

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

Основы программирования на языке Python

Урок 1

Введение в Python

Содержание

Введение в Python и программирование	6
Основы программирования	
Как работает компьютерная программа?	6
Естественные языки	
и языки программирования	8
Из чего состоит язык?.....	9
Компиляция и интерпретация	11
Что на самом деле делает интерпретатор?.....	13
Компиляция и интерпретация —	
преимущества и недостатки	15
Python — инструмент, а не рептилия.....	17
Что такое Python?	17
Кто создал Python?.....	18
Программный проект-хобби	19
Цели Python.....	19
Почему Python особенный?	20

Конкуренты Python?	22
Где мы можем увидеть Python в действии?	23
Почему не Python?	23
Python не один	24
Python aka CPython	26
Cython	27
Jython	28
PyPy и RPython	29
Начните свое путешествие с Python	31
Как установить и использовать Python.....	31
Как скачать и установить Python	32
Начинаем работать с Python	33
Как написать и запустить вашу первую программу	35
Как испортить и исправить свой код	38
Типы данных, переменные, основные операции ввода-вывода, основные операторы	42
Пишем первую программу Hello, World!	42
Функция print()	43
Литералы Python	59
Литералы — данные в себе.....	59
Целые числа (Integers).....	60
Целые числа: восьмеричные и шестнадцатеричные числа	62
Числа с плавающей точкой (Floating-point numbers)	63
Кодирование чисел с плавающей точкой	65
Строки	66

Кодирование строк.....	68
Булевые значения (логические типы данных).....	68
Ключевые выводы	70
Операторы — инструменты управления данными ...	72
Python как калькулятор	72
Основные операторы.....	72
Арифметические операторы:	
возведение в степень.....	73
Арифметические операторы: умножение.....	74
Арифметические операторы: деление	74
Арифметические операторы:	
целочисленное деление	75
Операторы: остаток	
(деление по модулю, с остатком)	77
Операторы: как не делить.....	78
Операторы: суммирование	78
Оператор вычитания,	
унарные и бинарные операторы	79
Операторы и их приоритеты	80
Операторы и связывание.....	81
Операторы и связывание: возведение в степень	81
Список приоритетов	82
Операторы и скобки	83
Ключевые выводы	83
Переменные — поля в форме данных.....	85
Что такое переменные?	85
Правильные и неправильные имена переменных....	87
Ключевые слова.....	88

Создание переменных	89
Использование переменных	90
Присвоение нового значения уже существующей переменной	91
Решение простых математических задач.....	93
Сокращенные формы записи	94
Ключевые выводы	95
Комментарий к комментариям.....	97
Комментарии в коде: зачем, как и когда	97
Ключевые выводы	99
Как общаться с компьютером.....	101
Функция ввода input()	101
Функция input() с аргументом.....	102
Результат функции input()	103
Функция input() — запрещенные операции	103
Преобразование типов.....	104
Больше об input() и преобразовании типов.....	105
Строковые операторы — введение.....	107
Конкатенация (concatenation).....	107
Повторение строки (replication).....	108
Преобразование типов: str()	109
Снова возвращаемся к прямоугольному треугольнику.....	109
Ключевые выводы	110

Введение в Python и программирование

- Основы компьютерного программирования
- Настройка среды программирования
- Компиляция и интерпретация
- Знакомство с Python

Основы программирования

Как работает компьютерная программа?

Задача этого курса — рассказать вам о том, что такое язык программирования Python и для чего он используется. Давайте начнем с самых основ.

Мы можем пользоваться компьютером благодаря программам. Без программ компьютер, даже самый мощный, — просто какой-то объект. Точно так же и пианино без пианиста — не более, чем деревянная коробка.

Компьютеры могут выполнять очень сложные задачи, но эта способность не является естественной. Природа компьютера совсем иная.

Он может выполнять только чрезвычайно простые операции, например, компьютер не может самостоятельно оценить значение сложной математической функции, хотя это не выходит за рамки возможного в ближайшем будущем.

Современные компьютеры могут только оценивать результаты простых арифметических операций, таких как

сложение или деление, но они это делают очень быстро и могут повторять эти действия любое количество раз.

Представьте, что вы хотите узнать вашу среднюю скорость, с которой вы шли в течение долгого путешествия. Вы знаете расстояние и время, но вам нужна скорость.



Рисунок 1

Естественно, компьютер сможет вычислить это, но компьютер не понимает, что такое расстояние, скорость или время. Следовательно, нам нужно проинструктировать компьютер, чтобы он:

- принял число, представляющее расстояние;
- принял число, представляющее время в пути;
- разделил первое значение на второе и сохранил результат в памяти;
- вывел результат (представляющий среднюю скорость) в формате, который человек может прочитать.

Эти четыре простые действия образуют программу. Конечно, эти примеры не формализованы и они очень далеки от того, что компьютер может понять, но их достаточно, чтобы перевести это все на понятный компьютеру язык.

Язык здесь ключевое слово.

Естественные языки и языки программирования

Язык — это средство (и инструмент) для выражения и записи мыслей. В мире существует огромное количество языков. Некоторые из них не требуют ни речи, ни письма, например язык тела. Вы можете выразить свои самые глубокие чувства очень точно, не говоря ни слова.

Другим языком, которым вы пользуетесь каждый день, является ваш родной язык, при помощи которого вы декларируете свои желания и думаете. У компьютеров тоже есть свой язык под названием машинный язык, но он очень примитивный.

Компьютер, даже самый технически сложный, лишен следов интеллекта. Можно сказать, что он похож на хорошо обученную собаку — она реагирует только на заранее определенный набор известных ей команд.

Команды, которые он распознает, очень просты. Можно представить, что компьютер реагирует на такие команды, как «взять это число, разделить на другое и сохранить результат».

Полный набор известных команд называется списком инструкций (*instruction list*) и иногда сокращается до IL. Компьютеры различаются в зависимости от размера их IL, и инструкции могут быть совершенно разными в разных моделях.

Примечание: машиинные языки разрабатываются людьми.

Ни один компьютер в настоящее время не способен создавать новый язык. Однако это может скоро измениться.

Люди тоже используют огромное количество очень разных языков, но эти языки создали сами себя. Более того, они все еще развиваются.

Новые слова создаются каждый день, а старые слова исчезают. Эти языки называются естественными.

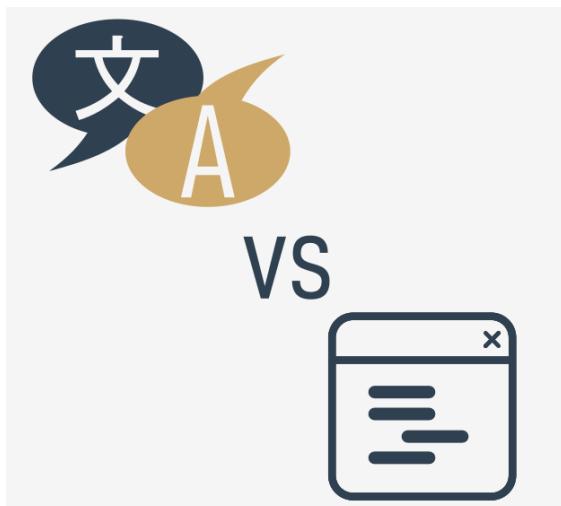


Рисунок 2

Из чего состоит язык?

Можно смело сказать, что каждый язык (машинный или естественный, неважно) состоит из следующих элементов:

- **Алфавит** — набор символов, используемых для построения слов определенного языка (например, латинский алфавит для английского, кириллический алфавит для русского, кандзи для японского и т.д.)
- **Лексика** — (словарь) набор слов, которые язык предлагает своим пользователям (например, слово «computer»)

происходит из словаря английского языка, в то время как «*смотреть*» нет; слово «*chat*» присутствует как в английском, так и во французском словарях, но имеют разные значения)

- **Синтаксис** — набор правил (формальных или неформальных, написанных или воспринимаемых интуитивно), которые помогают определить, является ли тот или иной набор словосочетаний правильным предложением (например, «*I am a python*» — синтаксически правильная фраза, а «*I a python am*» — нет)
- **Семантика** — набор правил, определяющих, имеет ли смысл определенная фраза (например, «я съел пончик» имеет смысл, а «пончик съел меня» — нет)

На самом деле, **список инструкций** — это азбука машинного языка. Это самый простой и самый базовый набор символов, который мы можем использовать, чтобы отдавать команды компьютеру. Это родной язык компьютера.

К сожалению, этот язык очень далек от родного языка людей. Нам всем (и компьютерам, и людям) нужно что-то еще, общий язык для компьютеров и людей или мост между двумя разными мирами.

Нам нужен язык, на котором люди могут писать свои программы, и язык, который компьютеры могут использовать для выполнения программ, язык, который намного сложнее, чем машинный язык, и все же намного проще, чем естественный язык.

Такие языки часто называют **высокоуровневыми языками программирования**. Они чем-то похожи на естественные

в том, что они используют символы, слова и условные обозначения, которые человек способен понять. Эти языки позволяют людям передавать компьютерам команды, которые намного сложнее, чем список инструкций.

Программа, написанная на высокоуровневом языке программирования, называется «исходный код» (в отличие от машинного кода, выполняемого компьютерами). А файл, содержащий исходный код, называется «исходный файл».

Компиляция и интерпретация

Компьютерное программирование — это процесс составления элементов выбранного языка программирования в том порядке, который будет вызывать желаемый эффект. Эффект может отличаться в каждом конкретном случае. Все зависит от воображения, знаний и опыта программиста.

Конечно, такое сочинение должно быть правильным по многим пунктам:

- **Алфавит** — программа должна быть написана узнаваемым набором букв, например, латиницей, кирилицей и т. д.
- **Лексика** — у каждого языка программирования есть свой словарь, и вам нужно освоить его; к счастью, он намного проще и меньше, чем словарь любого естественного языка;
- **Синтаксис** — у каждого языка есть свои правила, и они должны соблюдаться;
- **Семантика** — программа должна иметь смысл.

К сожалению, программист также может ошибиться в каждом из четырех указанных выше пунктов. Ошибка в любом из них может сделать программу совершенно бесполезной.

Представим, что вы успешно написали программу. Как мы убедим компьютер выполнить ее? Вам нужно перевести вашу программу на машинный язык. К счастью, сам компьютер может выполнить этот перевод, а значит, это будет быстро и эффективно.

Есть два разных способа преобразование программы из высокоуровневого языка программирования в машинный язык:

- **КОМПИЛЯЦИЯ** — исходная программа переводится один раз (однако, это действие необходимо повторять каждый раз, когда вы изменяете исходный код), вы получаете файл (например, файл .exe, если код предназначен для запуска в MS Windows), содержащий машинный код. Теперь вы можете распространять файл по всему миру. Программа, выполняющая этот перевод, называется компилятором или переводчиком;
- **ИНТЕРПРЕТАЦИЯ** — вы (или любой человек, который использует этот код) можете переводить исходную программу каждый раз, когда она должна быть запущена; программа, выполняющая такое преобразование, называется интерпретатором, так как она интерпретирует код каждый раз, когда он должен быть выполнен; это также означает, что вы не можете просто взять и распространить исходный код как есть, потому что конечному пользователю также будет нужен интерпретатор для его выполнения.

По некоторым основополагающим причинам тот или иной высокоуровневый язык программирования должен попасть в одну из этих двух категорий.

Существует очень мало языков, которые можно и компилировать, и интерпретировать. Создатели языков программирования обычно сразу решают, будет ли новый язык компилироваться или интерпретироваться.

Что на самом деле делает интерпретатор?

Давайте еще раз представим, что вы написали программу. Теперь она существует как компьютерный файл: компьютерная программа на самом деле представляет собой фрагмент текста, поэтому исходный код обычно хранится в текстовых файлах.

Примечание: это должен быть чистый текст, без каких-либо украшений, таких как разные шрифты, цвета, встроенные изображения, аудио, видео и т.д. Теперь вы должны вызвать интерпретатор, чтобы он прочитал ваши исходный файл.

Интерпретатор читает исходный код так, как это принято в западной культуре: сверху вниз и слева направо. Есть некоторые исключения — они будут рассмотрены далее в этом курсе.

Прежде всего, интерпретатор проверяет правильность всех последующих строк (через призму тех четырех пунктов, которые были рассмотрены выше).

Если компилятор находит ошибку, он немедленно завершает свою работу. Единственным результатом в этом

случае является **сообщение об ошибке**. Интерпретатор сообщает вам, где находится ошибка и что ее вызвало. Однако эти сообщения могут и запутывать, поскольку интерпретатор не в состоянии проследить ваши точные намерения и может сообщить об ошибке на некотором расстоянии от места, которое, собственно, и содержит эту ошибку.

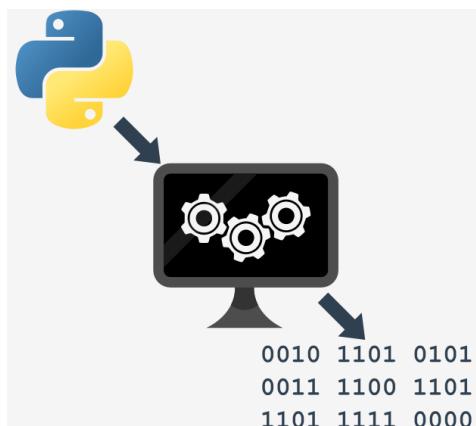


Рисунок 3

Например, если вы попытаетесь использовать объект с неизвестным именем, это приведет к ошибке, но интерпретатор обнаружит ошибку там, где он попытается использовать объект, а не там, где было введено имя нового объекта.

Другими словами, фактическая причина обычно находится немного выше в коде, например, в том месте, где вы должны были сообщить интерпретатору, что вы собираетесь использовать объект с этим именем.

Если строка выглядит хорошо, интерпретатор пытается выполнить ее.

Примечание: каждая строка обычно выполняется отдельно, поэтому трио «чтение-проверка-выполнение» может повторяться много раз — больше, чем фактическое количество строк в исходном файле, так как некоторые части кода могут быть выполнены более одного раза).

Также возможно, что значительная часть кода может быть успешно выполнена до того, как интерпретатор обнаружит ошибку. Это нормальное поведение в этой модели выполнения.

У вас может возникнуть логичный вопрос — а что же лучше? Модель компиляции или модель интерпретации? Тут нет очевидного ответа. Если бы он был, одна из этих моделей давно бы перестала существовать. У каждой из них есть свои преимущества и недостатки.

Компиляция и интерпретация — преимущества и недостатки

	КОМПИЛЯЦИЯ	ИНТЕРПРЕТАЦИЯ
ПРЕИМУЩЕСТВА	<ul style="list-style-type: none"> ■ Выполнение переведенного кода обычно происходит быстрее; ■ Только у программиста должен быть компилятор — конечный пользователь может использовать код без него; ■ Переведенный код хранится за счет машинного языка — поскольку его очень трудно понять, ваши собственные находки и приемы программирования, вероятно, останутся вашим секретом. 	<ul style="list-style-type: none"> ■ Код можно запустить сразу же, как только закончите его — дополнительных этапов перевода нет; ■ Код хранится за счет языка программирования, а не машинного — это означает, что его можно запускать на компьютерах, которые используют разные машинные языки; не нужно компилировать код для каждой отдельной архитектуры.

	КОМПИЛЯЦИЯ	ИНТЕРПРЕТАЦИЯ
НЕДОСТАТКИ	<ul style="list-style-type: none"> Сама компиляция может быть очень трудоемким процессом — вы не сможете запустить код сразу же после любого изменения; Количество используемых компиляторов должно соответствовать количеству аппаратных платформ, на которых вы хотите, чтобы ваш код работал. 	<ul style="list-style-type: none"> Не думайте, что интерпретация увеличит скорость вашего кода — ваш код поделится мощностью компьютера с интерпретатором, так что на скорость рассчитывать не приходится; И у вас, и у конечного пользователя должен быть интерпретатор для запуска вашего кода.

Что все это значит?

- Python является интерпретируемым языком программирования. Это означает, что он наследует все описанные выше преимущества и недостатки. Конечно, он добавляет и свои уникальные особенности к обоим наборам.
- Если вы хотите программировать на Python, вам понадобится интерпретатор Python. Вы не сможете запустить код без него. К счастью, Python бесплатный. Это одно из его самых важных преимуществ.

Python — инструмент, а не рептилия

По историческим причинам интерпретируемые языки программирования часто называют «языками сценариев» (сценарными языками, скриптовыми языками), а исходные программы, закодированные с их использованием, называются «сценарии» (скрипты).

Что такое Python?

Python — это широко используемый, интерпретируемый, объектно-ориентированный, высокоуровневый и многоцелевой язык программирования с динамической семантикой.

Возможно, первое, что вы вспоминаете, когда слышите слово «питон», это большая змея, однако название языка программирования Python происходит от старого комедийного сериала BBC под названием «Летающий цирк Монти Пайтона».

На пике своего успеха команда Monty Python разыгрывала свои скетчи для живой аудитории по всему миру, в том числе на Голливуд-боул, знаменитом концертном зале в виде амфитеатра.

Поскольку Монти Пайтон считается одним из двух основных питательных веществ для программиста (второе — пицца), создатель Python назвал язык в честь этого телешоу.

Кто создал Python?

Одной из удивительных особенностей Python является тот факт, что на самом деле это работа одного человека. Обычно новые языки программирования разрабатываются и публикуются крупными компаниями, в которых работает много профессионалов, и авторские права часто не позволяют назвать кого-либо из участников проекта. Python исключение.

Существует не так много языков, авторов которых мы знаем поименно. Python был создан Гвидо Ван Россумом, который родился в 1956 году в Харлеме, Нидерланды. Конечно, Гвидо Ван Россум не разрабатывал и не развивал абсолютно все составляющие Python.



Рисунок 4

Скорость, с которой Python распространился по всему миру, является результатом непрерывной работы тысяч (часто анонимных) программистов, тестировщиков, пользователей (многие из которых не являются ИТ-специалистами) и энтузиастов, но следует сказать, что самая первая идея (семя, из которого вырос Python) пришла только в одну голову — Гвидо.

Программный проект-хобби

Python был создан в немного странных условиях. По словам Гвидо Ван Россума:

«В декабре 1989-го года я искал проект, который бы стал хобби на рождественские каникулы. Офис со всем оборудованием был закрыт, но дома у меня был компьютер, а других планов, чем заняться, не было. Я решил написать интерпретатор для нового скриптового языка, о котором думал в последнее время: о потомке ABC, который был бы привлекателен для Unix/C-хакеров. Я выбрал Python в качестве рабочего названия пребывая в слегка непочтительном настроении (и будучи большим фанатом летающего цирка Монти Пайтона)».

Цели Python

В 1999 году Гвидо Ван Россум определил цели Python:

- простой и интуитивно понятный язык, такой же мощный, как и основные конкуренты;
- открытый исходный код, чтобы каждый мог внести свой вклад в его развитие;
- код должен быть таким же понятным, как обычный английский язык;
- подходящий для повседневных задач, позволяет уложиться в короткие сроки разработки.

Спустя 20 лет стало ясно, что все поставленные цели были осуществлены. Некоторые источники говорят, что Python является самым популярным языком программирования в мире, в то время как другие утверждают, что он занимает третье или пятое место по популярности.



Рисунок 5

В любом случае, он по-прежнему занимает высокое место в первой десятке в рейтинге языков программирования PYPL (*Popularity of Programming Language*) и в индексе TIOBE (*TIOBE programming community index*).

Python не молодой язык. Он зрелый и заслуживает доверия. Это не язык-однодневка. Это яркая звезда на небосклоне программирования, и время, потраченное на изучение Python, является очень хорошей инвестицией.

Почему Python особенный?

Как получилось, что программисты, молодые и старые, опытные и начинающие, хотят его использовать? Как так получилось, что крупные компании признали Python и создали свои флагманские продукты, используя его?

На это есть много причин — мы уже перечислили некоторые из них, но давайте перечислим их снова в более структурированном порядке:

- его легко учить — время, необходимое для изучения Python, обычно меньше, чем для многих других языков

ков; это означает, что можно начать программировать быстрее;

- ему легко учить — учебная нагрузка меньше, чем требуется для других языков; это означает, что учитель может уделять больше внимания общим (независимым от языка) методам программирования, не теряя времени на экзотические приемы, странные исключения и непонятные правила;
- его легко использовать для написания нового программного обеспечения — с Python на код часто уходит гораздо меньше времени;
- его легко понять — также часто легче понять чужой код, если он написан на Python;
- его легко получить, установить и внедрить — Python бесплатный, открытый и мультиплатформенный; не все языки могут этим похвастаться.

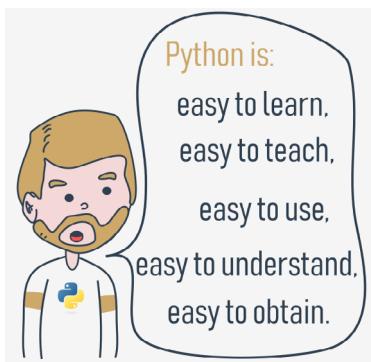


Рисунок 6

Конечно, у Python есть и свои недостатки:

- он не супер-скоростной — Python не обеспечивает исключительную производительность;

- в некоторых случаях он может быть устойчив к некоторым более простым методам тестирования — это значит, что отладка кода Python может быть более сложной, чем в других языках; к счастью, и делать ошибки в Python сложнее.

Следует также отметить, что Python не является единственным решением подобного рода, доступным на рынке информационных технологий.

У него много последователей, но многие предпочитают другие языки и даже не рассматривают Python для своих проектов.

Конкуренты Python?

У Python есть два прямых конкурента с сопоставимыми свойствами и проблемами. Это:

- **Perl** — язык сценариев, созданный Ларри Уоллом;
- **Рубин** — язык сценариев, созданный Юкихиро Мацумото.

Первый более традиционный, более консервативный, чем Python, и напоминает некоторые старые добрые языки, основанные на классическом языке программирования С.

А последний является более инновационным, с более свежими идеями, чем Python. Сам Python лежит где-то между этими двумя языками.

Интернет полон форумов с бесконечными дискуссиями о превосходстве одного из этих трех над другими, на случай, если вы хотите узнать больше о каждом из них.

Где мы можем увидеть Python в действии?

Мы видим его каждый день и чуть ли не на каждом шагу. Он широко используется для реализации сложных Интернет-сервисов, например, поисковых систем, облачных хранилищ и инструментов, социальных сетей и так далее. Всякий раз, когда вы используете какой-либо из этих сервисов, вы на самом деле очень близки к Python, хотя вы этого и не знаете.

Многие инструменты разработки реализованы на Python. Все больше и больше ежедневно используемых приложений пишутся на Python. Многие ученые отказались от дорогих фирменных инструментов и перешли на Python. Многие тестировщики ИТ-проектов начали использовать Python для выполнения повторяющихся процедур тестирования. Список большой.



Рисунок 7

Почему не Python?

Несмотря на растущую популярность Python, есть еще ниши, где Python совсем отсутствует или встречается редко:

- низкоуровневое программирование (иногда его называют программированием «ближе к железу»): если вы хотите реализовать чрезвычайно эффективный драйвер или графический движок, вы не станете использовать Python;
- приложения для мобильных устройств: пока что Python не завоевал эту территорию, но, скорее всего, когда-нибудь завоюет.

Python не один

Существует два основных типа Python, которые называются Python 2 и Python 3.

Python 2 является более старой версией оригинального Python. С тех пор его развитие намеренно застопорилось, хотя это не значит, что его не обновляют. Напротив, обновления регулярно выходят, но они не вносят каких-либо существенных изменений в язык. Они скорее исправляют любые недавно обнаруженные ошибки и дыры в безопасности. Путь разработки Python 2 уже зашел в тупик, но сам Python 2 все еще очень даже жив.

Python 3 является более новой (точнее, текущей) версией языка. Он проходит свой собственный путь развития, создавая свои стандарты и привычки.

Первый более традиционный, более консервативный, чем Python, и напоминает некоторые старые добрые языки, основанные на классическом языке программирования C.

Эти две версии Python не совместимы друг с другом. Скрипты Python 2 не будут работать в среде Python 3 и наоборот, поэтому, если вы хотите, чтобы старый код

Python 2 выполнялся интерпретатором Python 3, единственное возможное решение — переписать его, конечно же, не с нуля. Так как большие части кода могут остаться нетронутыми, но вам нужно пересмотреть весь код, чтобы найти все возможные несовместимости. К сожалению, этот процесс не может быть полностью автоматизирован.

Перенос старого приложения Python 2 на новую платформу — это слишком сложно, занимает слишком много времени, слишком дорого и слишком рискованно. Вполне вероятно, что переписанный код будет содержать новые ошибки. Проще и разумнее оставить эти системы в покое и улучшить существующий интерпретатор, вместо того чтобы пытаться работать внутри уже работающего исходного кода.

Python 3 — это не просто лучшая версия Python 2, это совершенно другой язык, хотя он очень похож на своего предшественника. Если посмотреть на них издалека, они кажутся одинаковыми, но, если присмотреться, можно заметить множество различий.

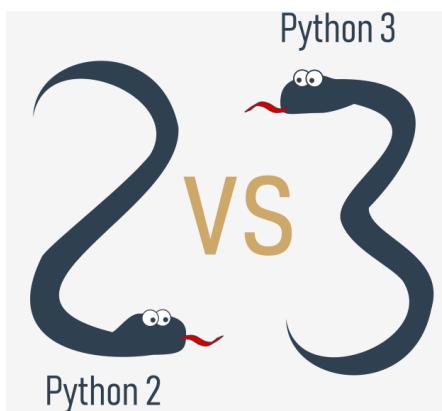


Рисунок 8

Если вы модифицируете старое решение, написанное на Python, то весьма вероятно, что оно было написано именно на Python 2. Вот почему Python 2 все еще используется. Сейчас существует слишком много приложений на Python 2, чтобы полностью от них отказаться.

Примечание: если вы хотите написать новый проект на Python, используйте Python 3. Именно эту версию Python мы будем использовать в этом курсе.

Важно помнить, что между последующими релизами Python 3 могут быть маленькие или большие различия (например, в Python 3.6 введены упорядоченные словарные ключи по умолчанию под реализацией CPython) — хорошая новость заключается в том, что все более новые версии Python 3 обратно совместимы с предыдущими версиями. Мы всегда будем стараться подчеркнуть эти различия в курсе, если это будет важно и полезно.

Все примеры кода, которые вы найдете в этом курсе, протестированы на Python 3.4, Python 3.6 и Python 3.7.

Python aka CPython

Кроме Python 2 и Python 3 существует более одной версии каждого из них.

Прежде всего, есть языки Python, которые поддерживают люди, принадлежащие к PSF ([Python Software Foundation](#)), сообщество, которое стремится развивать, улучшать, расширять и популяризировать Python и его среду. Президентом PSF является сам Гвидо Ван Россум, поэтому эти языки Python называются «каноническими».

Они также считаются референсными языками Python, так как любая другая реализация языка, должна соответствовать всем стандартам, установленным PSF.



Рисунок 9

Гвидо Ван Россум использовал язык программирования «С» для реализации самой первой версии своего языка, и это решение остается в силе. Все языки Python, поступающие из PSF, написаны на языке «С». Такой подход обусловлен многими причинами и имеет много последствий. Одно из них (вероятно, самое важное) заключается в том, что благодаря этому Python можно легко портировать и мигрировать на все платформы с возможностью компиляции и запуска программ на языке «С» (практически у всех платформ есть эта функция, что открывает множество возможностей расширения для Python).

Вот почему реализацию PSF часто называют **CPython**. Это самый влиятельный Python среди всех языков Python в мире.

Cython

Другой член семьи Python — Cython.

Cython — это одно из возможных решений для самой проблемной черты Python — недостаточной эффективности. Большие и сложные математические вычисления

можно легко закодировать в Python (намного проще, чем в «С» или любом другом традиционном языке), но выполнение результирующего кода может быть чрезвычайно трудоемким.

Как примирить эти два противоречия? Одно из решений — написать ваши математические идеи с использованием Python, и когда вы абсолютно уверены в том, что ваш код верен и дает правильные результаты, перевести его в «С». Конечно, «С» будет работать намного быстрее, чем чистый Python.

Именно это и делает Cython — автоматически переводит код Python (чистый и понятный, но не слишком быстрый) в код «С» (сложный и многословный, но гибкий).



Рисунок 10

Jython

Еще одна версия Python называется Jython.

«J» расшифровывается как «Java». Представьте себе Python, написанный на Java вместо С. Это полезно, например, если вы разрабатываете большие и сложные системы, полностью написанные на Java, и хотите добавить к ним некоторую гибкость Python. Традиционный CPython может быть трудно интегрировать в такую среду, поскольку С и Java живут в совершенно разных мирах и не разделяют многие идеи.

Jython может более эффективно взаимодействовать с существующей инфраструктурой Java. Вот почему некоторые проекты считают его полезным и нужным.

Примечание: текущая реализация Jython соответствует стандартам Python 2. Пока что не существует Jython, который соответствует Python 3.



Рисунок 11

PyPy и RPython

Посмотрите на логотип ниже. Это ребус. Вы можете его разгадать?



Рисунок 12

Это логотип PyPy — Python внутри Python. Другими словами, он представляет собой среду Python, написанную на Python-подобном языке и названную **RPython** (*Restricted Python, ограниченный Python*). На самом деле,

это разновидность Python. Исходный код PyPy не интерпретируется, вместо этого он переводится на язык программирования C, а затем выполняется отдельно.

Это полезно в том случае, если вам нужно протестировать какую-нибудь новую функцию, которая может быть (но не обязательно) внедрена в обычную реализацию Python, тогда ее проще проверить с помощью PyPy чем с CPython. Таким образом, PyPy скорее инструмент для людей, разрабатывающих Python, чем для остальных пользователей.

Конечно, это не умаляет заслуг PyPy и не делает его менее серьезным по сравнению с CPython.

К тому же, PyPy совместим с языком Python 3.

В мире есть еще много разных языков Python. Вы найдете их, если поищите, но в этом курсе используется CPython.

Начните свое путешествие с Python

Как установить и использовать Python

Есть несколько способов получить собственную копию Python 3 в зависимости от используемой операционной системы.

У пользователей Linux, скорее всего, уже установлен Python, поскольку инфраструктура Python интенсивно используется многими компонентами ОС Linux.

Например, некоторые дистрибуторы могут сочетать свои инструменты с системой, и многие из этих инструментов, такие как менеджеры пакетов, часто написаны на Python. Некоторые элементы графических сред, которые доступны в мире Linux, также могут использовать Python.

Если вы пользователь Linux, откройте терминал/консоль и введите:

```
python3
```

в командной строке нажмите **Enter** и подождите.

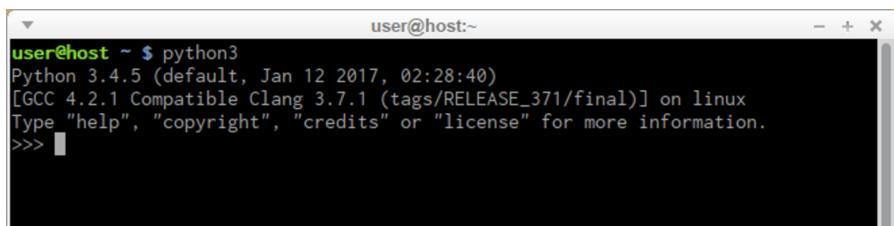
Если вы видите что-то вроде этого:

```
Python 3.4.5 (default, Jan 12 2017, 02:28:40)
[GCC 4.2.1 Compatible Clang 3.7.1 (tags/RELEASE_371/
final)] on linux Type "help", "copyright", "credits"
or "license" for more information. >>>
```

тогда вам не нужно больше ничего делать.

Если Python 3 отсутствует, обратитесь к документации Linux, чтобы узнать, как использовать менеджер пакетов для загрузки и установки нового пакета. Тот, который вам нужен, называется «python3» или его имя начинается с этого слова.

Те, кто не использует Linux, могут скачать копию по этой ссылке: <https://www.python.org/downloads/>.



A screenshot of a terminal window titled "user@host:~". The window shows the command "python3" being run and its output. The output includes the Python version (3.4.5), the date it was built (Jan 12 2017), the time (02:28:40), the compiler used (GCC 4.2.1 Compatible Clang 3.7.1), the platform (linux), and instructions for getting more information ("help", "copyright", "credits" or "license"). The prompt ">>> " is visible at the bottom of the terminal window.

Рисунок 13

Как скачать и установить Python

Поскольку браузер сообщает сайту, на который вы зашли, используемую ОС, единственное, что вам нужно сделать — щелкнуть по нужной версии Python.

В нашем случае это Python 3. Сайт всегда предлагает вам последнюю версию.

Если вы пользователь Windows, запустите загруженный файл .exe и выполните все шаги.

Пока оставьте настройки по умолчанию, которые предлагает установщик, за одним исключением — посмотрите на пункт **Add Python 3.x to PATH** и отметьте его.

Это многое упростит.

Если вы используете Macos, версия Python 2, возможно, уже была предварительно установлена на вашем компьютере, но, поскольку мы будем работать с Python 3,

вам все равно потребуется загрузить и установить соответствующую версию файла *.pkg* с сайта Python.



Рисунок 14

Начинаем работать с Python

Теперь, когда у вас установлен Python 3, пришло время проверить как он работает и впервые использовать его.

Это будет очень простая процедура, но ее должно быть достаточно, чтобы убедить вас, что среда Python является полноценной и функциональной.

Существует много способов использования Python, особенно если вы собираетесь стать Python-разработчиком.

Для начала работы вам понадобятся следующие инструменты:

- **редактор**, который поможет вам в написании кода (у него должны быть некоторые специальные функции,

недоступные в простых инструментах); этот специальный редактор даст вам больше, чем стандартная программа вашей ОС;

- консоль, в которой вы можете запустить свой свеженаписанный код и принудительно остановить его, когда он выйдет из-под контроля;
- инструмент под названием **отладчик**, который запускает пошаговое выполнение кода и позволяет вам проверять его на каждом этапе выполнения.

Помимо множества полезных компонентов, стандартная установка Python 3 содержит очень простое, но чрезвычайно полезное приложение под названием IDLE.

IDLE — это аббревиатура от *Integrated Development and Learning Environment* (интегрированная среда разработки и обучения).

Откройте меню вашей ОС, найдите IDLE где-нибудь под Python 3.x и запустите ее. Вот что у вас должно появиться:

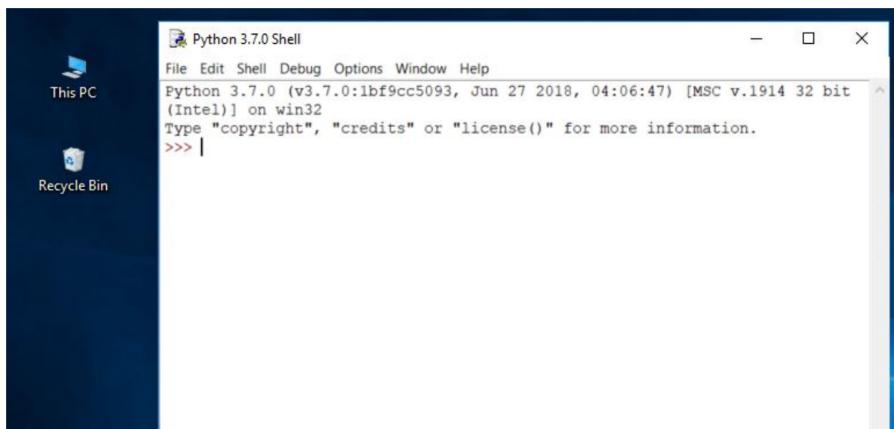


Рисунок 15

Как написать и запустить вашу первую программу

Теперь пришло время написать и запустить вашу первую программу на Python 3. Это будет очень просто, пока что.

Первый этап — создать новый исходный файл и заполнить его кодом. Нажмите на **File** в меню **IDLE** и выберите **New file**.

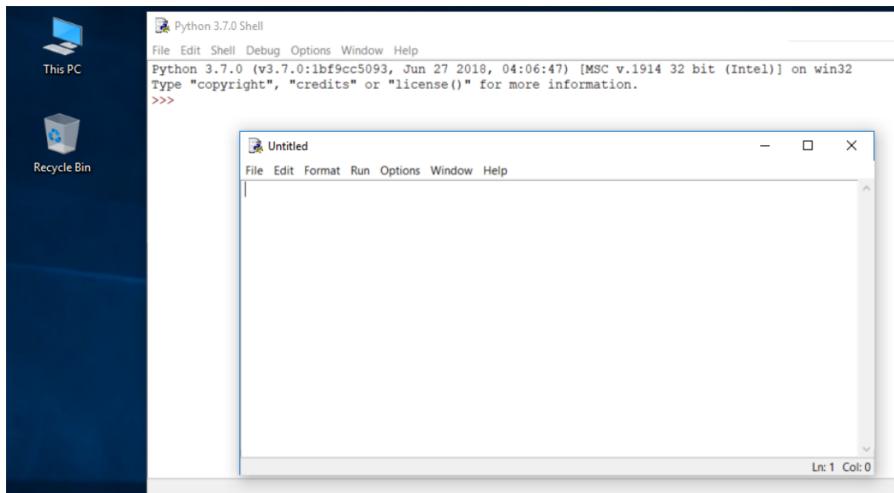


Рисунок 16

Как видите, IDLE открывает для вас новое окно. Вы можете использовать его, чтобы писать и изменять свой код.

Это окно редактора. Его единственная цель — быть рабочим местом, в котором обрабатывается ваш исходный код. Не путайте окно редактора с окном оболочки. Они выполняют разные функции.

Окно редактора в настоящее время без названия, но начинать работу рекомендуется с названия исходного файла.

Нажмите на **File** (в новом окне), затем нажмите **Save as...**, выберите папку для нового файла (рабочий стол — неплохое место для ваших первых попыток в программировании) и выберите имя для нового файла.

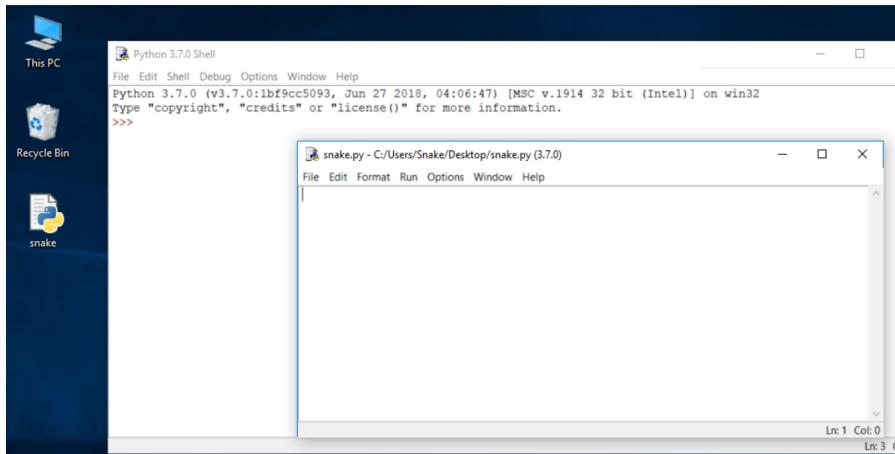


Рисунок 17

Примечание: не устанавливайте расширение для имени файла, который вы собираетесь использовать. Python требует расширения .py для своих файлов, поэтому вы должны полагаться на значения по умолчанию в диалоговом окне. Стандартное расширение .py позволяет ОС правильно открывать эти файлы.

Теперь добавьте только одну строку в окно редактора, которое вы недавно открыли и назвали.

Строка выглядит так:

```
print("Hisssssss...")
```

Можно использовать буфер обмена, чтобы скопировать текст в файл.

Пока что мы не будем объяснять смысл программы. Вы найдете подробное обсуждение в следующем уроке.

Обратите внимание на кавычки. Это самые простые кавычки (нейтральные, прямые, двойные и т.д.), которые обычно используются в исходных файлах. Не пытайтесь использовать типографские кавычки (закругленные, фигурные, книжные и т.д.), которые используются продвинутыми текстовыми процессорами, поскольку Python их не воспринимает.

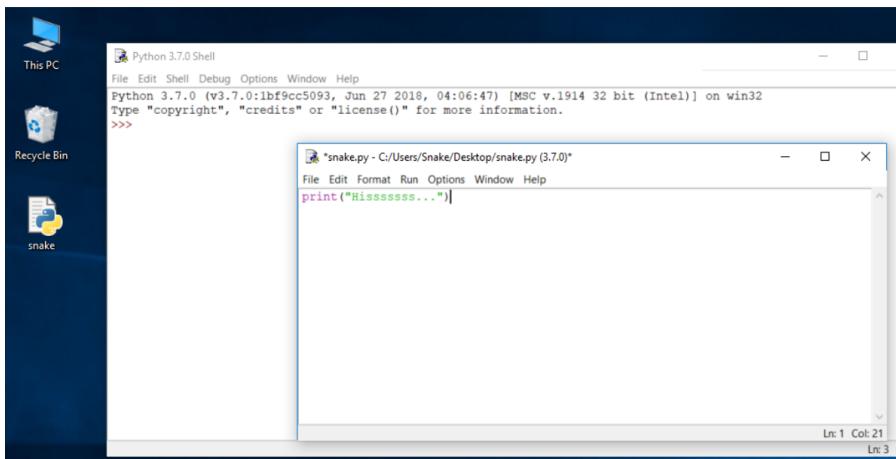


Рисунок 18

Если все идет хорошо и в коде нет ошибок, в окне консоли отобразятся эффекты, вызванные запуском программы.

В этом случае, программа шипит. Попробуйте запустить ее еще раз. И еще раз. Теперь закройте оба окна и вернитесь на рабочий стол.

Урок 1

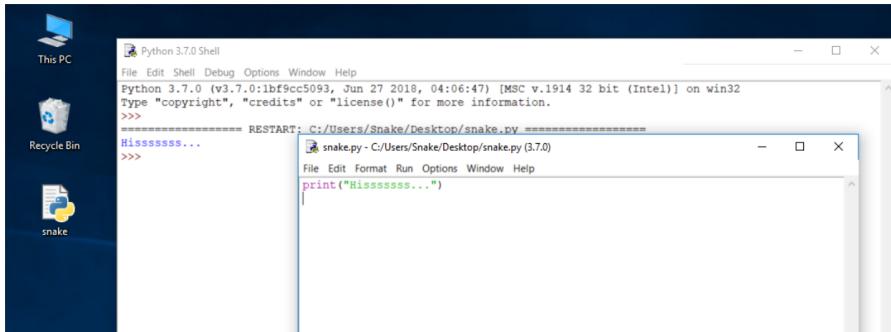


Рисунок 19

Как испортить и исправить свой код

Теперь снова запустите IDLE. Нажмите на [File, Open](#), выберите файл, который вы сохранили ранее, и дайте IDLE прочитать его.

Попробуйте запустить его снова, нажав **F5**, когда окно редактора активно. Как видите, IDLE может сохранять ваш код и извлекать его, когда он вам понадобится снова.

У IDLE есть еще одна полезная функция. Сначала удалите закрывающую скобку. Затем добавьте скобку снова.

Ваш код должен выглядеть, примерно, так:

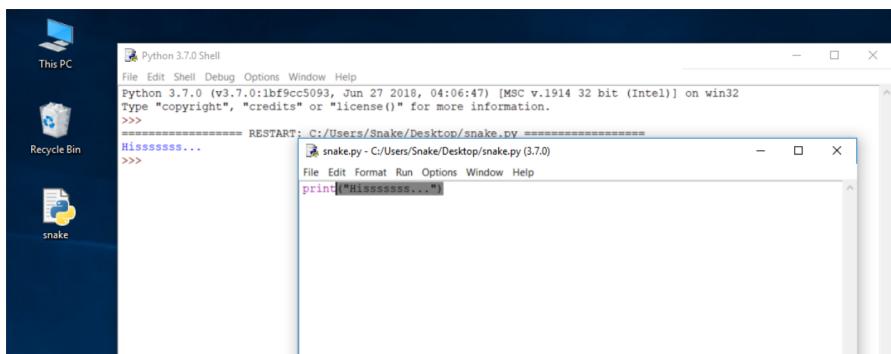


Рисунок 20

Каждый раз, когда вы ставите закрывающую скобку в своей программе, IDLE будет отображать ту часть текста, которая ограничена парой соответствующих скобок. Это позволит вам не забывать добавлять их парами.

Удалите закрывающую скобку еще раз. Код выдает ошибку. Теперь это синтаксическая ошибка. IDLE не должен позволить вам запустить его.

Попробуйте запустить программу еще раз. IDLE напомнит вам сохранить измененный файл. Следуйте инструкциям.

Внимательно смотрите на все окна.

Появится новое окно — оно говорит, что интерпретатор столкнулся с EOF (*end-of-file*, конец файла) хотя (по его мнению) код должен содержать еще какой-то текст.

Окно редактора четко показывает, где это произошло.

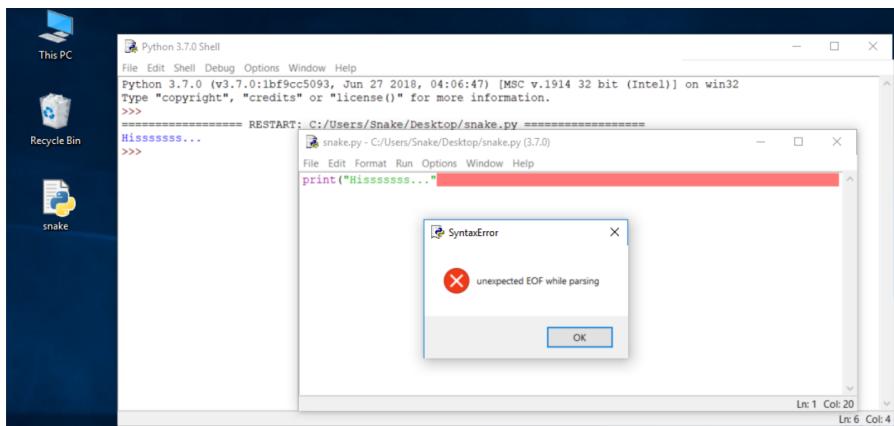


Рисунок 21

Исправьте код. Это должно выглядеть так:

```
print("Hisssssss...")
```

Запустите его, чтобы увидеть, «шипит» ли он снова. Давайте испортим код еще раз. Удалите одну букву из слова **print**. Запустите код, нажав **F5**. Как видите, Python не может распознать ошибку.

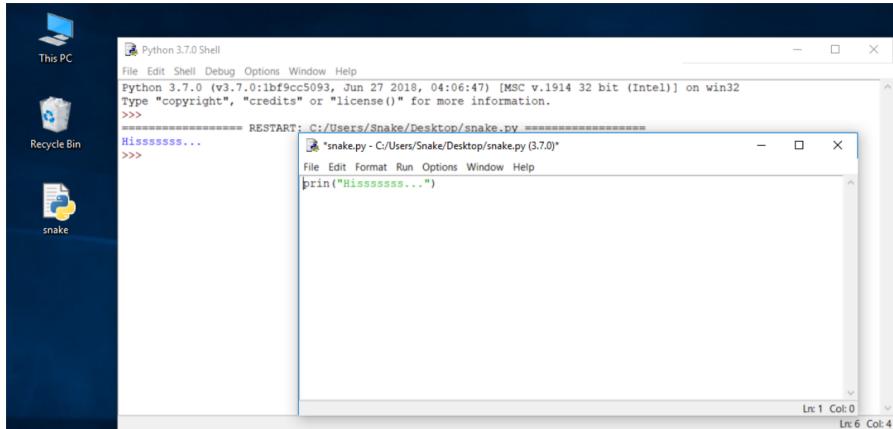


Рисунок 22

Возможно, вы заметили, что сообщение об ошибке, сгенерированное для предыдущей ошибки, сильно отличается от первого.

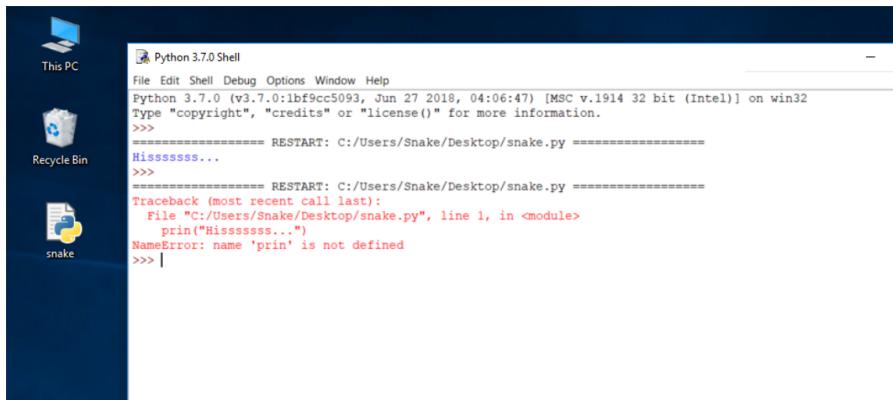


Рисунок 23

Это потому, что природа ошибки другая, и ошибка обнаружена на другом этапе интерпретации.

Окно редактора не даст никакой полезной информации об ошибке, а вот окна консоли могут.

Сообщение (красным) показывает (в последующих строках):

- обратная трассировка (это путь, по которому код проходит через различные части программы. Вы можете пока игнорировать его, поскольку в таком простом коде он пустой);
- местонахождение ошибки (имя файла, содержащего ошибку, номер строки и имя модуля); примечание: цифра может вводить в заблуждение, поскольку Python обычно показывает место, где он впервые замечает последствия ошибки, а не саму ошибку;
- содержание ошибочной строки; примечание: окно редактора IDLE не показывает номера строк, но отображает текущее положение курсора в правом нижнем углу; используйте его, чтобы найти строку с ошибкой в длинном исходном коде;
- название ошибки и краткое объяснение.

Поэкспериментируйте с созданием новых файлов и запуском вашего кода. Попробуйте вывести на экран другое сообщение, например, «roar!», «meow» или даже «oink!». Попробуйте испортить и исправить свой код и посмотрите, что получится.

Типы данных, переменные, основные операции ввода-вывода, основные операторы

Пишем первую программу Hello, World!

Пришло время начать писать реальный, рабочий код на Python. До поры до времени он будет очень простым.

Учитывая, что мы будем разбирать фундаментальные понятия и термины, эти фрагменты кода не будут серьезными или сложными.

Запустите IDLE, создайте новый исходный файл Python, заполните его этим кодом: `print("Hello, World!")`, назовите файл как-нибудь и сохраните его. Теперь запустите код. Если все пойдет хорошо, вы увидите строку в окне консоли IDLE. Код, который вы запустили, должен выглядеть знакомым. Вы видели нечто очень похожее, когда мы настраивали среду IDLE.

Теперь нам нужно потратить какое-то время на объяснение того, что вы на самом деле видите и почему это выглядит именно так.

Как видите, первая программа состоит из следующих частей:

- слово «`print`»;
- открывающая круглая скобка;
- кавычка;
- строка текста: «`Hello, World!`»;

- еще одна кавычка;
- закрывающая круглая скобка.

Каждая из вышеперечисленных составляющих играет очень важную роль в коде.

Функция `print()`

Посмотрите на строку кода ниже:

```
print("Hello, World!")
```

Слово «`print`» — это имя функции. Это не значит, что где бы ни появилось слово, оно всегда будет именем функции. Значение слова происходит из контекста, в котором эир слово было использовано.

Вы, вероятно, неоднократно встречали термин «функция» на уроках математики. А еще вы, наверняка, можете вспомнить несколько названий математических функций, таких как синус или логарифм.

Но функции Python более гибкие и могут вмещать в себя больше, чем их математические братья.

Функция (в этом контексте) — это отдельная часть компьютерного кода, которая:

- вызывает некоторый эффект (например, отправляет текст в терминал, создает файл, рисует изображение, воспроизводит звук и т.д.); это нечто совершенно неслыханное в мире математики;
- вычисляет значение или некоторые значения (например, квадратный корень значения или длины данного текста); это то, что роднит функции Python с математическими понятиями.

Более того, многие функции Python могут выполнять две вышеуказанные функции вместе.

Откуда поступают функции?

- Они могут поступать от самого Python. Функция `print` является одной из них; эта функция — бонус, который вы получаете вместе с Python и его окружением (она встроенная), то есть вам не нужно делать ничего особенного (например, просить кого-либо сделать что-либо), если вам нужно ее использовать;
- они могут происходить от одного или нескольких надстроек Python, которые называются «модули»; некоторые модули поставляются с Python, другие могут требовать отдельной установки, в любом случае все они должны быть явно связаны с вашим кодом (мы скоро покажем вам, как это сделать);
- вы можете и сами написать их, добавляя в вашу программу столько функций, сколько вам нужно, чтобы сделать ее проще, понятнее и элегантнее.

Имя функции должно быть понятным (имя функции `print` самоочевидно, то есть «вывести», «отобразить»).

Конечно, если вы собираетесь использовать уже существующую функцию, вы не можете влиять на ее имя, но когда вы начнете писать свои собственные функции, нужно тщательно продумывать выбор имен.

Как мы уже говорили, функция может иметь:

- эффект;
- результат.

Есть также третий, очень важный, компонент функции — аргумент(-ы).

Математические функции обычно принимают один аргумент, например, `sin(x)` принимает `x`, который является мерой угла. Но функции Python более универсальны. В зависимости от индивидуальных потребностей они могут принять любое количество аргументов — столько, сколько необходимо для выполнения их задач.

Примечание: любое число включает ноль — некоторые функции Python не нуждаются в аргументах.

```
print("Hello, World!")
```

Несмотря на количество необходимых/предоставленных аргументов, функции Python требуют наличия пары скобок — открывающую и закрывающую, соответственно.

Если вы хотите передать функции один или несколько аргументов, нужно поместить их внутри скобок. Если вы собираетесь использовать функцию, не принимающую никаких аргументов, вам все равно нужны круглые скобки.

Примечание: чтобы отличить обычные слова от имен функций, добавьте пару пустых скобок после имени функции, даже если соответствующая функция требует один или несколько аргументов. Это стандартное соглашение.

Здесь мы говорим о функции `print()`. Есть ли у функции `print()` из нашего примера аргументы? Конечно, есть, но кто они?

Единственный аргумент, переданный в функцию `print()` в этом примере, — это строка:

```
print("Hello, World!")
```

Как видите, строка отделяется кавычками. На самом деле, кавычки образуют строку, они вырезают часть кода и присваивают ему другое значение. Вы можете представить, что кавычки говорят что-то вроде: «Текст между нами — не код. Он не предназначен для выполнения, и вы должны принять его как есть».

Почти все, что помещено в кавычки, будет восприниматься буквально, не как код, а как данные. Попробуйте поиграть с этой конкретной строкой: измените ее, введите новое значение, удалите часть существующего значения.

Есть несколько способов указать строку внутри кода Python, но этого пока достаточно.



Рисунок 24

Итак, вы узнали о двух важных частях кода: функции и строке. Мы говорили о них с точки зрения синтаксиса, но теперь пришло время обсудить их с точки зрения семантики.

Имя функции (в этом случае, `print`), вместе с круглыми скобками и аргументом (-ами), образует вызов функции.

Мы скоро обсудим это более подробно, а прямо сейчас разберем только основы.

```
print("Hello, World!")
```

Что происходит, когда Python встречает вызов, подобный приведенному ниже?

```
function_name(argument)
```

Давайте посмотрим:

- Во-первых, Python проверяет, является ли указанное имя разрешенным (он просматривает свои внутренние данные, чтобы найти существующую функцию с этим именем; если поиск не удался, Python прерывает код);
- во-вторых, Python проверяет, позволяет ли условие функции по количеству аргументов вызвать функцию именно таким образом (например, если конкретная функция требует ровно двух аргументов, любой вызов, передающий только один аргумент, будет считаться ошибочным и прервет выполнение кода);
- в-третьих, Python ненадолго оставляет код и переходит в функцию, которую вы хотите вызвать; конечно, он также принимает ваши аргументы и передает их функции;
- в-четвертых, функция выполняет свой код, вызывает желаемый эффект (если он есть), оценивает желаемый результат(-ы) (если они есть) и завершает задачу;
- и, наконец, Python возвращается к вашему коду (в точку сразу после вызова) и возобновляет его выполнение.

Три важных вопроса, на которые нужно ответить как можно скорее:

1. Какой эффект вызывает функция `print()`?

Эффект очень полезный и зреющий. Функция:

- принимает аргументы (может принимать более одного аргумента или менее одного аргумента)
- при необходимости преобразует их в читабельную форму (вы уже наверняка догадались, что строки этого не требуют, так как строка уже и так читабельна)

- и отправляет полученные данные на устройство вывода (обычно консоль); другими словами, все, что вы кладете в функцию `print()`, появится на вашем экране.

Не удивительно, что теперь вы будете часто использовать `print()`, чтобы увидеть результаты ваших операций и вычислений.

2. Каких аргументов ожидает `print()`?

Любых. Мы скоро покажем вам, что функция `print()` способна работать практически со всеми типами данных, которые предлагает Python. Строки, числа, символы, логические значения, объекты — любые из них могут быть успешно переданы в `print()`.

3. Какое значение вычисляет функция `print()`?

Никакое. Ей достаточно того, что она делает, — `print()` ничего не вычисляет.

Функция `print()` — инструкция

Вы уже знаете, что эта программа содержит один вызов функции. В свою очередь, вызов функции является одним из возможных видов инструкции в Python. Следовательно, эта программа состоит только из одной инструкции.

Конечно, любая сложная программа обычно содержит гораздо больше инструкций, чем одну. Вопрос в том, как соединить более одной инструкции в коде Python?

Синтаксис Python довольно специфичен в этой области. В отличие от большинства языков, Python требует, чтобы строка не содержала более одной инструкции.

Строка может быть пустой (т.е. она может вообще не содержать инструкции), но она не должна содержать две, три или более инструкций. Это строго запрещено.

Примечание: Python делает одно исключение из этого правила — он позволяет одной инструкции распространяться на более чем одну строку (что может быть полезно, когда ваш код содержит сложные конструкции).

Давайте немного расширим код, как на примере ниже:

```
print(" The itsy bitsy spider climbed up  
      the waterspout.")  
print("Down came the rain and washed the spider out.")
```

Запустите его и обратите внимание на то, что появилась консоли.

Теперь ваша консоль Python должна выглядеть так:

```
The itsy bitsy spider climbed up the waterspout.  
Down came the rain and washed the spider out.
```

Давайте озвучим некоторые наблюдения:

- программа вызывает функцию `print()` дважды, и теперь в консоли две отдельные строки. Это означает, что `print()` начинает вывод с новой строки каждый раз, когда начинается выполнение; вы можете изменить это поведение или использовать его в своих интересах;
- каждый вызов функции `print()` содержит другую строку, как видно из ее аргумента и содержимого консоли, а это означает, что инструкции в коде выполняются в том же порядке, в котором они были помещены в исходный файл; следующая инструкция не выполняется до тех пор, пока предыдущая не бу-

дет завершена (есть некоторые исключения из этого правила, но их пока можно игнорировать)

Мы немного изменили пример и добавили один пустой вызов функции `print()`.

```
print("The itsy bitsy spider climbed up  
      the waterspout.")  
print()  
print("Down came the rain and washed the spider out.")
```

Мы называем его пустым, потому что мы не представили функции никаких аргументов.

Запустите код. Что происходит? Если все хорошо, вы должны увидеть что-то вроде этого:

```
The itsy bitsy spider climbed up the waterspout.  
Down came the rain and washed the spider out.
```

Как видите, пустой вызов `print()` не такой уж пустой, как могло показаться, раз он выводит пустую строку или (эта интерпретация тоже правильная) его вывод — просто новая строка.

Это не единственный способ добавить новую строку в консоли вывода. Сейчас мы покажем вам другой способ.

Функция `print()` — экранирование символов и переход на новую строку

Мы снова изменили код. Внимательно посмотрите на него.

```
print("The itsy bitsy spider\nclimbed up  
      the waterspout.")
```

```
print()
print("Down came the rain\nand washed the spider out.")
```

В нем два очень маленьких изменения — мы вставили в стишок странную пару символов: `\n`. Интересно, что человек видит два символа, а Python — один.

Обратная косая черта (`\`) имеет особое значение при использовании внутри строк — это называется «экранированием символа» (*escape character*).

Слово «escape» (*побег*) здесь следует понимать буквально — это означает, что последовательность символов в строке ненадолго сбегает (очень ненадолго) и заменяется на специальную подстановку. Другими словами, обратная косая черта сама по себе ничего не значит, это лишь своеобразное объявление о том, что следующий символ после обратной косой черты также имеет другое значение.

Буква «`n`», которая идет за обратной косой чертой, это сокращение от «`newline`» (*новая строка*).

И обратная косая черта, и буква `n` формируют специальный символ, который называется «символ новой строки». Он заставляет консоль начать вывод с новой строки.

Запустите код. Теперь ваша консоль должна выглядеть так:

```
The itsy bitsy spider
climbed up the waterspout.

Down came the rain
and washed the spider out.
```

Как видите, две новые строки появляются в детском стихотворении там, где используется символ `\n`.

У этого соглашения есть два важных последствия:

- Если вы хотите поместить только одну обратную косую черту внутри строки, не забывайте о ее экранирующей природе — вы должны удвоить ее, например, такой вызов вызовет ошибку:

```
print("\\")
```

а этот нет:

```
print("\\\\")
```

- Не все пары в экранировании символов (обратная косая черта в сочетании с другим символом) что-то значат.

Поэкспериментируйте с вашим кодом в редакторе, запустите его и посмотрите, что произойдет.

Функция `print()` – использование нескольких аргументов

До сих пор мы тестировали поведение функции `print()` либо без аргументов, либо с одним аргументом. Стоит также попытаться добавить в функцию `print()` два и больше аргументов.

Посмотрите на код. Вот что мы собираемся протестировать сейчас:

```
print("The itsy bitsy spider" ,  
      "climbed up" ,  
      "the waterspout.")
```

В нем один вызов функции `print()`, но он содержит три аргумента. Все они строки.

Аргументы разделены запятыми. Мы выделили их пробелами, чтобы они стали более заметными, но в этом нет необходимости, и мы больше не будем этого делать.

В этом случае запятые, разделяющие аргументы, играют совершенно иную роль, чем запятая внутри строки. Первый является частью синтаксиса Python, последний предназначен для отображения в консоли.

Если вы посмотрите на код еще раз, вы увидите, что внутри строк нет пробелов.

Запустите код и посмотрите, что произойдет.

Теперь консоль должна отображать следующий текст:

```
The itsy bitsy spider climbed up the waterspout.
```

Пробелы, удаленные из строк, появились снова. Вы можете объяснить, почему?

Из этого примера вытекают два вывода:

- если функция `print()` вызывается с двумя и более аргументами, она выводит их всех в одной строке;
- функция `print()` ставит пробел между выведенными аргументами по собственной инициативе.

Функция `print()` – позиционный способ передачи аргументов

Теперь, когда вы знаете о порядке использования функции `print()`, давайте посмотрим, как его изменять.

Вы уже можете предсказать результат, не запуская этот код в редакторе.

```
print("My name is", "Python.")
print("Monty Python")
```

То, как мы передаем аргументы в функцию `print()` является наиболее распространенным способом в Python и называется позиционными аргументами (название появилось из-за того, что значение аргумента продиктовано его позицией, например, второй аргумент будет выведен после первого, а не наоборот).

Запустите код и проверьте, соответствует ли вывод вашим прогнозам.

Функция `print()` – ключевые аргументы

Python предлагает другой механизм для передачи аргументов, который может быть полезен, если вы хотите, чтобы функция `print()` немного изменила свое поведение.

Мы не будем сейчас это подробно объяснять. Мы планируем сделать это, когда будем говорить о функциях вообще. Сейчас мы просто хотим показать вам, как это работает. Может смело использовать это в своих программах.

Этот механизм называется **ключевые аргументы**. Название отражает тот факт, что значение этих аргументов берется не из его местоположения (позиции), а из специального (ключевого) слова, используемого для их идентификации.

У функции `print()` есть два ключевых аргумента, которые вы можете использовать для своих целей. Первый из них называется `end`.

Ниже приведен очень простой пример использования **ключевого аргумента**:

```
print("My name is", "Python.", end=" ")
print("Monty Python")
```

Прежде чем его использовать, нужно узнать некоторые правила:

- ключевой аргумент состоит из трех элементов: ключевое слово, которое определяет аргумент (тут, `end`); знак равно (`=`); и значение, назначенное на этот аргумент;
- любые ключевые аргументы должны идти после последнего позиционного аргумента (это очень важно).

В нашем примере мы использовали ключевой аргумент `end` и поместили его в строку, содержащую один пробел.

Запустите код, чтобы увидеть, как он работает. Теперь консоль должна отображать следующий текст:

```
My name is Python. Monty Python.
```

Как видите, ключевой аргумент `end` определяет символы, которые функция `print()` отправляет на вывод, как только доходит до конца своих позиционных аргументов.

Поведение по умолчанию отражает ситуацию, когда ключевой аргумент `end` используется косвенно следующим образом: `end="\n"`.

А теперь пришло время попробовать что-то более сложное. Если вы внимательно посмотрите, вы увидите, что мы использовали аргумент `end`, но назначенная ему строка является пустой (она не содержит символов вообще).

Что произойдет дальше? Запустите программу в редакторе, чтобы увидеть.

Из-за того, что аргумент `end` не содержит ничего, функция `print()` тоже ничего не выводит, когда ее позиционные аргументы исчерпаны.

Теперь консоль должна отображать следующий текст:

```
My name is Monty Python.
```

Примечание: новые строки не были отправлены на вывод

Строка, присвоенная ключевому аргументу `end`, может быть любой длины. Поэкспериментируйте с ней, если хотите.

Ранее мы говорили, что функция `print()` разделяет свои выходные аргументы пробелами. Это поведение тоже можно изменить.

Ключевой аргумент, который может это сделать, называется `sep` (от *separator*, разделитель).

Посмотрите на код ниже и запустите его.

```
print("My", "name", "is", "Monty", "Python.", sep="-")
```

Аргумент `sep` дает следующие результаты:

```
My-name-is-Monty-Python.
```

Теперь функция `print()` использует тире вместо пробела для разделения выводимых аргументов.

Примечание: значением аргумента `sep` может быть и пустая строка. Попробуйте сами.

Оба ключевых аргумента могут быть использованы в одном вызове так же, как в коде, приведенном ниже.

```
print("My", "name", "is", sep="_", end="*")
print("Monty", "Python.", sep="*", end="*\n")
```

Пример не имеет смысла, но он наглядно показывает взаимодействие между `end` и `sep`.

Можете ли вы предсказать, что получится на выходе?

Запустите код и посмотрите, соответствует ли он вашим прогнозам.

Теперь, когда вы понимаете функцию `print()`, вы готовы рассмотреть хранение и обработку данных в Python.

Без `print()` вы не сможете увидеть никаких результатов.

Ключевые выводы

- Функция `print()` является встроенной функцией. Она выводит указанное сообщение на экран/консоль.
- Встроенные функции, в отличие от пользовательских функций, всегда доступны и не требуют импорта. У Python 3.7.1 69 встроенных функций. Полный список в алфавитном порядке находится в Стандартной библиотеке Python (*Python Standard Library*).
- Для вызова функции нужно использовать имя функции и круглые скобки. Можно передать аргументы в функцию, поместив их в круглые скобки. Разделите аргументы запятой, например, `print("Hello,","world!")`. «Пустая» функция `print()` выводит пустую строку.
- Строки Python разделены кавычками например, «Я строка» или «Я тоже строка».
- Компьютерные программы — это наборы инструкций. Инструкция — это команда, которая выполняет конкретную задачу при запуске, например, вывод определенного сообщения на экран.
- В строках Python обратная косая черта (`\`) является специальным символом, который объявляет, что сле-

дующий символ имеет другое значение, например, `\n` (*символ новой строки*) начинает вывод с новой строки.

7. Позиционные аргументы — это те, чей смысл определяется их положением, например, второй аргумент выводится после первого, третий выводится после второго и т.д.
8. Ключевые аргументы — это те, чье значение определяется не их местоположением, а специальным (*ключевым*) словом, используемым для их идентификации.
9. Параметры `end` и `sep` могут быть использованы для форматирования вывода функции `print()`. Параметр `sep` указывает на разделитель между выводимыми аргументами (например, `print("H", "E", "L", "L", "O", sep="-")`), тогда как параметр `end` указывает, что выводится в конце оператора `print`.

Литералы Python

Литералы – данные в себе

Теперь, когда вы немного познакомились с некоторыми мощными особенностями функции `print()`, пришло время узнать о новых аспектах и запомнить один важный термин — литерал. Литералы — это данные, значения которых определяются самим литералом.

Поскольку это сложная для понимания концепция, давайте обратимся к примеру. Взгляните на следующий набор цифр:

123

Вы можете угадать, какое значение он представляет? Конечно, можете — сто двадцать три.

Но как насчет этого:

с

У него есть какое-нибудь значение? Возможно. Это может быть символом скорости света, например. Это также может быть постоянной интегрирования. Или даже длиной гипотенузы в рамках теоремы Пифагора. Тут много вариантов.

Не имея дополнительной информации, сложно сказать, какой из них правильный. А вот подсказка: `123` это литерал, а «`с`» — нет.

Литералы используются, чтобы кодировать данные и помещать их в свой код. Теперь нам нужно рассмотреть

некоторые соглашения, которые вы должны соблюдать при программировании на Python.

Давайте начнем с простого эксперимента — посмотрите на фрагмент кода ниже.

```
print("2")
print(2)
```

Первая строка выглядит знакомой. Во второй, кажется, есть ошибка, потому что не хватает кавычек. Попробуйте запустить этот код. Если все прошло хорошо, вы должны увидеть две одинаковые строки.

Что произошло? Что это означает? В этом примере два разных типа литералов:

- строка, с которой вы уже знакомы,
- и целое число, что-то совершенно новое.

Функция `print()` выводит их абсолютно одинаково, и для человека они тоже выглядят одинаковыми, но внутренне, в памяти компьютера, эти два значения хранятся совершенно по-разному: строка существует как просто строка, то есть набор букв.

Число преобразуется в машинное представление (набор битов). Функция `print()` выводит их в понятной для человека форме.

Сейчас мы рассмотрим числовые литералы и чем они живут.

Целые числа (Integers)

Возможно, вы уже что-то знаете о том, как компьютеры выполняют операции с числами. Возможно, вы слышали о двоичной системе счисления и знаете, что это система,

которую компьютеры используют для хранения чисел и что они могут выполнять с ними любые операции.

Мы не будем здесь изучать тонкости позиционных систем счисления, но мы скажем, что числа, которые обрабатывают современные компьютеры, бывают двух типов:

- целые, то есть те, которые лишены дробной части;
- и с плавающей точкой, которые содержат (или могут содержать) дробную часть.

Это определение не совсем точно, но пока что его вполне достаточно. Это различие очень важно, и граница между этими двумя типами чисел очень четкая. Оба типа чисел существенно различаются по тому, как они хранятся в памяти компьютера, и по диапазону допустимых значений.

Характеристика числового значения, определяющая его вид, диапазон и применение, называется типом.

Если закодировать литерал и поместить его в код Python, форма литерала определяет представление (тип), которое Python будет использовать для его хранения в памяти.

А пока отложим числа с плавающей точкой (мы скоро к ним вернемся) и рассмотрим вопрос о том, как Python распознает целые числа.

Этот процесс очень похож на то, как вы бы написали их карандашом на бумаге — это просто ряд цифр, которые составляют число. Но с оговоркой — вы не можете вставлять символы, которые не являются цифрами внутри числа.

Взять, к примеру, число одиннадцать миллионов сто одиннадцать тысяч сто одиннадцать. Если бы вы прямо сейчас взяли в руку карандаш, вы бы написали число следующим образом: [11,111,111](#) или вот так: [11.111.111](#) или даже так: [11 111 111](#).

Понятно, что такая запись облегчает чтение, особенно когда число состоит из большого количества цифр. Тем не менее, Python таких вещей не понимает. Это запрещено. А вот что Python разрешает, так это использование подчёркивания в числовых литералах.

Следовательно, вы можете написать это число так: **11111111** или вот так: **11_111_111**.

Примечание: в Python 3.6 введены подчёркивания в числовых литералах, позволяющие размещать одинарные подчёркивания между цифрами и спецификаторы после основания, чтобы упростить чтение больших чисел. Эта функция недоступна в более старых версиях Python.

А как отрицательные числа кодируются в Python? Как обычно — добавляется минус. Можно написать: **-11111111** или **-11_111_111**.

Положительные числа не обязательно сопровождаются знаком плюс, но это допустимо, если вам это нужно. Следующие строки описывают одно и то же число: **+11111111** и **11111111**.

Целые числа: восьмеричные и шестнадцатеричные числа

В языке программировани Python есть два дополнительных соглашения, которые не известны миру математики. Первое из них позволяет нам использовать числа в восьмеричном представлении.

Если целому числу предшествует префикс **0O** или **0o** (нуль-о), Python будет его воспринимать как восьмеричное

значение. Это означает, что число должен содержать только те цифры, которые находятся в диапазоне [0..7].

0o123 является восьмеричным числом, десятичное значение которого равно 83.

Функция `print()` выполняет преобразование автоматически. Попробуйте запустить этот код:

```
print(0o123)
```

Второе соглашение позволяет нам использовать шестнадцатеричные числа. Таким числам должен предшествовать префикс 0x или 0X (нуль-x).

0x123 это шестнадцатеричное число, десятичное значение которого равняется 291. Функция `print()` может управлять и этими значениями. Попробуйте запустить этот код:

```
print(0x123)
```

Числа с плавающей точкой (Floating-point numbers)

Теперь поговорим о другом типе, который предназначен для представления и хранения чисел, которые (как сказал бы математик) имеют непустую десятичную дробь.

Это числа, у которых есть (или может быть) дробная часть после десятичной точки, и хотя такое определение очень плохое, этого, безусловно, достаточно для того, что мы будем разбирать.

Каждый раз, когда мы используем такой термин, как «два с половиной» или «минус нуль целых четыре», мы

имеем в виду числа, которые компьютер воспринимает как числа с плавающей точкой: **2.5 -0.4**.

Примечание: число два с половиной выглядит абсолютно нормально, когда вы записываете его в программе, но обязательно убедитесь, что в вашем числе нет других запятых.

Python либо не сможет его прочитать вообще или (в очень редких, но возможных случаях) может неправильно вас понять, так как у самой запятой есть собственное значение, зарезервированное в Python.

Если вы хотите использовать просто значение два с половиной, вы должны написать его так, как показано выше. Обратите внимание еще раз — между **2** и **5** стоит точка, не запятая.

Как вы уже, наверное, догадались, число нуль целых четыре может быть написано на Python так: **0.4**.

Но не забывайте одно простое правило — вы можете опустить нуль, если это единственная цифра перед или после десятичной запятой.

По сути, вы можете написать значение **0.4** так: **.4**. А значение **4.0** может быть написано так: «**4.**». Это не изменит ни его тип, ни его значение.

Десятичный разделитель крайне важен для распознавания чисел с плавающей точкой в Python. Посмотрите на эти два числа: **4, 4.0**.

Можно подумать, что они абсолютно одинаковы, но Python видит их совершенно по-разному. **4** это целое число, тогда как **4.0** это число с плавающей точкой. Плавает именно точка.

С другой стороны, плавают не только точки. Буква **e** имеет такой же эффект.

Если вы хотите использовать очень большие или очень маленькие числа, вы можете использовать экспоненциальное представление числа.

Взять, к примеру, скорость света, выраженную в метрах в секунду. Если записать ее буквально, это будет выглядеть так: **300000000**. Чтобы не записывать так много нулей, в учебниках по физике используется сокращенная форма записи, которую вы, вероятно, уже видели: **3×10^8** . Читается это так: три, умноженное на десять в восьмой степени.

В Python тоже достигается немного другим способом: **3E8**. Буква **E** (можно использовать и строчную букву **e**, от слова «*exponent*», степень) — это краткая запись фразы «столько-то, умноженное на десять в такой-то степени».

Примечание:

- *степень (значение после E) должно быть целым числом;*
- *основание (значение перед E) может быть целым числом.*

Кодирование чисел с плавающей точкой

Давайте посмотрим, как это соглашение используется для записи очень маленьких чисел (в смысле их абсолютного значения, близкого к нулю).

Физическая константа называется постоянной Планка (и обозначается как **h**), согласно учебникам, она составляет: **$6,62607 \times 10^{-34}$** .

Если вы хотите использовать ее в программе, то записывается она так:

6.62607E-34

Примечание: тот факт, что вы выбрали одну из возможных форм представления значений с плавающей точкой, не означает, что Python представит ее в такой же форме.

Иногда Python выбирает другую форму записи. Например, допустим, вы решили использовать следующий литерал с плавающей точкой:

0.00000000000000000000000000000001

Когда вы запустите этот литерал через Python:

```
print(0.0000000000000000000000000001)
```

результат будет таким:

1e-22

Python всегда выбирает более экономичную форму представления числа, и вы должны учитывать это при создании литералов.

Строки

Строки используются, когда нужно обработать текст (например, разные имена, адреса и т. д.), а не цифры.

Вы уже немного знаете о них, например, что строкам нужны кавычки, точно так же и числам с плавающей точкой нужны точки.

Вот абсолютно стандартная строка:

"Я строка".

Но здесь есть одна загвоздка. Она заключается в том, как закодировать кавычку внутри строки, если она уже разделена кавычками.

Например, мы хотим напечатать очень простое сообщение:

```
I like "Monty Python"
```

Как мы это сделаем без ошибок? Есть два возможных решения.

Первое основано на уже известной нам концепции экранирования символов, которое вводится при помощи обратной косой черты. Обратная косая черта может экранировать и кавычки. Кавычка, которой предшествует обратная косая черта, меняет свое значение — это не разделитель, а просто кавычка. Так что это сработает, как и должно:

```
print("I like \"Monty Python\"")
```

Примечание: *внутри строки есть две экранированные кавычки — вы все видите?*

Второе решение неординарное. Python может использовать апостроф вместо кавычки. Любой из этих символов может разделять строки, но будьте последовательны. Если вы открываете строку с кавычкой, вы должны и закрыть ее с кавычкой. Если вы начинаете строку с апострофа, вы и заканчивать ее должны апострофом.

Этот пример тоже будет работать:

```
print('I like "Monty Python"')
```

Примечание: тут экранирование не нужно.

Кодирование строк

Теперь следующий вопрос: как вставить апостроф в строку, если она уже окружена апострофами?

Вы уже должны знать ответ, или, если быть точным, два возможных ответа.

Попробуйте вывести строку, содержащую следующее сообщение:

```
I'm Monty Python.
```

Как видите, обратная косая черта — очень мощный инструмент, который может экранировать не только кавычки, но и апострофы.

Мы уже это продемонстрировали, но давайте еще раз сделаем акцент на этом явлении — строка может быть пустой, то есть в ней может вообще не быть символов.

Пустая строка — все еще строка:

```
''  
'''
```

Булевые значения (логические типы данных)

Чтобы закончить тему литералов в Python, давайте рассмотрим еще два. Они не так очевидны, как предыдущие, так как используются для представления очень абстрактного значения — правдивости.

Каждый раз, когда вы спрашиваете Python, является ли одно число больше, чем другое, этот вопрос приводит к созданию некоторых конкретных данных — логических (булевых) значений.

Джордж Буль (1815–1864) — автор фундаментальной работы «Исследование законов мышления», в которой дано определение **булевой алгебры** — это раздел алгебры, использующий только два значения: правда и ложь, которые обозначаются как **1** и **0**, соответственно.

Программист пишет программу, а программа задает вопросы. Python выполняет программу и дает ответы. Программа должна уметь реагировать в соответствии с полученными ответами. К счастью, компьютеры знают только два вида ответов:

- Да, это правда;
- Нет, это ложь.

Вы никогда не получите ответ типа: «Я не знаю» или «Наверное, да, но точно не знаю». Python — двоичная рептилия. У этих логических значений есть строгие обозначения в Python: **True** (*Правда*) и **False** (*Ложь*)

Вы не можете ничего изменить — вы должны принять эти символы такими, какие они есть, включая регистр.

Задача: Каким будет результат следующего фрагмента кода?

```
print(True > False) print(True < False)
```

Запустите код, чтобы проверить. Вы можете объяснить результат?

Ключевые выводы

1. **Литералы** — запись, которая используется для представления некоторых фиксированных значений в коде. В Python есть разные типы литералов, например, литерал может быть числом (числовые литералы: `123`) или строкой (строковые литералы: «`I am a literal`»).
2. **Двоичная система** — это система чисел с основанием `2`. Поэтому двоичное число может состоять только из `0` и `1`, например, `1010` — это `10` в десятичной системе счисления.
Основания восьмеричной и шестнадцатеричной систем счисления — `8` и `16`, соответственно. Шестнадцатеричная система использует десятичные числа и шесть дополнительных букв.
3. **Целые числа (`integers`, `ints`)** — один из числовых типов, который поддерживает Python. Это числа, написанные без дробной составляющей, например, `256` или `-1` (отрицательные целые числа).
4. **Числа с плавающей точкой (`floats`)** — еще один числовой тип, который поддерживает Python. Это числа, которые содержат (или могут содержать) дробную часть, например, `1.27`.
5. Чтобы закодировать апостроф или кавычку внутри строки, можно использовать экранирование символа, например, `'I\'m happy.'` или открывающие и закрывающие символы, противоположные тем, которые вы хотите кодировать, например, в `"I'm happy"` закодирован апостроф, а в `'He said "Python", not "typhoon'"` закодирована (двойная) кавычка.

6. Логические (булевы) значения — это значения `True` и `False`, которые используются для представления истинности значений (в числовом выражении `1` — Правда, а `0` — Ложь).

Есть еще один специальный литерал, который используется в Python — `None`. Этот литерал является так называемым `NoneType`-объектом и используется для представления отсутствие значения. Скоро мы обсудим его подробнее.

Операторы — инструменты управления данными

Python как калькулятор

Теперь мы покажем вам совершенно новую сторону функции `print()`. Вы уже знаете, что эта функция может показать вам значения литералов, которые передаются ей через аргументы. На самом деле, она может гораздо больше. Посмотрите на фрагмент кода:

```
print(2+2)
```

Скопируйте код в редактор и запустите его. Вы сможете угадать, что появится на экране?

Должна быть цифра четыре. Не стесняйтесь экспериментировать с другими операторами.

Вот так легко и просто вы только что обнаружили, что Python можно использовать как калькулятор. Не самый удобный и, конечно, не карманный, но, тем не менее, калькулятор.

Если взглянуть на него чуть более серьезно, мы сейчас входим в область операторов и выражений.

Основные операторы

Оператор — это символ языка программирования, который оперирует значениями.

Например, как и в арифметике, знак `+` (плюс) — это оператор, который может сложить два числа и выдать

результат сложения. Однако не все операторы Python так же очевидны, как знак сложения, поэтому давайте рассмотрим некоторые операторы, доступные в Python, и объясним, какие правила определяют их использование и как интерпретировать выполняемые ими операции.

Начнем с операторов, которые связаны с наиболее широко известными арифметическими операциями:

```
+, -, *, /, //, %, **
```

Их порядок в этой строке не случаен. Мы поговорим об этом подробнее, когда пройдем их все.

Запомните! Данные и операторы при соединении образуют выражения. Самым простым выражением является сам литерал.

Арифметические операторы: возвведение в степень

Знак ****** (двойная звездочка) — оператор возвведения в степень. Его левый аргумент является основой, а правый — степенью.

Классическая математика предпочитает запись с верхним индексом, вот такую: **23**. Чистые текстовые редакторы не понимают такой записи, поэтому Python использует ****** вместо нее, например, **2 ** 3**.

Посмотрите на примеры ниже.

```
print (2 ** 3)
print (2 ** 3.)
print (2. ** 3)
print (2. ** 3.)
```

Примечание: в этом примере мы окружили двойные звездочки пробелами. Это не обязательно, но это улучшает читабельность кода.

Примеры показывают очень важную особенность практически всех числовых операторов в Python.

Запустите код и внимательно посмотрите на результат. Есть ли здесь какая-то закономерность?

Запомните! На основе этого результата можно сформулировать следующие правила:

- если аргументы по обе стороны от `**` целые числа, результат тоже будет целым числом;
- если хотя бы один аргумент `**` будет числом с плавающей точкой, результат тоже будет числом с плавающей точкой.

Это очень важно, поэтому помните об этом.

Арифметические операторы: умножение

Знак `*` (звездочка) — оператор умножения.

Запустите приведенный ниже код и проверьте, работает ли правило целых чисел и чисел с плавающей точкой.

```
print(2 * 3)
print(2 * 3.)
print(2. * 3)
print(2. * 3.)
```

Арифметические операторы: деление

Знак `/` (косая черта) — оператор деления.

Значение перед косой чертой — делимое, значение после косой черты — делитель.

Запустите приведенный ниже код и проанализируйте результаты.

```
print(6 / 3)
print(6 / 3.)
print(6. / 3)
print(6. / 3.)
```

Вы должны увидеть, что есть исключение из правила. Результат, полученный оператором деления, всегда является числом с плавающей точкой независимо от того, будет ли результат на первый взгляд десятичным числом: `1/2` или он скорее будет целым числом: `2/1`.

Может ли это быть проблемой? Да. Иногда случается так, что вам очень нужно деление, которое дает целочисленное значение, а не дробь.

К счастью, Python может и с этим справиться.

Арифметические операторы: целочисленное деление

Знак `//` (двойная косая черта) — оператор целочисленного деления. Он отличается от стандартного оператора `/` в двух мелочах:

- у его результата нет дробной части. Она отсутствует (для целых чисел) или всегда равна нулю (для чисел с плавающей запятой); это значит, что результаты всегда округляются;
- это соответствует правилу целых чисел и чисел с плавающей точкой.

Запустите пример приведенный ниже и посмотрите на результаты:

```
print(6 // 3)
print(6 // 3.)
print(6. // 3)
print(6. // 3.)
```

Как видите, если разделить целое число на целое число, то и результат будет целым числом. Во всех остальных случаях вы получите дробь.

Давайте сделаем несколько более сложных тестов. Посмотрите на следующий фрагмент кода:

```
print(6 // 4)
print(6. // 4)
```

Представим, что мы использовали `/` вместо `//`. Вы можете предсказать результат? Да, получится `1.5` в обоих случаях. Это понятно. Но каких результатов нам ждать с делением через оператор `//`? Запустите код и посмотрите сами. Мы получим два числа — одно целое и одно с плавающей точкой.

Результат целочисленного деления всегда округляется до ближайшего целочисленного значения, которое меньше реального (не округленного) результата. **Это очень важно:** округление всегда идет к меньшему целому числу.

Посмотрите на код ниже и попробуйте спрогнозировать результаты еще раз:

```
print(-6 // 4)
print(6. // -4)
```

Примечание: некоторые значения являются отрицательными. Очевидно, это повлияет на результат. Но как?

Результат — две отрицательные двойки. Реальный (не округленный) результат **-1.5** в обоих случаях. Тем не менее, результаты всегда округляются. Округление идет к меньшему целому значению, а меньшее целочисленное значение — это **-2**, отсюда: **-2** и **-2.0**.

Операторы: остаток (деление по модулю, с остатком)

Следующий оператор довольно своеобразен, потому что он не имеет эквивалента среди традиционных арифметических операторов. Его графическое представление в Python выглядит так: **%** (знак процента), который может сбить с толку неподготовленного человека. Постарайтесь думать об этом как о косой черте (*операторе деления*), которая окружена двумя маленькими кружочками.

Результатом оператора является остаток после целочисленного деления. Другими словами, это значение, которое остается после деления одного значения на другое, чтобы получить целочисленное частное.

Примечание: в других языках программирования этот оператор иногда называют делением по модулю.

Посмотрите на фрагмент кода ниже и попытайтесь предсказать его результат, а затем запустите его:

```
print(14 % 4)
```

Как видите, результат будет 2. И вот почему:

- $14 // 4$ будет 3 — это целое число, частное;
- $3 * 4$ будет 12 — как результат умножения частного на делитель;
- $14 - 12$ будет 2 — это остаток.

Следующий пример несколько сложнее:

```
print(12 % 4.5)
```

Каким будет результат?

Ответ: 3.0 — не 3, а 3.0 (правило все еще работает: $12 // 4.5$ будет 2.0; $2.0 * 4.5$ будет 9.0; 12 — 9.0 будет 3.0)

Операторы: как не делить

Как вы, наверное, знаете, деление на ноль не работает. Не пытайтесь делать следующее:

- делить на ноль;
- делить целое число на ноль;
- находить остаток от деления на ноль.

Операторы: суммирование

Оператор суммирования — это знак + (плюс), который полностью соответствует математическим стандартам.

Опять же, взгляните на фрагмент программы ниже:

```
print(-4 + 4)
print(-4. + 8)
```

Результат не должен вас удивить. Запустите код, чтобы проверить это.

Оператор вычитания, унарные и бинарные операторы

Оператором вычитания, очевидно, будет знак «`-`» (минус), хотя нужно заметить, что этот оператор также имеет другое значение — он может изменить знак числа.

Это прекрасная возможность представить очень важное различие между унарными и бинарными операторами.

В операциях вычитания оператор минус принимает два аргумента: левый (уменьшаемое число, если использовать математический термин) и правый (вычитаемое).

Поэтому оператор вычитания считается одним из бинарных операторов, так же как операторы сложения, умножения и деления.

Но оператор минус может быть использован и другим (унарным) способом. Посмотрите на последнюю строку фрагмента ниже:

```
print(-4 - 4)
print(4. - 8)
print(-1.1)
```

Кстати: там есть и унарный оператор `+`. Его можно использовать так:

```
print(+2)
```

Оператор сохраняет знак своего единственного аргумента справа.

Хотя такая конструкция синтаксически правильна, ее использование не имеет особого смысла, и будет трудно придумать причину его использовать.

Посмотрите на фрагмент кода выше — вы можете предсказать, что будет выведено на экран?

Операторы и их приоритеты

До сих пор мы рассматривали каждый оператор так, будто он не связан с остальными. Конечно, такая идеальная и простая ситуация это редкость в реальном программировании.

Кроме того, очень часто можно встретить более одного оператора в одном выражении, и тогда предсказание результата уже не столь очевидно.

Рассмотрим следующее выражение:

```
2 + 3 * 5
```

Вы, наверное, помните со школы, что сначала идет умножение, а потом сложение.

И вы наверняка помните, что сначала нужно умножить 3 на 5, запомнить результат — 15, после этого прибавить к 15 2, и в итоге получить 17.

Феномен, который заставляет некоторые операторы действовать раньше других, называется иерархией приоритетов.

Python точно определяет приоритеты всех операторов и предполагает, что операторы с более высоким приоритетом выполняют свои операции раньше операторов с более низким приоритетом.

Итак, если вы знаете, что у умножения `*` более высокий приоритет, чем у сложения `+`, результат должен быть для вас очевидным.

Операторы и связывание

Связывание оператора определяет порядок вычисления операторов с равным приоритетом, которые идут подряд в одном выражении.

Большинство операторов в Python имеют левостороннее связывание, что означает, что вычисление выражения производится слева направо.

Этот простой пример покажет вам, как это работает. Посмотрите:

```
print(9 % 6 % 2)
```

Есть два возможных способа оценки этого выражения:

- слева направо: первый **9 % 6** дает **3**, затем **3 % 2** дает **1**;
- справа налево: первый **6 % 2** дает **0**, затем **9 % 0** выдает фатальную ошибку.

Запустите пример и посмотрите, что получится.

Результат должен быть **1**. У этого оператора левостороннее связывание. Но есть одно интересное исключение.

Операторы и связывание: возвведение в степень

Повторите эксперимент, но теперь с возведением в степень.

Используйте этот фрагмент кода:

```
print(2 ** 2 ** 3)
```

Два возможных результата:

- **2 ** 2 → 4; 4 ** 3 → 64**
- **2 ** 3 → 8; 2 ** 8 → 256**

Запустите код. Что вы видите? Результат ясно показывает, что оператор возвведения в степень использует правостороннее связывание.

Список приоритетов

Поскольку вы только знакомитесь с операторами в Python, мы не хотим сейчас давать вам полный список приоритетов операторов.

Вместо этого мы покажем вам его сокращенную форму и будем последовательно ее расширять по мере появления новых операторов.

Посмотрите на таблицу ниже:

Приоритет	Оператор	
1	+, -	унарный
2	**	
3	*, /, %	
4	+, -	бинарный

Примечание: мы перечислили операторы по порядку от самого высокого (1) до самого низкого (4) приоритета.

Попробуйте проработать следующее выражение:

```
print(2 * 3 % 5)
```

У обоих операторов (`*` и `%`) одинаковый приоритет, поэтому результат можно предсказать только в том случае, если вы знаете направление связывания. Как вы думаете? Каким будет результат?

Ответ: 1.

Операторы и скобки

Конечно, всегда можно использовать скобки, которые могут изменить естественный порядок вычислений.

В соответствии с арифметическими правилами, подвыражения в скобках всегда вычисляются первыми.

Вы можете использовать столько скобок, сколько вам нужно, и они часто используются для улучшения читабельности выражения, даже если они не меняют порядок операций.

Пример выражения с несколькими круглыми скобками приведен здесь:

```
print((5 * ((25 % 13) + 100) / (2 * 13)) // 2)
```

Попробуйте сами посчитать значение, которое будет выведено на консоль. Каким будет результат функции `print()`?

Ответ: **10.0**.

Ключевые выводы

1. **Выражение** — это комбинация значений (или переменных, операторов, вызовов функций — вы скоро узнаете о них), которая вычисляет значение, например, **1 + 2**.
2. **Операторы** являются специальными символами или ключевыми словами, которые могут работать со значениями и выполнять (математические) операции, например, оператор ***** умножает два значения: **x * y**.
3. Арифметические операторы в Python: **+** (*сложение*), **-** (*вычитание*), ***** (*умножение*), **/** (*классическое деление* —

возвращает значение с плавающей точкой, если одно из значений является числом с плавающей точкой), `%` (деление по модулю — делит левый операнд на правый операнд и возвращает остаток операции, например, `5 % 2 = 1`), `**` (возведение в степень — левый операнд возводится в степень правого операнда, например, `2 ** 3 = 2 * 2 * 2 = 8`), `//` (целочисленное деление — возвращает число, полученное в результате деления, но округленное до ближайшего целого числа, например, `3 // 2.0 = 1.0`).

4. Унарный оператор — это оператор только с одним операндом, например, `-1` или `+3`.
5. Бинарный оператор — это оператор с двумя операндами, например, `4 + 5` или `12 % 5`.
6. Некоторые операторы выполняются раньше других — иерархия приоритетов:
 - у унарных `+` и `-` наивысший приоритет
 - затем идет `**`, затем `*`, `/`, и `%`, а уже потом самый низкий приоритет: двоичный `+` и `-`.
7. Подвыражения в скобках всегда вычисляются первыми, например, `15 — 1 * (5 * (1 + 2)) = 0`.
8. Оператор возведения в степень использует правостороннее связывание, например, `2 ** 2 ** 3 = 256`.

Переменные — поля в форме данных

Что такое переменные?

Кажется довольно очевидным, что Python должен разрешать кодирование литералов, которые содержат и числа, и текстовые значения.

Вы уже знаете, что с этими числами можно выполнять некоторые арифметические операции: сложение, вычитание и т. д. Вы будете использовать их неоднократно.

Абсолютно логичным будет вопрос о том, как хранить результаты этих операций так, чтобы использовать их в других операциях и так далее.

Как сохранить промежуточные результаты и использовать их снова для получения последующих?

Python справится и с этим. Он предлагает специальные «контейнеры», которые называются «переменными» — само название предполагает, что содержимое этих контейнеров может быть изменено (почти) любым способом.

Что есть в каждой переменной Python?

- имя;
- значение (содержимое контейнера).

Давайте начнем с вопросов, связанных с именем переменной.

Переменные не появляются в программе автоматически. Как разработчик, вы должны решить, сколько

и какие переменные использовать в ваших программах и присвоить им имя.

Если хотите дать переменной имя, вы должны соблюдать некоторые строгие правила:

- имя переменной должно состоять из прописных или строчных букв, цифр и символа подчеркивания _;
- имя переменной должно начинаться с буквы;
- символ подчеркивания — это буква;
- большие и маленькие буквы воспринимаются по-разному (не так, как в реальном мире — Алиса и АЛИСА — одно и то же имя, но в Python это два разных имени переменных и, следовательно, две разные переменные);
- имя переменной не должно совпадать с зарезервированными словами в Python (ключевые слова, о которых мы скоро расскажем подробнее).



Рисунок 9

Правильные и неправильные имена переменных

Обратите внимание, что те же ограничения относятся и к именам функций.

Python не накладывает ограничений на длину имен переменных, но это не означает, что длинное имя переменной всегда лучше короткого.

Вот несколько примеров правильных, но неудобных имен переменных:

```
MyVariable, i, t34, Exchange_Rate,
    counter, days_to_christmas,
    TheNameIsSoLongThatYouWillMakeMistakesWithIt, _.
```

Кроме того, Python позволяет использовать не только латинские буквы, но и символы, которые принадлежат к другим алфавитам.

Эти имена переменных также правильны:

```
Adiós_Señora, súr_la_mer, Einbahnstraße, переменная.
```

А теперь приведем примеры неправильных имен: **10т** (начинается не с буквы), **Exchange Rate** (есть пробел).

Примечание. PEP 8 — руководство по коду на Python рекомендует следующее соглашение об именовании переменных и функций в Python:

- Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания — это необходимо, чтобы их было легче прочитать (например, *var*, *my_variable*).
- Имена функций соблюдают те же правила, что и имена переменных (например, *fun*, *my_function*).

- Стиль *mixedCase* (смешанный регистр, например, *myVariable*) допускается в тех местах, где уже преобладает такой стиль, для сохранения обратной совместимости.

Ключевые слова

Посмотрите на список слов, которые играют особую роль в каждой программе Python.

```
['False', 'None', 'True', 'and', 'as', 'assert',
'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global',
;if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try',
'while', 'with', 'yield']
```

Они называются «ключевые слова» или точнее — «зарезервированные ключевые слова». Они называются зарезервированными, потому что их нельзя использовать как имена ни для переменных, ни для функций или любых других именованных сущностей, которые вы будете создавать.

Значение зарезервированного слова определено заранее и не должно никак изменяться.

К счастью, из-за того, что Python чувствителен к регистру, вы можете изменить любое из этих слов, изменив регистр любой буквы, и создать, таким образом, новое слово, которое больше не считается зарезервированным.

Например, переменную нельзя назвать вот так:

```
import
```

Это запрещено, но вы можете назвать ее так:

Import

Возможно, вы не понимаете значение этого сейчас, но мы скоро во всем разберемся.

Создание переменных

Что можно поместить в переменную? Что угодно. Переменная может хранить любое значение любого типа данных.

Значение переменной — это то, что вы положили в нее. Оно может меняться так часто, как вам нужно. Сначала она может быть целым числом, а через секунду — числом с плавающей точкой, которое в итоге становится строкой.

Теперь поговорим о двух важных вещах — как создаются переменные и как поместить в них значение (точнее, как им передать значения).

ЗАПОМНИТЕ

- Переменная возникает в результате присвоения ей значения. В отличие от других языков программирования, вам не нужно об этом как-то по-особенному объявлять.
- Если вы присвоите какое-либо значение несуществующей переменной, переменная автоматически создается. Больше вам ничего не нужно делать.
- Создание (или синтаксис) предельно простое: используйте имя нужной переменной, затем знак равенства (=) и значение, которое вы хотите поместить в переменную.

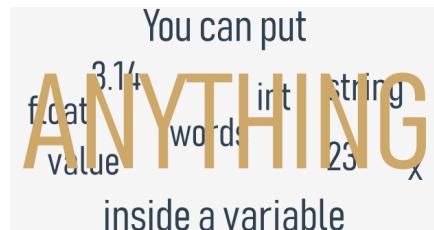


Рисунок 10

Посмотрите на этот фрагмент кода:

```
var = 1  
print(var)
```

Он состоит из двух простых инструкций:

- Первая создает переменную с именем `var` и назначает литерал с целочисленным значением, которое равно 1.
- Вторая выводит значение переменной в консоль.

Примечание: у функции `print()` есть еще одна особенность — она может обрабатывать и переменные. Вы знаете, каким будет вывод этого фрагмента кода?

Ответ: 1.

Использование переменных

Можно использовать столько объявлений переменных, сколько вам нужно для достижения цели, например:

```
var = 1  
account_balance = 1000.0  
client_name = 'John Doe'  
print(var, account_balance, client_name)  
print(var)
```

Нельзя использовать переменную, которой не существует (другими словами, переменную, которой не присвоено значение).

Код в этом примере будет заканчиваться с ошибкой:

```
var = 1
print(Var)
```

Мы пытались использовать переменную с именем **Var**, которой не присвоено значение.

Примечание: *var* и *Var* разные сущности, и не имеют ничего общего с точки зрения Python).

Примечание! Можно использовать выражение *print()* и объединить текст и переменные, используя оператор *+* для вывода строк и переменных, например:

```
var = "3.7.1"
print("Python version: " + var)
```

Вы можете предсказать вывод фрагмента кода выше?

Ответ: [Python version: 3.7.1.](#)

Присвоение нового значения уже существующей переменной

Как присвоить новое значение уже созданной переменной? Точно так же. Просто нужно использовать знак равенства.

Знак равенства — это, на самом деле, оператор присваивания. Хотя это и может показаться странным,

у этого оператора простой синтаксис и однозначная интерпретация.

Он присваивает значение своего правого аргумента левому, в то время как правый аргумент может быть произвольно сложным выражением, включающим литералы, операторы и уже определенные переменные.

Посмотрите на код ниже:

```
var = 1 print(var)
var = var + 1
print(var)
```

Код выводит две строки в консоли:

```
1
2
```

Первая строка фрагмента кода создает новую переменную, которая называется `var` и присваивает ей значение `1`.

Выражение означает следующее: присвоить значение `1` переменной с именем `var`. Можно сказать короче: присвоить `var 1`. Кто-то предпочитает читать такое утверждение следующим образом: `var` становится `1`.

Третья строка присваивает ту же переменную с новым значением, которое берется из самой переменной, и суммируется с `1`. Если бы математик увидел такую запись, он, наверное, возразил бы — ни одно значение не может быть равным самому себе плюс единица. Это противоречие. Но Python воспринимает знак `=` не как «равно», а как «присвоить значение».

Итак, как вы прочитаете такую запись в программе? Принять текущее значение переменной `var`, прибавить к ней **1** и сохранить результат в переменной `var`.

По сути, значение переменной `var` увеличили на единицу, которая не имеет ничего общего со сравнением переменной с каким-либо значением.

Знаете ли вы, каким будет вывод следующего фрагмента кода?

```
var = 100
var = 200 + 300
print(var)
```

Ответ: **500** — почему? Ну, во-первых, переменная `var` создается и ей присваивается значение **100**. Затем той же переменной присваивается новое значение: результат суммирования **200** и **300**, который равен **500**.

Решение простых математических задач

Сейчас вы уже можете создать короткую программу, которая решает простые математические задачи, например, теорему Пифагора: «Квадрат гипотенузы равен сумме квадратов двух других сторон».

Следующий код вычисляет длину гипотенузы (т.е. самую длинную сторону прямоугольного треугольника), используя теорему Пифагора:

```
a = 3.0
b = 4.0
c = (a ** 2 + b ** 2) ** 0.5
print("c =", c)
```

Примечание: нам нужно использовать оператор `**`, чтобы найти квадратный корень таким образом:
 $\sqrt{x} = x^{(1/2)}$

и таким:

$$c = \sqrt{a^2 + b^2}.$$

Вы можете предсказать вывод кода? Ответ находится ниже, но сначала запустите код в редакторе, чтобы подтвердить свои прогнозы.

Ответ: `c = 5.0.`

Сокращенные формы записи

Настало время для следующего набора операторов, которые помогают разработчику. Нам нужно использовать одну и ту же переменную как справа, так и слева от оператора `=`. Например, если нам нужно вычислить серию последовательных значений степеней двойки, можно использовать такой фрагмент:

```
x = x * 2
```

Вы можете использовать подобное выражение, если не можете уснуть и пытаетесь считая овец:

```
sheep = sheep + 1
```

Python предлагает вам сокращенную форму записи таких операций, которые можно кодировать следующим образом:

```
x *= 2
sheep += 1
```

Попробуем представить общее описание этих операций.

Если **оп** это оператор с двумя аргументами (это очень важное условие), и оператор используется в следующем контексте:

```
переменная = переменная оп выражение
```

Его можно упростить и записать следующим образом:

```
переменная оп= выражение
```

Посмотрите на примеры ниже. Убедитесь, что вы понимаете их все.

```
i = i + 2 * j ⇒ i += 2 * j
var = var / 2 ⇒ var /= 2
rem = rem % 10 ⇒ rem %= 10
j = j - (i + var + rem) ⇒ j -= (i + var + rem)
x = x ** 2 ⇒ x **= 2
```

Ключевые выводы

1. Переменная является именованным местом, зарезервированным для хранения значений в памяти. Переменная создается или инициализируется автоматически, когда вы присваиваете ей значение в первый раз.
2. У каждой переменной должно быть уникальное имя — идентификатор. Имя допустимого идентификатора должно быть непустой последовательностью символов, оно должно начинаться с нижнего подчеркивания () или буквы, и оно не может быть ключевым словом, зарезервированным в Python. За первым символом

могут идти подчеркивания, буквы и цифры. Идентификаторы в Python чувствительны к регистру.

3. Python является динамически типизированным языком программирования, а это значит, что вам не нужно объявлять переменные. Чтобы присвоить значения переменным, можно использовать простой оператор присваивания в виде знака равно (=), т.е. `var = 1`.
4. Также можно использовать составные операторы присваивания (операторы с сокращенной формой записи) для изменения значений, назначенных переменным, например, `var += 1` или `var /= 5 * 2`.
5. Можно присвоить новые значения уже существующим переменным, используя оператор присваивания или один из составных операторов, например:

```
var = 2
print(var)

var = 3
print(var)

var += 1
print(var)
```

6. Можно комбинировать текст и переменные, используя оператор + и функцию `print()` для вывода строк и переменных, например:

```
var = "007"
print("Agent " + var)
```

Комментарий к комментариям

Комментарии в коде: зачем, как и когда

Скорее всего, вы захотите добавить несколько слов, чтобы объяснить другим людям, которые будут читать ваш код, как работают приемы, используемые в коде, или значения переменных, и, в конце концов, чтобы сохранить информация о том, кто является автором кода и когда программа была написана.

Комментарий — это заметка, вставленная в программу, которая игнорируется во время выполнения.

Как оставить такой комментарий в исходном коде? Это нужно сделать так, чтобы Python не интерпретировал его как часть кода. Когда Python встречает комментарий в программе, он полностью невидим для него — с точки зрения Python, это всего лишь один пробел (независимо от того, насколько длинным будет реальный комментарий).

В Python комментарий — это фрагмент текста, который начинается с `#` (знак решетки) и заканчивается в конце строки.

Если вам нужен комментарий, который занимает несколько строк, то каждую закомментированную строку нужно начинать со знака решетки.

Как здесь:

```
# Эта программа считает гипотенузу с.  
# а и b – длина сторон.
```

```
a = 3.0 b = 4.0 c = (a ** 2 + b ** 2) ** 0.5
# Мы используем ** вместо квадратного корня.
print("c =", c)
```

Хорошие, ответственные разработчики описывают каждый важный кусок кода, например, чтобы объяснить роль переменных; хотя надо сказать, что лучший способ комментировать переменные — назвать их абсолютно однозначно и понятно.

Например, если определенная переменная предназначена для хранения области некоторого уникального квадрата, то имя «`squareArea`» очевидно будет лучше, чем `«auntJane»`.

Обычно говорят, что имя самокомментируется.

Комментарии могут быть полезны в другом отношении — вы можете использовать их, чтобы пометить фрагмент кода, который в данный момент не нужен по какой-то причине. Посмотрите на пример ниже, если вы раскомментируете выделенную строку, это повлияет на вывод кода:

```
# Это тестовая программа.
x = 1
y = 2

# y = y + x
print (x + y)
```

Этот прием часто используют во время тестирования программы, чтобы изолировать то место, где может крываться ошибка.

СОВЕТ

*Если вы хотите быстро закомментировать или раскомментировать несколько строк кода, выделите строки, которые вы хотите изменить, и используйте следующую комбинацию клавиш: **CTRL + /** (Windows) или **CMD + /** (Mac OS). Это очень полезный прием.*

Ключевые выводы

1. Комментарии могут давать дополнительную информацию в коде. Они игнорируются во время выполнения. Информация, оставленная в исходном коде, адресована читателям. В Python комментарий — это фрагмент текста, который начинается с **#**. Комментарий заканчивается в конце строки.
2. Если вам нужен комментарий, который занимает несколько строк, то каждую закомментированную строку нужно начинать со знака решетки (**#**). Кроме того, при помощи комментария можно отметить фрагмент кода, который в данный момент не нужен (см. последнюю строку в фрагменте кода ниже), например:

```
# Эта программа выводит
# на экран приветствие.
print("Hello!") # Вызывает функцию print()
# print("I'm Python.")
```

3. Когда это возможно и оправдано, давайте самокомментирующие имена переменным, например, если вы используете две переменные для хранения дли-

ны и ширины чего-либо, имена переменных `length` и `width` лучше, чем `MyVar1` и `myvar2`.

4. Важно использовать комментарии, чтобы облегчить понимание программ, а также использовать читаемые и осмыслиенные имена переменных в коде. Однако не менее важно не использовать имена переменных, которые сбивают с толку, не оставлять комментарии, содержащие неверную или искаженную информацию!
5. Комментарии могут быть полезны и вам, когда вы спустя какое-то время читаете собственный код (поверьте, разработчики забывают, что делает их собственный код), и когда другие люди читают ваш код (они помогают им понять, что делают ваши программы и как это можно сделать быстрее).

Как общаться с компьютером

Функция ввода `input()`

А сейчас мы познакомимся с совершенно новой функцией, которая на первый взгляд кажется зеркальным отражением старой доброй функции `print()`. Почему? Ну, потому что `print()` отправляет данные в консоль.

А новая функция получает данные из нее. У `print()` нет полезного результата. Смысл новой функции заключается в том, чтобы вернуть полезный результат.

Функция называется `input()` (*ввод*). Название функции говорит само за себя. Функция `input()` считывает введенные пользователем данные и возвращает те же данные в работающую программу.

Программа может манипулировать данными, добавляя интерактивности в код.

Практически любая программа читает и обрабатывает данные. Программа, которая не получает ввод от пользователя, это глухая программа.

Посмотрите на пример ниже:

```
print("Tell me anything...")
anything = input()
print("Hmm...", anything, "... Really?")
```

Он демонстрирует очень простой случай использования функции `input()`.

Попробуйте запустить код и посмотрите, на что способна эта функция.

Примечание:

- Программа предлагает пользователю ввести некоторые данные с консоли (скорее всего, с помощью клавиатуры, хотя это может быть и голосовой ввод или изображение);
- Функция `input()` вызывается без аргументов (это самый простой способ использования функции); функция будет переключать консоль в режим ввода; вы увидите мигающий курсор и сможете ввести несколько символов по нажатию клавиш и закончить, нажав клавишу `Enter`; все введенные данные будут отправлены в вашу программу через результат функции (вам нужно присвоить результат ввода в переменную, иначе это приведет к потере данных);
- затем мы используем функцию `print()` для вывода полученных данных, но с некоторыми оговорками.

Функция `input()` с аргументом

Функция `input()` может еще кое-что — давать подсказки пользователю без использования `print()`. Мы немного изменили наш пример, посмотрите на код:

```
anything = input("Tell me anything...")
print("Hmm...", anything, "...Really?")
```

Примечание:

- Функция `input()` вызывается с одним аргументом — это строка, содержащая сообщение;

- сообщение отобразится в консоли до того, как пользователь сможет что-либо ввести;
- и тогда `input()` просто выполнит свою работу.

Этот вариант вызова функции `input()` упрощает код и делает его более понятным.

Результат функции `input()`

Мы уже об этом говорили, но стоит повторить еще раз: результат функции `input()` — это строка. Страна, которая содержит все символы, введенные пользователем с клавиатуры. Это не целое число и не число с плавающей точкой.

Это значит, что ее не нужно использовать в качестве аргумента для арифметических операций, например, вы не сможете использовать эти данные, чтобы возвести их в квадрат, разделить на что-либо или разделить что-либо на них.

```
anything = input("Enter a number: ")
something = anything ** 2.0
print(anything, "to the power of 2 is", something)
```

Функция `input()` – запрещенные операции

Посмотрите на код ниже. Запустите его, введите любое число и нажмите клавишу `Enter`.

```
#Testing TypeError message
anything = input("Enter a number")
something = anything ** 2.0
print(anything, "to the power of 2 is", something)
```

Что происходит?

У вас должен был получиться такой вывод:

```
Traceback (most recent call last):
File ".main.py", line 4, in <module>
    something = anything ** 2.0
TypeError: unsupported operand type(s) for **
            or pow(): 'str' and 'float'
```

Последняя строка все объясняет — вы попытались применить оператор `**` для `'str'` (строки) с `'float'`. Это запрещено. Но это очевидно — вы сами можете предсказать значение строки **«быть или не быть»**, возвведенной во вторую степень? Не можете. И Python тоже не может. Это тупик? Есть ли решение этой проблемы? Конечно, есть.

Преобразование типов

Python предлагает две простые функции для указания типа данных и решения этой проблемы. Вот они: `int()` и `float()`. Их имена понятны без дополнительных комментариев:

- функция `int()` принимает один аргумент (например, строку `int(string)`) и пытается преобразовать его в целое число; если ей это не удается, вся программа тоже не сработает (для этой ситуации есть обходной путь, но мы разберем его чуть позже);
- функция `float()` принимает один аргумент (например, строку `float(string)`) и пытается преобразовать его в число с плавающей точкой.

Это очень просто и очень эффективно. Более того, можно вызвать любую из функций, передав результаты

`input()` непосредственно этим функциям. Тут нет необходимости использовать переменные в качестве промежуточного хранилища.

Мы реализовали эту идею в редакторе, посмотрите на код.

```
anything = float(input("Enter a number: "))
something = anything ** 2.0
print(anything, "to the power of 2 is", something)
```

У вас есть предположения о том, как строка, введенная пользователем, переходит из `input()` в `print()`?

Попробуйте запустить измененный код. Не забудьте ввести корректное число.

Проверьте несколько разных значений, маленьких и больших, отрицательных и положительных. Ноль — тоже неплохой вариант ввода.

Больше об `input()` и преобразовании типов

Трио из функций `input()-int()-float()` открывает много новых возможностей.

В итоге вы сможете писать законченные программы, принимать данные в виде чисел, обрабатывать их и выводить результаты.

Конечно, эти программы будут очень примитивными и не очень удобными, поскольку они не смогут принимать решения а, следовательно, будут не способны по-разному реагировать на разные ситуации.

Хотя на самом деле, это не проблема; мы скоро разберем способы преодоления таких ситуаций.

В нашем следующем примере мы используем ранее написанную программу, которая находит длину гипотенузы. Давайте перепишем код и сделаем так, чтобы он мог считывать длины сторон с консоли.

Посмотрите на код, приведенный ниже. Вот так он выглядит сейчас.

```
leg_a = float(input("Input first leg length: "))
leg_b = float(input("Input second leg length: "))
hypo = (leg_a**2 + leg_b**2) ** .5
print("Hypotenuse length is", hypo)
```

Программа дважды просит пользователя ввести длину для обеих сторон, считает гипотенузу и выводит результат. Запустите код и введите отрицательные значения.

К сожалению, программа не реагирует на очевидные ошибки. Пока что мы будем игнорировать этот недостаток, но скоро вернемся к нему.

Обратите внимание на то, что переменная `hypo` используется в этом коде только с одной целью — сохранить найденное значение между выполнениями соседних строк кода.

Вы сможете удалить эту переменную из кода после того, как функция `print()` примет выражение в качестве аргумента.

Бот так:

```
leg_a = float(input("Input first leg length: "))
leg_b = float(input("Input second leg length: "))
print("Hypotenuse length is",
      (leg_a**2 + leg_b**2) ** .5)
```

Строковые операторы – введение

Настало время вернуться к этим арифметическим операторам: `+` и `*`.

Наша цель — показать вам, как еще их можно использовать, ведь они выполняют не только функции сложения и умножения.

Мы знаем, как они работают с аргументами в виде чисел (целых или с плавающей точкой).

А теперь мы рассмотрим ситуации, когда они обрабатывают строки, хотя этот процесс весьма специфичен.

Конкатенация (concatenation)

Если применить знак `+` (плюс) к двум строкам, он превращается в оператор конкатенации:

строка + строка

Он просто конкatenирует (объединяет, склеивает) две строки в одну. Естественно, этот оператор можно использовать более одного раза в одном выражении и в этом случае он использует левостороннее связывание.

В программировании, в отличие от математики, оператор конкатенации не является перестановочным, т.е. выражение `"ab" + "ba"` это не то же самое, что `"ba" + "ab"`.

Не забывайте, что если знак `+` вам нужен для конкатенации, а не для сложения, то оба аргумента должны быть строками.

Здесь нельзя смешивать разные типы данных.

Эта простая программа демонстрирует возможности знака `+` во втором варианте его использования:

```

fnam = input("May I have your first name, please? ")
lnam = input("May I have your last name, please? ")
print("Thank you.")
print("\nYour name is " + fnam + " " + lnam + ".")

```

Примечание: использование `+` для объединения строк позволяет вам более точно выстраивать выходные данные, чем с помощью чистой функции `print()`, даже если у нее есть ключевые аргументы `end=` и `sep=`.

Запустите код и посмотрите, соответствует ли вывод вашим прогнозам.

Повторение строки (replication)

Если знак `*` (звездочка) применить к строке и числу, то он станет оператором повторения строки:

```

строка * число
число * строку

```

Он повторяет строку столько раз, сколько указано в числе.

Например:

- результатом этого кода: «James» `* 3` будет «James-James-James»;
- `3 * "an"` будет «ananan»;
- `5 * "2"` (or `"2" * 5`) будет «22222» (не `10!`).

ЗАПОМНИТЕ! Если число меньше или равно нулю, результатом вывода будет пустая строка.

Эта простая программа «рисует» прямоугольник, используя оператор (+) в новой роли:

```
print("+" + 10 * "-" + "+")
print(("|" + " " * 10 + "|\\n") * 5, end="")
print("+" + 10 * "-" + "+")
```

Обратите внимание, как мы использовали круглые скобки во второй строке кода. Потренируйтесь создавать другие фигуры или даже произведения искусства!

Преобразование типов: str()

Вы уже знаете, как использовать функции `int()` и `float()` для преобразования строки в число. Этот тип преобразования работает в обе стороны. Можно преобразовать число в строку, а это намного проще и безопаснее, чем в обратную сторону. Эта функция называется `str()`:

```
str(число)
```

На самом деле, это не предел ее возможностей, но мы вернемся к этому позже.

Снова возвращаемся к прямоугольному треугольнику

Еще раз повторим код нашей программы, которая считает гипотенузу:

```
leg_a = float(input("Input first leg length: "))
leg_b = float(input("Input second leg length: "))
print("Hypotenuse length is " +
      str((leg_a**2 + leg_b**2) ** .5))
```

Мы немного изменили его, чтобы показать вам, как работает функция `str()`. Благодаря ей мы можем передать весь результат функции `print()` как одну строку и забыть про запятые.

Вы добились серьезных успехов на пути к программированию на Python.

Вы уже знаете основные типы данных и набор фундаментальных операторов, как организовать вывод и как получить данные от пользователя. Это очень прочная основа для следующих модулей.

Ключевые выводы

- Функция `print()` отправляет данные на консоль, а функция `input()` получает данные из консоли.
- У функции `input()` есть необязательный параметр: строка подсказки (`prompt`). Она позволяет вам вывести сообщение до пользовательского ввода, например:

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")
```

- Когда происходит вызов функции `input()`, поток программы останавливается, курсор мигает (побуждая пользователя что-то сделать, когда консоль переключается в режим ввода) до тех пор, пока пользователь не введет какие-то данные и/или не нажмет клавишу `Enter`.

Примечание: вы можете на своем компьютере проверить, как работает функция `input()` во всех

возможных вариантах. Откройте IDLE, скопируйте и вставьте приведенный выше фрагмент кода, запустите программу и ничего не делайте — просто подождите несколько секунд, чтобы увидеть, что произойдет.

Совет: вышеупомянутую особенность функции `input()` можно использовать, чтобы предложить пользователю завершить программу. Посмотрите на код ниже:

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")
print("\nPress Enter to end the program.")
input()
print("THE END.")
```

4. Результатом выполнения функции `input()` будет строка. Можно соединять строки друг с другом, используя оператор конкатенации (`+`). Посмотрите на этот код:

```
num1 = input("Enter the first number: ") # Введите 12
num2 = input("Enter the second number: ") # Введите 21
print(num1 + num2) # программа вернет 1221
```

5. А еще строки можно умножать (`*` — повторение строки), например:

```
myInput = ("Enter something: ") # Пример ввода: hello
print(myInput * 3) # Ожидаемый вывод: hellohellohello
```