIteration**.

ключевая функция

Ключевая функция или функция свертки - это вызываемая функция, которая возвращает

значение, используемое для сортировки или упорядочивания.
Например, **locale.strxfrm()** используется для создания ключа сортировки, который учитывает соглашения о сортировке, специфичные для конкретной локали.

лямбда

Анонимная встроенная функция, состоящая из одного expression, который оценивается при вызове функции. Синтаксис для создания лямбда-функции
следующий **lambda [parameters]: expression**

кодировка локали

В Unix это кодировка локали LC_CTYPE. Она может быть установлена с помощью **locale.setlocale(locale.LC_CTYPE, new_locale)**.

Python использует filesystem encoding and error handler для преобразования между именами файлов Unicode и именами байтов.

список

Встроенный в Python sequence. Несмотря на свое название, он больше похож на массив в других языках, чем на связанный список, поскольку доступ к элементам имеет значение O(1).

понимание списка

Компактный способ обработки всех или части элементов последовательности и возврата списка с результатами. **result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]** генерирует список строк, содержащих четные шестнадцатеричные числа (0x...) в диапазоне от 0 до 255.
Пункт **if** является необязательным. Если он опущен, обрабатываются все элементы в **range(256)**.

погрузчик

Объект, загружающий модуль. Он должен определять метод с именем **load_module()**. Загрузчик обычно возвращается командой finder.

магический метод

Неофициальный синоним special method.

картирование

Объект-контейнер, поддерживающий произвольный поиск ключей и реализующий методы, указанные в **Mapping** или **MutableMapping** abstract base classes. Примерами являются **dict**, **collections.defaultdict**, **collections.OrderedDict** и **collections.Counter**.

поиск мета-пути

finder, возвращаемый при поиске **sys.meta_path**. Мета-поисковики путей связаны с , но отличаются от path entry finders.

метакласс

Класс класса. Определения классов создают имя класса, словарь класса и список базовых классов. Метакласс отвечает за принятие этих трех аргументов и создание класса. Большинство объектно-ориентированных языков программирования предоставляют реализацию по умолчанию. Особенностью Python является то, что в нем можно создавать собственные метаклассы. Большинству пользователей этот инструмент никогда не понадобится, но когда

возникает необходимость, метаклассы могут обеспечить мощные и элегантные решения. Они используются для протоколирования доступа к атрибутам, добавления потокобезопасности, отслеживания создания объектов, реализации синглтонов и многих других задач.

метод

Функция, которая определена внутри тела класса. Если метод вызывается как атрибут экземпляра этого класса, он получит объект экземпляра в качестве своего первого argument (который обычно называется **self**). См. function и nested scope.

порядок разрешения метода

Порядок разрешения метода - это порядок, в котором базовые классы ищут член при поиске. Смотрите The Python 2.3 Method Resolution Order для подробностей алгоритма, используемого интерпретатором Python начиная с версии 2.3.

модуль

Объект, который служит организационной единицей кода Python. Модули имеют пространство имен, содержащее произвольные объекты Python. Модули загружаются в Python с помощью процесса importing.

спецификация модуля

Пространство имен, содержащее связанную с импортом информацию, используемую для загрузки модуля. Экземпляр **importlib.machinery.ModuleSpec**.

изменяемый

Мутабельные объекты могут изменять свое значение, но сохранять свое **id()**. См. также immutable.

именованный кортеж

Термин «именованный кортеж» применяется к любому типу или классу, который наследуется от кортежа и индексируемые элементы которого также доступны с помощью именованных атрибутов. Тип или класс может иметь и другие характеристики.

пространство имен

Место, где хранится переменная. Пространства имен реализованы в виде словарей. Существуют локальные, глобальные и встроенные пространства имен, а также вложенные пространства имен в объектах (в методах). Пространства имен поддерживают модульность, предотвращая конфликты имен. Например, функции **builtins.open** и **os.open()** различаются пространствами имен. Пространства имен также способствуют удобочитаемости и удобству сопровождения, поскольку становится ясно, какой модуль реализует ту или иную функцию. Например, написание **random.seed()** или **itertools.islice()** дает понять, что эти функции реализуются модулями **random** и **itertools** соответственно.

пакет пространства имён

Пакет **PEP 420** package, который служит только как контейнер для подпакетов. Пакеты пространства имен могут не иметь физического представления и, в частности, не похожи на regular package, поскольку у них нет файла **__init__.py**.

вложенная область

Возможность ссылаться на переменную во вложенном определении. Например, функция, определенная внутри другой функции, может ссылаться на переменные во внешней функции. Обратите внимание, что вложенные области видимости по умолчанию работают только для ссылок, но не для присваивания. Локальные переменные читаются и записываются во внутренней области видимости. Аналогично, глобальные переменные читают и записывают в глобальное пространство имен. Символ **nonlocal** позволяет записывать во внешние области видимости.

класс нового стиля

Старое название стиля классов, которое теперь используется для всех объектов класса. В ранних версиях Python только классы нового стиля могли использовать более новые, универсальные возможности Python, такие как **__slots__**, дескрипторы, свойства, **__getattribute__()**, методы класса и статические методы.

объект

Любые данные, имеющие состояние (атрибуты или значение) и определенное поведение (методы). Также является конечным базовым классом любого new-style class.

пакет

Python module, который может содержать подмодули или, рекурсивно, подпакеты. Технически, пакет - это модуль Python с атрибутом **__path__**.

вход в тракт

Одно место на import path, к которому обращается path based finder для поиска модулей для импорта.

поиск входа по пути

finder, возвращаемый вызываемой программой **sys.path_hooks** (т.е. path entry hook), которая знает, как найти модули, заданные path entry.

входной крючок

Вызываемый модуль на списке **sys.path_hook**, который возвращает path entry finder, если он знает, как найти модули на определенном path entry.

поиск пути

Один из стандартных meta path finders, который ищет модули в import path.

PEP

Python Enhancement Proposal. PEP - это проектный документ, предоставляющий информацию сообществу Python или описывающий новую функцию для Python, его процессов или среды. PEP должны содержать краткую техническую спецификацию и обоснование предлагаемых функций.

порция

Набор файлов в одном каталоге (возможно, хранящихся в zip-файле), которые вносят вклад в пакет пространства имен, как определено в **PEP 420**.

предварительный API

Временный API - это интерфейс, который был намеренно исключен из гарантий обратной

совместимости стандартной библиотеки. Хотя серьезных изменений в таких интерфейсах не ожидается, пока они помечены как временные, несовместимые с обратной совместимостью изменения (вплоть до удаления интерфейса) могут происходить, если разработчики ядра сочтут это необходимым. Такие изменения не будут вноситься беспричинно - они будут происходить только в случае обнаружения серьезных фундаментальных недостатков, которые были упущены до включения API.

Python 3000

Прозвище для линейки релизов Python 3.x (придумано давно, когда выход третьей версии был чем-то в далеком будущем). Это также сокращенное название «Py3k».

Пифонический

Идея или фрагмент кода, который точно следует наиболее распространенным идиомам языка Python, а не реализует код, используя концепции, характерные для других языков. Например, распространенной идиомой в Python является перебор всех элементов итерабельной таблицы с помощью оператора **for**. Во многих других языках нет такой конструкции, поэтому люди, незнакомые с Python, иногда используют вместо нее числовой счетчик:

квалифицированное имя

Точечное имя, показывающее «путь» от глобальной области видимости модуля к классу, функции или методу, определенному в этом модуле, как определено в **PEP 3155**. Для функций и классов верхнего уровня квалифицированное имя совпадает с именем объекта:

количество ссылок

Количество ссылок на объект. Когда количество ссылок на объект падает до нуля, он деаллоцируется. Подсчет ссылок обычно не виден в коде Python, но он является ключевым элементом реализации CPython. Модуль **sys** определяет функцию **getrefcount()**, которую программисты могут вызвать, чтобы вернуть количество ссылок для конкретного объекта.

обычный пакет

Традиционный package, например, каталог, содержащий **__init__.py** файл.

__слоты__

Объявление внутри класса, которое экономит память за счет предварительного выделения места для атрибутов экземпляра и исключения словарей экземпляров. Несмотря на свою популярность, эта техника довольно сложна для правильного применения, и ее лучше использовать в редких случаях, когда в приложении, критичном к памяти, имеется большое количество экземпляров.

последовательность

Последовательность iterable, которая поддерживает эффективный доступ к элементам с использованием целочисленных индексов через специальный метод **__getitem__()** и определяет метод **__len__()**, возвращающий длину последовательности. Некоторые встроенные типы последовательностей - **list**, **str**, **tuple** и **bytes**. Обратите внимание, что **dict** также поддерживает **__getitem__()** и **__len__()**, но считается отображением, а не последовательностью, поскольку для поиска используются произвольные ключи immutable, а не целые числа.

Абстрактный базовый класс **collections.abc.Sequence** определяет гораздо более богатый интерфейс, который выходит за рамки только **__getitem__()** и **__len__()**,

добавляя **count()**, **index()**, **__contains__()** и **__reversed__()**. Типы, реализующие этот расширенный интерфейс, могут быть зарегистрированы в явном виде с помощью **register()**.

понимание набора

Компактный способ обработки всех или части элементов итерабельной таблицы и возврата набора с результатами. **results = {c for c in 'abracadabra' if c not in 'abc'}** генерирует набор строк **{'r', 'd'}**. См. Дисплеи для списков, наборов и словарей.

разовая отправка

Форма диспетчеризации generic function, при которой реализация выбирается на основе типа единственного аргумента.

нарезать

Объект, обычно содержащий часть sequence. Слайс создается с использованием нотации subscript, [] с двоеточиями между числами, когда их несколько, например, **variable_name[1:3:5]**. Скобочная (субскриптовая) нотация использует объекты **slice** внутри.

специальный метод

Метод, который неявно вызывается Python для выполнения определенной операции над типом, например, сложения. Такие методы имеют имена, начинающиеся и заканчивающиеся двойным подчеркиванием. Специальные методы документированы в Специальные имена методов.

заявление

Утверждение является частью набора («блока» кода). Утверждение - это либо expression, либо одна из нескольких конструкций с ключевым словом, например **if**, **while** или **for**.

сильная ссылка

В C API Python сильная ссылка - это ссылка на объект, которая увеличивает счетчик ссылок объекта при его создании и уменьшает счетчик ссылок объекта при его удалении.

Функция **Py_NewRef()** может быть использована для создания сильной ссылки на объект. Обычно функция **Py_DECREF()** должна быть вызвана на сильной ссылке до выхода из области видимости сильной ссылки, чтобы избежать утечки одной ссылки.

кодирование текста

Строка в Python - это последовательность кодовых точек Unicode (в диапазоне **U+0000–U+10FFFF**). Чтобы сохранить или передать строку, ее необходимо сериализовать в виде последовательности байтов.

Сериализация строки в последовательность байтов известна как «кодирование», а воссоздание строки из последовательности байтов - как «декодирование».

Существует множество различных сериализаций текста codecs, которые в совокупности называются «текстовыми кодировками».

текстовый файл

file object, способный читать и записывать объекты **str**. Часто текстовый файл фактически обращается к байт-ориентированному потоку данных и автоматически обрабатывает text encoding. Примерами текстовых файлов являются файлы, открытые в текстовом режиме

('r' или 'w'), **sys.stdin**, **sys.stdout** и экземпляры **io.StringIO**.

строка с тройными кавычками

Строка, связанная тремя экземплярами либо кавычек («), либо апострофа („). Хотя они не предоставляют никаких функциональных возможностей, недоступных для строк с одинарными кавычками, они полезны по ряду причин. Они позволяют включать в строку одинарные и двойные кавычки без раскрытия, а также могут охватывать несколько строк без использования символа продолжения, что делает их особенно полезными при написании документальных строк.

тип

Тип объекта Python определяет, к какому типу он относится; каждый объект имеет тип. Тип объекта доступен как атрибут **__class__** или может быть получен с помощью **type(obj)**.

псевдоним типа

Синоним типа, созданный путем присвоения типа идентификатору.

Псевдонимы типов полезны для упрощения type hints.

подсказка типа

annotation, который определяет ожидаемый тип для переменной, атрибута класса, параметра или возвращаемого значения функции.

Подсказки типов необязательны и не применяются в Python, но они полезны для инструментов статического анализа типов и помогают IDE в завершении кода и рефакторинге.

Подсказки типов глобальных переменных, атрибутов класса и функций, но не локальных переменных, могут быть доступны с помощью **typing.get_type_hints()**.

универсальные новые строки

Способ интерпретации текстовых потоков, при котором окончанием строки считаются все следующие символы: соглашение Unix о конце строки **'\n'**, соглашение Windows **'\r\n'** и старое соглашение Macintosh **'\r'**.

виртуальная среда

Совместно изолированная среда выполнения, которая позволяет пользователям и приложениям Python устанавливать и обновлять пакеты дистрибутива Python, не вмешиваясь в поведение других приложений Python, работающих на той же системе.

См. также **venv**.

виртуальная машина

Компьютер, полностью определяемый программно. Виртуальная машина Python выполняет bytecode, испускаемый компилятором байткода.

Дзен питона

Список принципов проектирования Python и философских концепций, полезных для понимания и использования языка. Листинг можно найти, набрав «**import this**» в интерактивной подсказке.

Список литературы

Основная литература:

9. Python для детей. Самоучитель по программированию / Джейсон Бриггс ; пер. с англ. Станислава Ломакина ; [науч. ред. Д. Абрамова]. — М. : Манн, Иванов и Фербер, 2017. — 320 с
10. Бессмертный, И. А. Системы искусственного интеллекта : учеб. пособие для СПО / И. А. Бессмертный. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2018. — 130 с.
11. Гниденко, И. Г. Технология разработки программного обеспечения : учеб. пособие для СПО / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — М. : Издательство Юрайт, 2017. — 235 с.
12. Гордеев, С. И. Организация баз данных в 2 ч. Часть 2 : учебник для вузов / С. И. Гордеев, В. Н. Волошина. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2019. — 501 с.
13. Жмудь, В. А. Моделирование замкнутых систем автоматического управления : учеб. пособие для академического бакалавриата / В. А. Жмудь. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2019. — 128 с.
14. Жорняк А.Г., Морозова Т.А., Наумченко В.П. Специфика применения языка программирования python при работе с базами данных // Научно-технический вестник Поволжья. 2022. № 5. С. 90-92
15. Завьялова О.А., Маркелов В.К. Возможности онлайн-сред программирования при обучении языку python в школе // Информатика в школе. 2022. № 3 (176). С. 75-82.
16. Зыков, С. В. Программирование. Объектно-ориентированный подход : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2019. — 155 с.
17. Иванов, В. М. Интеллектуальные системы : учеб. пособие для вузов / В. М. Иванов ; под науч. ред. А. Н. Сесекина. — М. : Издательство Юрайт, 2017. — 91 с.
18. Иванов, В. М. Интеллектуальные системы : учеб. пособие для СПО / В. М. Иванов ; под науч. ред. А. Н. Сесекина. — М. : Издательство Юрайт, 2019. — 93 с.
19. Косенко А.А. Противоречия объектно-ориенированного и процедурного стиля программирования в java // Научный Лидер. 2022. № 28 (73). С. 7-10.
20. Кубенский, А. А. Функциональное программирование : учебник и практикум для академического бакалавриата / А. А. Кубенский. — М. : Издательство Юрайт, 2019. — 348 с.
21. Кудрина, Е. В. Основы алгоритмизации и программирования на языке c# : учеб. пособие для СПО / Е. В. Кудрина, М. В. Огнева. — М. : Издательство Юрайт, 2019. — 322 с.
22. Кудрина, Е. В. Основы алгоритмизации и программирования на языке c# : учеб. пособие для бакалавриата и специалитета / Е. В. Кудрина, М. В. Огнева. — М. : Издательство Юрайт, 2019. — 322 с.
23. Кудрявцев, К. Я. Методы оптимизации : учеб. пособие для вузов / К. Я. Кудрявцев, А. М. Прудников. — 2-е изд. — М. : Издательство Юрайт, 2019. — 140 с.
24. Лаврищева, Е. М. Программная инженерия и технологии программирования сложных систем : учебник для вузов / Е. М. Лаврищева. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2019. — 432 с.

25. Лебедев, В. М. Программирование на vba в ms excel : учеб. пособие для академического бакалавриата / В. М. Лебедев. — М. : Издательство Юрайт, 2019. — 272 с.
26. Малявко, А. А. Формальные языки и компиляторы : учеб. пособие для вузов / А. А. Малявко. — М. : Издательство Юрайт, 2018. — 429 с
27. Мамонова, Т. Е. Информационные технологии. Лабораторный практикум : учеб. пособие для СПО / Т. Е. Мамонова. — М. : Издательство Юрайт, 2019. — 178 с.
28. Маркин, А. В. Программирование на sql в 2 ч. Часть 2 : учебник и практикум для бакалавриата и магистратуры / А. В. Маркин. — М. : Издательство Юрайт, 2019. — 292 с.
29. Меленова М.М. Особенности языка программирования python, которые необходимо учитывать при подготовке к олимпиадам по информатике // Молодой ученый. 2022. № 22 (417). С. 496-498.
30. Нагаева, И. А. Программирование: delphi : учеб. пособие для академического бакалавриата / И. А. Нагаева, И. А. Кузнецов ; под ред. И. А. Нагаевой. — М. : Издательство Юрайт, 2017. — 302 с.
31. Носарев П.Ю., Бужинская Н.В. Возможности языка программирования Python для решения задачи резюмирования текста // Тенденции развития науки и образования. 2022. № 86-2. С. 31-35.
32. Плескунов, М. А. Операционное исчисление : учеб. пособие для вузов / М. А. Плескунов ; под науч. ред. А. И. Короткого. — М. : Издательство Юрайт, 2019. — 141 с.
33. Программирование на языке высокого уровня Python : учеб. пособие для прикладного бакалавриата / Д. Ю. Федоров. — 2-е изд., перераб. и доп. — М. : Издательство Юрайт, 2019. — 161 с. — (Серия : Бакалавр. Прикладной курс)
34. Простой Python. Современный стиль программирования. — СПб.: Питер, 2016. — 480 с.: ил. — (Серия «Бестселлеры O'Reilly»)
35. Советов, Б. Я. Базы данных : учебник для прикладного бакалавриата / Б. Я. Советов, В. В. Цехановский, В. Д. Чертовской. — 3-е изд., перераб. и доп. — М. : Издательство Юрайт, 2019. — 420 с.
36. Стасышин, В. М. Базы данных: технологии доступа : учеб. пособие для СПО / В. М. Стасышин, Т. Л. Стасышина. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2018. — 164 с.
37. Сысолетин, Е. Г. Разработка интернет-приложений : учеб. пособие для СПО / Е. Г. Сысолетин, С. Д. Ростунцев. — М. : Издательство Юрайт, 2019. — 90 с.
38. Сысолетин, Е. Г. Разработка интернет-приложений : учеб. пособие для вузов / Е. Г. Сысолетин, С. Д. Ростунцев ; под науч. ред. Л. Г. Доросинского. — М. : Издательство Юрайт, 2019. — 90 с
39. Трофимов, В. В. Основы алгоритмизации и программирования : учебник для СПО / В. В. Трофимов, Т. А. Павловская ; под ред. В. В. Трофимова. — М. : Издательство Юрайт, 2019. — 137 с.
40. Тухфатуллин, Б. А. Численные методы расчета строительных конструкций. Метод конечных элементов : учеб. пособие для академического бакалавриата / Б. А. Тухфатуллин. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2019. — 157 с.
41. Федоров, Д. Ю. Программирование на языке высокого уровня python : учеб. пособие для прикладного бакалавриата / Д. Ю. Федоров. — 2-е изд., перераб. и доп. — М. : Издательство Юрайт, 2019. — 161 с.
42. Федоров, Д. Ю. Программирование на языке высокого уровня python : учеб. пособие для СПО / Д. Ю. Федоров. — М. : Издательство Юрайт, 2019. — 126 с.
43. Халевин Т.А. Аннотации типов в языке программирования Python // E-Scio. 2022. № 6 (69). С. 497-502.
44. Халевин Т.А. Реализация калькулятора обратной польской нотации на языке программирования python // Modern Science. 2022. № 6-2. С. 252-257.
45. Черткова, Е. А. Статистика. Автоматизация обработки информации : учеб. пособие для вузов / Е. А. Черткова ; под общ. ред. Е. А. Чертковой. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2017. — 195 с.

46. Шумилина М.А., Коробко А.В. Разработка чат-бота на языке программирования python в мессенджере "telegram" // Научные известия. 2022. № 28. С. 47-55.

Интернет-ресурсы:

1. Руководство по языку программирования python [Электронный ресурс] / https://metanit.com/. – Режим доступа: https://metanit.com/python/tutorial/, свободный.

2. Самоучитель Python [Электронный ресурс] / https://pythonworld.ru/. Режим доступа: https://pythonworld.ru/samouchitel-python, свободный.

3. Самоучитель Python [Электронный ресурс] / http://pythoshka.ru/. Режим доступа: http://pythoshka.ru/p1138.html/samouchitel-python/p1138.html, свободный.