Глава 2 Основы языка Python

§ 2.1 Зачем изучать Python?

Современному программисту и так приходится иметь дело с большим количеством языков программирования. Поэтому для изучения очередного языка нужны какие-то веские причины.

Питон является одним из самых популярных и востребованных языков программирования. Об этом говорит целый ***ряд рейтингов за 2020 год***.

1. Индекс TIOBE (https://www.tiobe.com/tiobe-index/) присвоил Питону звание языка программирования года.
2. На Stack Overflow (https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-wanted) Питон возглавляет список языков, к изучению которых разработчики проявляют интерес.
3. В рейтинге популярных языков программирования журнала IEEE Spectrum (https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020) Питон находится на первом месте.
4. В исследовании, проведенном компанией JetBrains (https://www.jetbrains.com/ru-ru/lp/devecosystem-2020/), отмечается, что Питон "обошел Java в списке используемых языков. Кроме того, это самый изучаемый язык".
5. Среди используемых языков программирования в проектах Github (https://madnight.github.io/githut/#/pull_requests/2020/4) Питон занимает второе место после JavaScript.

Можно выделить следующие ***причины популярности Питона***:

- *Огромное количество библиотек* для самых разных областей применения.
- Возможность *быстрого прототипирования приложений*, благодаря наличию в языке динамической типизации, автоматического управления памятью ("сборка мусора"), а также благодаря свободной поддержке различных стилей программирования.
- Легкость обучения основам языка, *высокая читаемость кода*. Синтаксис Питона называют "исполняемым псевдокодом".

Питон широко применяется в проектах, связанных с **машинным обучением** и **обработкой естественного языка**. На сайте (https://www.tensorflow.org/about/case-studies) популярной библиотеки для глубокого обучения `TensorFlow` приведены примеры использования Питона крупнейшими мировыми компаниями. Этот язык применяется для задач машинного обучения в компаниях Яндекс (https://yandex.ru/jobs/vacancies/dev/?tags=python) и Google.

Питон популярен в области **анализа и визуализации данных**, где используются такие известные библиотеки, как `Pandas` и `Matplotlib`. Также популярны библиотеки `NumPy` и `SciPy`.

Еще одной естественной областью применения рассматриваемого языка является **разработка веб-приложений**, а также **извлечение данных из веб-страниц** (web scrapers, web crawlers). Фреймворк `Django` пользуется известностью среди веб-разработчиков. С его помощью разработаны (https://www.djangoproject.com/start/overview/) такие сайты, как, например, *Instagram* и *Pinterest*.

Питон пользуется популярностью в задачах **системного администрирования и DevOps**, а также для **автоматизации тестирования**. На Питоне разработана система управления конфигурациями Ansible (https://en.wikipedia.org/wiki/Ansible_(software)) и реализованы автоматические тесты (https://github.com/ghc/ghc/commit/e5063a042c9a1701ea7273da7bacb530d5c077d3) для языка Haskell.

В разработке **компьютерных игр** Питон также используется достаточно широко. Среди известных проектов с существенной долей кода на языке Питон можно вспомнить, например, EVE Online (https://www.eveonline.com/ru/article/stackless-python-2.7) и World of Tanks (https://habr.com/ru/company/wargaming/blog/272265/).

Как в играх, так и в более серьезных приложениях Питон часто играет роль встроенного языка для расширения функциональности программы. Питон **встроен в популярные графические редакторы** Gimp, Inkscape и Blender, а также в такое **музыкальное ПО**, как Ableton Live и Reaper.

Наконец, язык Питон сегодня является одним из самых популярных языков для обучения программированию. В этой роли он используется как в школах, так и в университетах.

§ 2.2 Что собой представляет Python

Среди русскоязычных разработчиков, равно как и в этом тексте, одинаково употребляется и "Питон", и "Python". При этом название языка никак не связано со змеями. Здесь лучше обратиться к истории создания Питона.

Автором языка является голландский программист *Гвидо ван Россум*. Работа над Питоном началась еще в далеком 1989 году, в стенах исследовательского института CWI в Амстердаме. Побудительным мотивом к созданию Питона было желание иметь высокоуровневый язык программирования, занимающий промежуточное положение между языком Си и языком оболочки операционной системы (shell).

Автор языка Питон

Название "Питон" появилось совершенно случайно. Оно происходит от названия английской комедийной передачи "Летающий цирк Монти Пайтона", популярной в том числе и среди программистов того времени. Сегодня программистов, использующих Питон, называют питонистами.

Для тех, кто уже имеет опыт программирования и начинает осваивать Питон сразу запоминаются следующие *особенности языка*:

1. ***Использовании отступов для выделения блоков***. Для программиста, который привык выделять блоки ключевыми словами `begin ... end` или фигурными скобками `{ ... }` такое решение может показаться непривычным, но в результате хорошо написанные программы на Питоне выглядят, как псевдокод из учебников.
2. ***Отсутствие объявления переменных и не указываются типы данных***. Питон является *динамически типизированным языком*. Это означает, что при задании какой-либо переменной, нам не надо объявлять ее тип (число, строка, и т.д.), как это сделано в языке С. То есть достаточно

просто присвоить ей значение и в зависимости от того, какое это значение, Python сам определит тип переменной. Кроме того проверка типов, в отличие от, к примеру, Java или C++, происходит в процессе выполнения программы, а не на стадии ее компиляции. В процессе выполнения программ на Питоне в большинстве случаев не допускается неопределенного состояния и при возникновении ошибки выполнение программы прекращается с выдачей информативного сообщения.

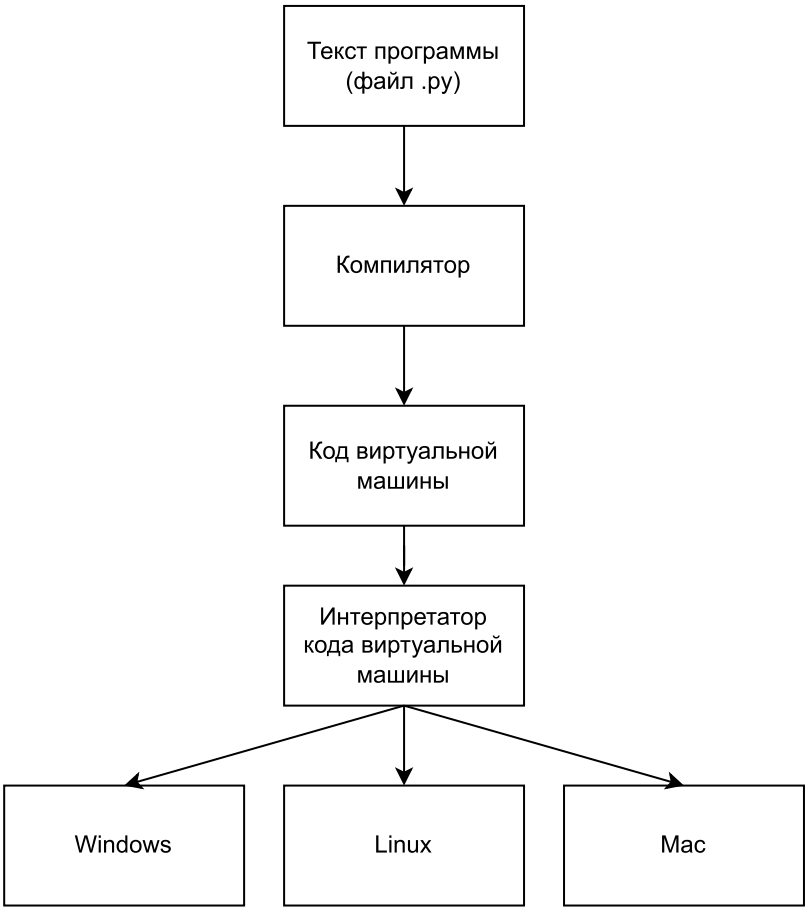
Питон также иногда называют динамическим языком. То есть таким языком, в котором программы во время выполнения имеют доступ к среде языка, включая ее внутренние структуры и компилятор.

В Питоне ***имеется командная строка для диалога с программистом***, так называемый *REPL* — интерактивный цикл "чтение, выполнение, вывод" (*read-eval-print loop*). REPL не только упрощает отладку программ, но и обеспечивает очень популярный в Питоне стиль исследовательского программирования. Этот стиль программирования удобен, когда программы развиваются эволюционно и на первых порах еще не ясны в деталях предметная область, необходимые алгоритмы и структуры данных.

В Питоне, аналогично языкам C# и Java, используется автоматическое управление памятью, то есть реализована ***сборка мусора***.

Питон относится к мультипарадигменным языкам. В нем поддерживаются элементы процедурного, объектно-ориентированного и функционального программирования. Кроме того, имеются возможности метапрограммирования.

Иногда возникает путаница в вопросе, является ли Питон языком компилируемым или интерпретируемым. Основная реализация Питона, CPython, написанная на Си, включает в себя и компилятор, и интерпретатор. Первым делом программа на Питоне компилируется в байткод, то есть в представление для виртуальной машины, а затем полученное представление выполняется с помощью интерпретатора байткода.



Процесс компиляции и интерпретации программы на Питоне

Основные принципы, согласно которым развивается Питон, сформулированы в так называемом **"Дзене Питона"**. Его текст на английском языке можно получить с помощью следующей команды интерпретатора CPython.

В [2]:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Русский перевод:

Дзен Питона, Тим Питерс

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Сложное лучше, чем запутанное.
- Плоское лучше, чем вложенное.
- Разреженное, лучше чем плотное.
- Читаемость имеет значение.
- Особые случаи не настолько особые, чтобы нарушать правила.
- Но практичность важнее безупречности.
- Ошибки никогда не должны замалчиваться.
- Если только они не замалчиваются явно.
- Перед лицом двусмысленности откажитесь от соблазна угадывать.
- Должен существовать один и, желательно, только один очевидный способ сделать это.
- Хотя он поначалу может быть и не очевиден, если вы не голландец. `Намек на автора языка`
- Лучше сейчас, чем никогда.
- Хотя никогда зачастую лучше, чем прямо сейчас.
- Если реализацию трудно объяснить, то идея плоха.
- Если реализацию легко объяснить, идея, возможно, хороша.
- Пространства имен — отличная идея! Давайте делать их больше!

Эти пункты действительно напоминают положения какого-то религиозно-философского учения и могут допускать различные толкования.

В более строгой форме развитие языка происходит в соответствии с документами PEP (https://www.python.org/dev/peps/) (Python Enhancement Proposal — предложения по развитию Питона).

§ 2.3 Недостатки языка

Питон, как и все прочие языки программирования, имеет свои границы применимости и свои слабые стороны.

Ключевым недостатком Питона можно считать ***практически полное отсутствие проверок корректности программы до этапа ее выполнения***. Компилятор Питона своевременно оповестит программиста о синтаксических ошибках, но если вы передали в функцию, к примеру, список вместо числа, то в этом случае узнать об ошибке можно будет только во время выполнения соответствующей функции.

В Питоне, как и в других языках с динамической типизацией, указанный недостаток до некоторой степени преодолевается с помощью написания автоматических тестов и использования сторонних инструментов статического анализа. Кроме того, в новых версиях Питона введены аннотации типов, наличие которых позволяет производить необязательную статическую проверку типов (*gradual typing*) с использованием инструмента `mypy`.

Традиционно многие разработчики на Питоне жалуются на ***недостаточную производительность*** своих программ. Действительно, согласно этим (https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-gcc.html) данным, на многих задачах Питон отстает от Си более чем в 100 раз. Такой результат является платой за динамическую природу Питона. Тем не менее, существуют альтернативные реализации Питона, такие, например, как *PyPy* и *Numba*, значительно улучшающие производительность программ. Нужно, однако, добавить, что эти альтернативные реализации лишь до некоторой степени совместимы с *CPython* — эталонной реализацией Питона.

Еще одним важным недостатком Питона является ***недостаточная совместимость между различными версиями языка***. В какой-то момент существовало две основных версии Питона — вторая и третья. Сегодня большинство разработчиков, все-таки, перешли на версию 3 (но даже в рамках этой мажорной версии существовали минорные версии языка, несовместимые друг с другом). Проблемы несовместимости все еще могут давать о себе знать в тех случаях, когда вы сталкиваетесь со старым кодом на Питоне из учебников или из сети.

Следует также добавить, что многих питонистов беспокоит ***регулярное добавление в язык новых конструкций***, что приводит к все большему усложнению изначально простого языка.

§ 2.4 Комментирование кода

Комментарий пишется с использованием специального символа `#` (решетка), после которого идет текст комментария. Также комментарий можно писать на строках с выражениями. Это так называемые ***inline*** комментарии.

Многострочный комментарий **не игнорируется** интерпретатором Python. Это строковый литерал, который зачастую вы будете видеть, как часть документации функции, либо класса. Этот строковый литерал становится частью объекта, к которому можно достучаться, используя специальное свойство `__doc__` у объекта.

> **Строки документирования** – это литералы строк, которые присутствуют на верхнем уровне различных структур и автоматически сохраняются интерпретатором в атрибутах `__doc__` соответствующих им объектов. Строки документирования могут присутствовать в модулях, в инструкциях `def` , а также в определениях классов и методов.

Строки документирования доступны во время выполнения, но синтаксически они менее гибки, чем комментарии `#` (которые могут находиться в любом месте программы). Обе формы – полезные инструменты, и любая документация к программе – это хорошо (при условии, что она точная). Вообще говоря, строки документирования лучше использовать для функционального описания (что делают объекты), а комментарии `#` – для небольших пояснений (описывающих, как действуют выражения).

In [6]:

```
'''
Это строки документирования самой программы
'''

def mysqrt(arg):
    '''
    Функция извлекает квадратный корень из x
    TODO:
    '''
    return arg**0.5  # это возвращаемое значение

print(__doc__)
print(mysqrt.__doc__)
print(print.__doc__)
```

```
Функция извлекает квадратный корень из x
TODO:
```

> **TODO** (to do от англ. *try to do sth* — попробовать сделать что-то) — распространённый тип пометки в комментариях исходных текстов программ, документации и т. д., показывающий разработчику место, где следует продолжить работу (исправить ошибку или неточность, добавить функциональность, учесть какой-то специфичный случай и т.д.

Документацию по выбранной функции **можно получить** с помощью `help()` :

In [8]:

```
help(max)
```

```
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

In [9]:

```
help(mysqrt)
```

```
Help on function mysqrt in module __main__:

mysqrt(arg)
    Функция извлекает квадратный корень из x
    TODO:
```

§ 2.5 Понятие типа данных и переменных в Python

Стоит отметить, что в Python **регистр букв важен**, поэтому, к примеру, переменные `n` и `N` различны. Говоря более строго, имена (или, иначе, идентификаторы) могут содержать буквы в верхнем или нижнем регистре, символ `_`, а также цифры (за исключением первого символа имени).

Важно отметить, что если переменная состоит из нескольких слов, то есть это длинное название переменной, то ее принято называть так называемым *snake_case'ом* (змеиный язык). То есть слова начинаются с маленькой буквы, и отдельные слова разделены символом нижнего подчеркивания. В других языках вы могли видеть, что переменные называются в так называемом *camelCase'ом*. В Python'е так делать не принято (PEP8).

```
snake_case = 'Так нормуль'
camelCase = 'Так не принято'
```

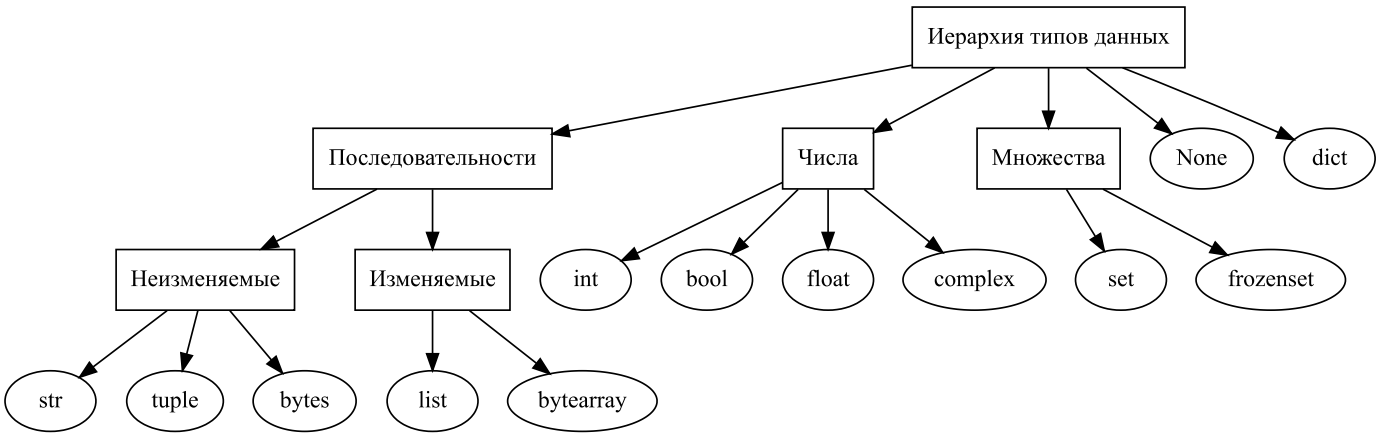
Каждое значение в Python имеет тип. Поскольку **всё в Python является объектом**, то типы являются классами, а значения – экземплярами (объектами) этих классов. *Все типы в Python являются объектами*. При создании объекта вызывается специальная функция – *конструктор*.

Тип данных (англ. *Data type*) – характеристика, определяющая множество значений и набор операций на этих значениях:

- ***множество допустимых значений***, которые могут принимать данные, принадлежащие к этому типу (например, объект типа *Целое число* может принимать только целочисленные значение в определенном диапазоне);
- ***набор операций***, которые можно осуществлять над данными, принадлежащими к этому типу (например, объекты типа *Целое число* умеют складываться, умножаться и т.д.).

Существует несколько видов типов данных - *встроенные* и *не встроенные*. Встроенные - те типы, которые встроены в интерпретатор, не встроенные - типы данных, которые можно импортировать из других модулей.

На рисунке ниже показаны основные **встроенные типы данных** языка Python.



Встроенные типы данных в Python

1. **None type** - тип, представляющий отсутствие значения. `None` - неопределенное значение переменной. Зачастую, чтобы отловить какую-то ошибку записи значения куда-либо, мы применяем проверку на отсутствие значения в переменной, ячейке базы данных, таблицы и т.д. или используем в качестве значения по умолчанию.

2. **Логический тип данных** (`bool`) удобно использовать, когда результатом условия может быть только "да" или "нет". В математическом представлении `True = 1` , `False = 0` . `True` - логическая переменная, истина; `False` - логическая переменная, ложь.

3. **Целые числа** используются для стандартных арифметических операций, когда нас не интересует точность, до *n*-го знака. `int` - целое число.

4. Напротив, числа, применяющиеся для точных вычислений до n — го знака после запятой - **числа с плавающей точкой**. При арифметическом взаимодействии двух типов (`int` и `float`), результат всегда будет иметь тип `float` . `float` - число с плавающей точкой.

5. **Комплексные числа** предназначены для более сложных математических вычислений, они состоят из вещественной и мнимой части. `complex` - комплексное число.

6. ***Строки*** используются для формирования сообщений, каких-либо сочетаний символов, текстовой информации. `str` - строка.

7. ***Списки*** являются своего рода хранилищем данных разного типа, другими словами списки это массивы, только хранить они могут данные разных типов. `list` - список.

8. ***Кортеж*** - это список, который после создания нельзя изменить, очень полезно его использовать для защиты "от дурака", чтобы по ошибке данные не были изменены. `tuple` - кортеж.

9. ***Множества*** - коллекции для неповторяющихся данных, хранящие эти данные в случайном порядке. `set` - множество `frozenset` - неизменяемое множество.

10. ***Словари*** - являются набором пар "ключ"-"значение", довольно удобный тип данных для формирований структур. `dict` - словарь

Следующие два типа перечислены для ознакомления, мы не будем уделять им практического значения.

11. ***Байт*** - это минимальная единица хранения и обработки цифровой информации. Данный тип допускает возможность производить изменение кодировки символов в зависимости от задач. `bytes` - байтовая последовательность.

12. ***Последовательность байт*** - представляет собой некую информацию (текст, картинка и т.д.). Помимо изменения кодировки, имеет дополнительные возможности применять методы к перекодированным строкам и вносить изменения. `bytearray` - байтовый массив.

Важно отметить, что каждое значение в Python является объектом-"ящиком", в котором хранится дополнительная информация, включая тип этого объекта. Узнать тип объекта можно с помощью встроенной функции `type` .

Все объекты независимо от типа поддерживают ряд общих функций.

Таблица. Общие функции для объектов

Оператор	Описание функции	Пример использования	Результат
`id(object)`	Возвращает уникальный идентификатор `object` (в CPython – адрес в памяти)	`age = 42; print(id(age))`	`140709756349520`
`x is y`	Возвращает `True` , если `x` и `y` указывают на один и тот же объект	`a = b = 5; a is b`	`True`
`type(object)`	Возвращает тип `object`	`ls = [1, 2, 4]; type(ls)`	`list`
`isinstance(object, class-or-type-ortuple)`	Возвращает `True` , если `object` является классом или подклассом `class-or-type-or-tuple`	`a = 5; isinstance(a, (int, float))`	`True`
`help([object])`	Отображает справку для `object`		

B [10]:

```
type(None)
```

Out[10]:

```
NoneType
```

B [11]:

```
type(True)
```

Out[11]:

```
bool
```

B [12]:

```
type(False)
```

Out[12]:

```
bool
```

B [13]:

```
type(1)
```

Out[13]:

```
int
```

B [14]:

```
type(5.3)
```

Out[14]:

```
float
```

B [15]:

```
type(5 + 4j)
```

Out[15]:

```
complex
```

B [16]:

```
type([1, 5.3, False, 4])
```

Out[16]:

```
list
```

B [17]:

```
type((1, True, 3, 5+4j))
```

Out[17]:

```
tuple
```

B [18]:

```
type(range(5))
```

Out[18]:

```
range
```

B [19]:

```
type('Hello')
```

Out[19]:

```
str
```

B [20]:

```
type(b'a')
```

Out[20]:

```
bytes
```

B [21]:

```
type(bytearray([1,2,3]))
```

Out[21]:

```
bytearray
```

B [22]:

```
type(memoryview(bytearray('XYZ', 'utf-8')))
```

Out[22]:

```
memoryview
```

B [23]:

```
type({'a', 3, True})
```

Out[23]:

```
set
```

B [25]:

```
type(frozenset({1, 2, 3}))
```

Out[25]:

```
frozenset
```

B [26]:

```
type({'a': 32, 'b': 45})
```

Out[26]:

```
dict
```

Литералы (нотации или синтаксические представления значений) целых могут быть **представлены в одной из четырех систем счисления**:

```
>>> 12345 # Основание 10
12345
>>> 0xDeadBeef # Основание 16
3735928559
>>> 0o757 # Основание 8
495
>>> 0b1100 # Основание 2
12
```

Числа с плавающей точкой записываются в привычной нотации с точкой, а также в экспоненциальной ("научной") нотации, то есть в формате $m \times 10^n$.

```
>>> 1234.5
1234.5
>>> 12345e-1
1234.5
```

Для **улучшения читаемости** в числовых литералах можно использовать символ `_` :

```
>>> 1_000_000
1000000
>>> 3.14_159
3.14159
```

Целые значения могут иметь произвольный размер (точнее говоря, насколько хватит ОЗУ). Булевы значения из множества `{True, False}` являются подмножеством целых. **Значения с плавающей точкой** подчиняются стандарту IEEE-754 для 64-битных значений, который позволяет обеспечивать представление примерно 17 значимых разрядов, с экспонентой в диапазоне от -308 до 308. Это полностью соответствует типу `double` в языке C++.
Также вещественные числа не поддерживают длинную арифметику:

```
>>> a = 3 ** 1000
>>> a + 1.5
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-23-05a22d8395f3> in <module>
      1 a = 3 ** 1000
----> 2 a + 1.5

OverflowError: int too large to convert to float
```

Следует помнить про следующие **особенности работы** с большими вещественными числами:

In [5]:

```
a = 1.3 * 10**20
print(a == a + 1000)
print(a)
print(a+1)
```

```
True
1.3e+20
1.3e+20
```

Python умеет работать с **комплексными числами** (тип `complex`), для того, чтобы определить комплексное число нужно для мнимой части использовать символ `J`, посмотрите на пример, мы определили комплексное число $6.5 + 2.3J$ и можем убедиться, что его тип действительно `complex`, используя встроенную функцию `type`.

Чтобы достучаться до реальной и мнимой части комплексного числа, мы можем воспользоваться атрибутами `real` и `imag`. С помощью инструкции `complex(x, y)` можно произвести преобразование к комплексному типу $x + i \cdot y$ (по умолчанию $y = 0$). Также для работы с комплексными числами используется также модуль `cmath`.

B [31]:

```
num = 6.5 + 2.3J

print(type(num))
print(num.real)
print(num.imag)
```

```
<class 'complex'>
6.5
2.3
```

В процессе работы программы мы можем из одного типа **конвертировать** переменные в другой тип. При конвертации чисел из действительного в целый тип будет потеряна дробная часть вещественного числа. Для преобразования чисел есть встроенные функции `int` и `float`:

```
>>> float(4)
4.0
>>> int(2.7)
2
>>> int(True)
1
>>> int(False)
0
```

Все объекты в Python относятся к одной из 2-х категорий:

1. **Мутирующие** или **изменяемые** (англ. *Mutable*): содержимое объекта можно изменить после создания - списки (`list`), множества (`set`), словари (`dict`);
2. **Немутирующие** или **неизменяемые** (англ. *Immutable*): содержимое объекта нельзя изменить после создания - числовые данные (`int` , `float` , `complex`), символьные строки (class `str`), кортежи (`tuple`).

Давайте посмотрим на пример использования **изменяемого объекта**:

In [14]:

```
age = 42
print(id(age))
print(type(age))
print(age)
age = 43
print(age)
print(id(age))
```

```
140736587181120
<class 'int'>
42
43
140736587181152
```

42 - целое число, типа `int` , который является неизменным. Здесь `age` - это имя, которое указывает на `int` объект, значение которого 42.

Когда мы печатаем `age = 43` создается другой объект, типа `int` со значением 43 и `age` уже указывает на другое место (`id` будет отличаться). Таким образом, мы не изменили 42 на 43.

Как вы можете видеть из печати, что `id(age)` до и после создания второго объекта с именем `age` - разные.

Теперь давайте посмотрим на тот же пример с использованием **изменяемого объекта**.

In [16]:

```
ls = [1, 2, 3]
print(ls)
print(id(ls))
ls.pop() # возвращает последний элемент, удаляя его из последовательности
print(ls)
print(id(ls))
```

```
[1, 2, 3]
3066276540672
[1, 2]
3066276540672
```

В этом примере мы создали список `ls` который содержит 3 целых числа $1, 2, 3$. После того, как мы изменим список, «выталкнув» последнее значение 3 идентификатор `ls` остается такой же!

Итак, объекты типа `int` - неизменны, а объекты типа `list` - изменяемые.

Как мутирующие, так и немутирующие объекты имеют свои **преимущества и недостатки**. Основным преимуществом немутирующих типов является гарантия неизменяемости с момента создания: каждый использующий участок кода имеет дело с копией объекта и не может его каким-либо образом изменить. Этот же принцип формирует основной недостаток немутирующих типов: большее количество потребляемой памяти на дополнительное копирование объектов при необходимости внесения изменений.

> **Переменная** (англ. *Variable*) – это идентификатор, который указывает на определенную область памяти, где хранятся произвольные данные – созданный объект (значение переменной).

Присваивание выполняется «справа налево» и подразумевает шаги:

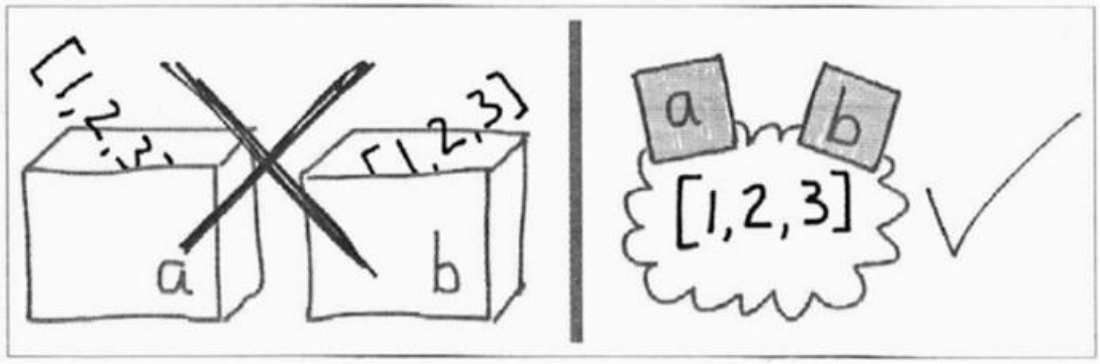
- если справа от оператора находится литерал (например, строка или число), то в операнд слева записывается ссылка, которая указывает на объект в памяти, хранящий значение литерала:

```
a = 100  # Создание объекта 100 и запись ссылки на него в переменную a
```

- если справа находится ссылка на объект, в левый операнд записывается ссылка, указывающая на тот же самый объект, на который ссылается правый операнд;

```
a = 100
b = a  # В переменную b копируется ссылка из a -
  # они будут указывать на один и тот же объект
```

Переменная лишь указывает на данные - хранит ссылку, а не сами данные. В виду того, что копирования данных при этом не происходит, операция присваивания выполняется с высокой скоростью. В связи с этим целесообразнее в Python в качестве метафоры рассматривать переменные как стикеры (этикетки), цепляемые к данным, а не ящики и говорить, что «переменная присвоена объекту», а не привычное «переменной присвоен объект».



B [35]:

```
a = 100
b = a
print(id(a), id(b))
a = 300
print('Значение a = %d, b = %d' % (a, b))
print(id(a), id(b))
```

```
140736445428608 140736445428608
Значение a = 300, b = 100
1456168984848 140736445428608
```

При изменении значения `a`, Python не изменяет объект 100 (оставляя его "как есть", т.к. знает, что он используется другими), а создает новый, меняя ссылку у `a`, при этом прочие объекты продолжают ссылаться на 100.

Говоря о присвоении значения переменной, стоит отметить, что в реальности происходит **процесс связывания ссылки на объект с объектом**, находящемся в памяти посредством оператора `=`. Таким образом в инструкции типа `var = 12`, `var` - ссылка на объект, а "12" - объект целочисленного типа.

Каждый раз, когда в тексте будет упоминаться процесс присвоения значения - помните, что в этот момент происходит процесс связывания ссылок с объектами.

> **Операцию присваивания** можно представить себе как запись по ключу в структуру данных словарь (хеш-таблицу). Первое присваивание создает новую запись в этом словаре, называемом глобальным пространством имен (*globals*). Последующие записи по тому же ключу-имени обновляют соответствующее значение, которое может быть объектом произвольного типа.

Оператор присваивания копирует ссылку на объект, создавая т.н. **поверхностную копию**. В ряде случае необходимо создать полную копию объекта - **глубокую копию**, например, для мутирующих коллекций, чтобы после изменять новую коллекцию без изменения оригинала.

In [8]:

```
x = [53, 68, ["A", "B", "C"]]

x1 = x          # Поверхностная копия (через присваивание)
x2 = x[:]       # Глубокая копия (создается при срезе)
x3 = x.copy()   # Глубокая копия (через метод copy())
print(id(x), id(x1), id(x2), id(x3))

x1 is x, x2 is x, x3 is x
```

```
3066276553728 3066276553728 3066276535104 3066277539648
```

Out[8]:

```
(True, False, False)
```

In [2]:

```
# Еще раз пример копирования для неизменяемых типов
a = 5
b = 5
c = a
id(a), id(b), id(c)
```

Out[2]:

```
(140736587179936, 140736587179936, 140736587179936)
```

Но **с мутирующими типами** (например, списком) Python поступает **по-другому**

In [4]:

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a
id(a), id(b), id(c)
```

Out[4]:

```
(3066276555200, 3066276554816, 3066276555200)
```

При изменении мутирующего типа "изменяются" и указывающие на него объекты - т.к. они хранят ссылку

на тот же объект!

In [6]:

```
a[0] = 5
a, b, c
```

Out[6]:

```
([5, 2, 3], [1, 2, 3], [5, 2, 3])
```

Важно помнить об отличии изменяемых (*mutable*) и неизменяемых (*immutable*) типов. Это поможет понять логику происходящего в следующих двух примерах.

In [15]:

```
x = y = 0
x += 1
print(x, y)
```

```
1 0
```

In [12]:

```
x = y = []
x.append(1)
x.append(2)
print(x, y)
```

```
[1, 2] [1, 2]
```

С помощью встроенной функции `globals` можно узнать текущее состояние глобального пространства имен:

```
>>> globals()
```

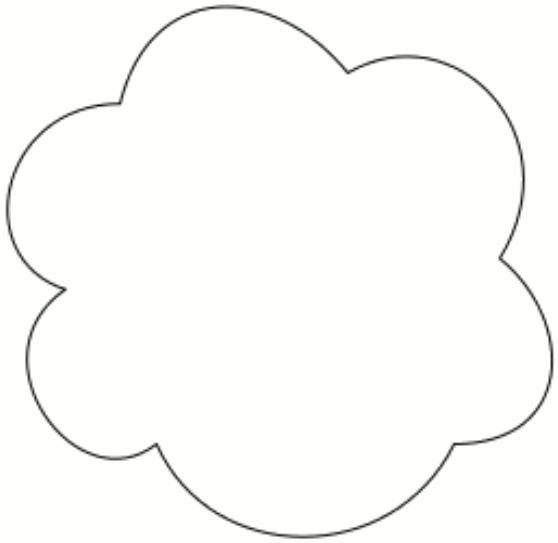
§ 2.6 Управление памятью и сборщик мусора

Создание объекта любого типа подразумевает выделение памяти для размещения данных об этом объекте. Когда объект больше не нужен – его необходимо удалить, очистив занимаемую память. Python – язык с встроенным менеджером управления памятью и выполняет данные операции автоматически за счет наличия ***сборщика мусора*** (англ.Garbage Collection, GC).

> **Сборка мусора** – технология, позволяющая, с одной стороны, упростить программирование, избавив программиста от необходимости вручную удалять объекты, созданные в динамической памяти, с другой – устранить ошибки, вызванные неправильным ручным управлением памятью.

Алгоритм, используемый сборщиком мусора называется подсчетом ссылок (англ. Reference Counting). Python хранит журнал ссылок на каждый объект и автоматически уничтожает объект, как только на него больше нет ссылок.

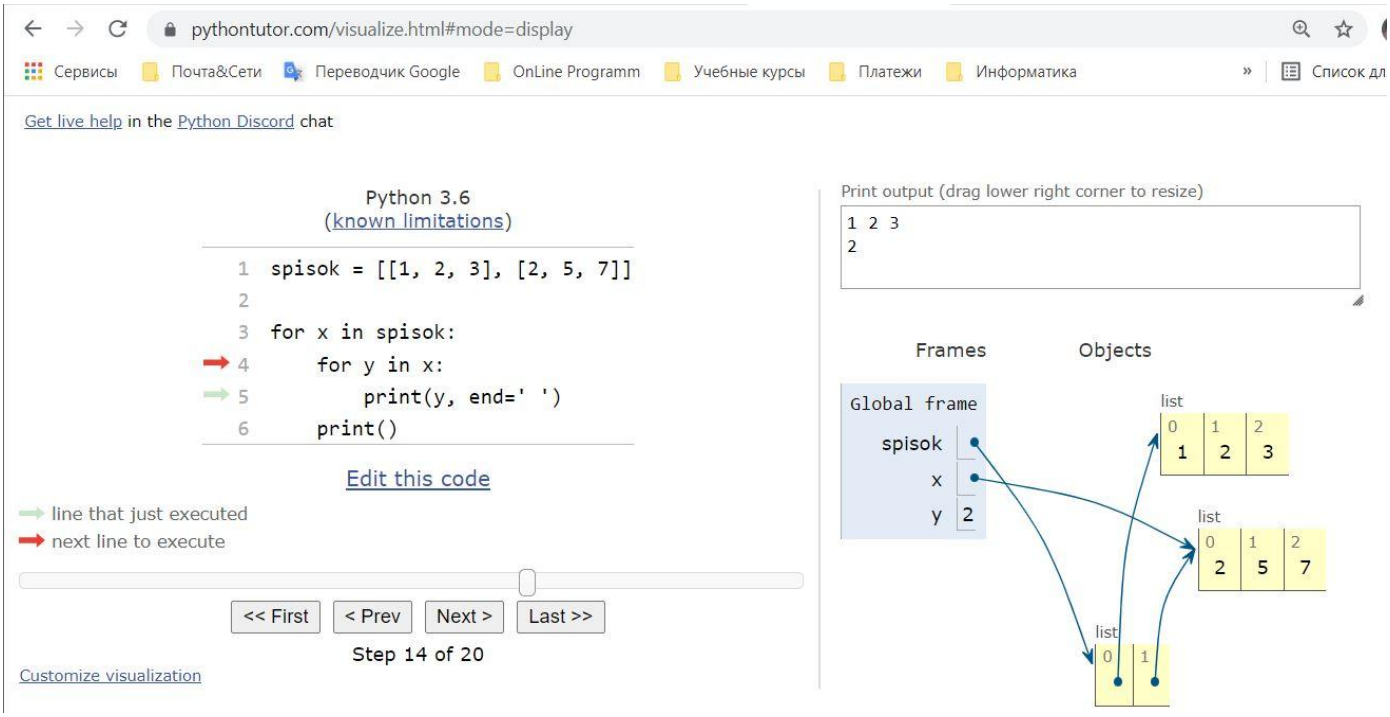
```
pi = 3.14
radius = 10
area = pi * radius**2

radius = 15
```



Время между созданием и уничтожением объекта – его **жизненный цикл**.

> Объекты, которые имеют на протяжении своего жизненного цикла одно неменяющееся и характеризующее их значение, а также умеют сравниваться, называются **хешируемыми**. К хешируемым объектам относятся все немутирующие типы данных, а также пользовательские объекты.

Очень интересно и поучительно посмотреть, что происходит в ячейках памяти при работе программы. Это можно сделать по ссылке: https://pythontutor.com/ (https://pythontutor.com/). Особенно интересно посмотреть, как работают циклы и списки.



In [1]:

```
spisok = [[1, 2, 3], [2, 5, 7]]

for x in spisok:
    for y in x:
        print(y, end=' ')
    print()
```

```
1 2 3
2 5 7
```

§ 2.7 Константы

В Python не существует привычного для, например, Си или Паскаля понятия константы. Вместо этого, значение, которое подразумевается как константа, обозначается заглавными буквами (`MEMORY_MAX = 1024`), визуально предупреждая, что данное значение менять не следует.

§ 2.8 Функция int

В языке программирования Python встроенная функция `int()` возвращает целое число в десятичной системе счисления (класс `int`).

Если вызвать функцию `int()` без аргументов, она вернет 0.

In [15]:

```
x = int()
x
```

Out[15]:

```
0
```

Если передать функции **целое число**, то она **вернет его же**. Если передать **вещественное число**, то **оно будет округлено** до целого в сторону нуля (т. е. дробная часть будет отброшена).

```
>>> int(-103)
-103
>>> int(234.879)
234
>>> int(-0.3)
0
>>> int(-0.9)
0
```

Попытка преобразовать строку, содержащую вещественное число, в целое число с помощью функции `int()` вызывает ошибку.

```
>>> int('-23.1')
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    int('-23.1')
ValueError: invalid literal for int() with base 10: '-23.1'
```

Отсюда следует, что если требуется преобразовать дробное число в строковом представлении в целое число, сначала надо использовать функцию `float()`, а затем `int()`.

```
>>> a = '15.76'
>>> a = float(a)
>>> a = int(a)
>>> a
15
```

В случае, **когда указывается второй аргумент** для функции `int()`, **первый** всегда должен быть **строкой**. С помощью второго аргумента функции `int()` сообщается, в какой **системе счисления** находится число, указанное в строке первого аргумента. Функция `int()` возвращает его значение в десятичной системе счисления.

В [12]:

```
print(int('ABC', 16))
print(int('5274', 8))
print(int('101010111100', 2))
```

```
2748
2748
2748
```

Для двоичной, восьмеричной и шестнадцатеричной систем счисления **можно использовать префиксы**:

- для двоичной системы - `0b`;
- для восьмеричной системы - `0o`;
- для шестнадцатеричной системы - `0x`;

При использовании префиксов аргумент функции в кавычки не заключается и второй аргумент не указывается.

В [14]:

```
print(int(0xABC))
print(int(0o5274))
print(int(0b101010111100))
```

```
2748
2748
2748
```

Также целые числа можно переводить в другие системы счисления используя методы:

- `bin(a)` - перевод числа в двоичную систему счисления
- `hex(a)` - перевод числа в 16-тиричную систему счисления
- `oct(a)` - перевод числа в 8-миричную системы счисления

In [2]:

```
N = 2748

print('2CC:', bin(N))
print('8CC:', oct(N))
print('16CC:', hex(N))
```

```
2CC: 0b101010111100
8CC: 0o5274
16CC: 0xabc
```

§ 2.9 Отступы

Программа на Питоне состоит из некоторого количества строк. Есть случаи, когда в Питоне объединяется несколько "физических" строк (тех, что мы наблюдаем в текстовом редакторе) в одну строку. Вот эти случаи:

- Использование символа `\` в конце строки означает ее соединение с последующей строкой.
- Выражение в круглых, квадратных или фигурных скобках занимает с точки зрения Питона одну строку, даже если элементы выражения располагаются на разных физических строках.

In [7]:

```
s = 'Иногда при выводе длинных строк \
удобно разрывать логически не связанные места \
и переходить на новую строку. Тогда пригодится символ \.'
print(s)
```

```
Иногда при выводе длинных строк удобно разрывать логически не связанные мест
а и переходить на новую строку. Тогда пригодится символ \.
```

In [21]:

```
data = [
    ['100412', 'Ботинки для горных лыж ATOMIC Hawx Prime 100', 9],
    ['100728', 'Скейтборд Jdbug RT03', 32],
    ['100732', 'Роллерсерф Razor RipStik Bright', 11],
    ['100803', 'Ботинки для сноуборда DC Tucknee', 20],
    ['100898', 'Шагомер Omron HJA-306', 2],
    ['100934', 'Пульсометр Beurer PM62', 17],
]

data
```

Out[21]:

```
[['100412', 'Ботинки для горных лыж ATOMIC Hawx Prime 100', 9],
 ['100728', 'Скейтборд Jdbug RT03', 32],
 ['100732', 'Роллерсерф Razor RipStik Bright', 11],
 ['100803', 'Ботинки для сноуборда DC Tucknee', 20],
 ['100898', 'Шагомер Omron HJA-306', 2],
 ['100934', 'Пульсометр Beurer PM62', 17]]
```

В Питоне, как и во многих других языках программирования, используется блочная структура. При этом

блоки отмечаются исключительно позицией отступа в строке. Питонисты придерживаются следующего соглашения: отступ занимает 4 пробела.

Если вы увеличиваете отступ очередной строки, то тем самым переносите эту строку в более внутренний блок. Разумеется, строки, относящиеся к одному блоку должны обладать одинаковым отступом. Чтобы выйти из некоторого количества блоков нужно просто вернуть обратно позицию отступа до позиции желаемого блока.

§ 2.10 Ввод - вывод данных

Обработка данных, как правило, осуществляется после получения данных из какого-либо источника, а после обработки осуществляется их вывод. Ввод и вывод в Python, используя стандартный терминал ОС, осуществляется с помощью функций `input()` и `print()` соответственно.

2.10.1 Функция input()

Встроенная функция `input()` позволяет получить ввод пользователя в виде строки.

```
input('prompt')
```

Печатает строку `prompt` (без переноса строки и если задана) и ожидает ввода пользователя. Ввод подтверждается клавишей Enter, возвращая строку с введенными данными в качестве результата.

Пример:

```
name = input ("Введите ваше имя")
```

После этого нам предлагается ввести имя. После ввода имени и нажатия Enter внутри переменной name содержится как раз то значение, которое мы ввели, и мы можем с ним работать.

2.10.2 Функция print()

Встроенная функция `print()` используется для вывода полученных результатов на консоль.

```
print(value1, ..., [valueN], [sep = ' '], [end = '\n'], [file = sys.stdout], [flus
h = False])
```

Параметры:

- `value1, ..., valueN` — список выводимых значений любого типа (при печати все объекты преобразуются в строки.);
- `sep` — разделитель при выводе нескольких объектов (`value1, ..., valueN`). По умолчанию одиночный пробел;
- `end` — строка, завершающая вывод. По умолчанию — символ переноса строки;
- `file` — выходной поток. По умолчанию — консоль (экран монитора);
- `flush` — флаг, определяющий режим вывода в файл при наличии цикла. Действует только в том случае, если выбран файл в аргументе `file` . При значении `True` вывод в файл происходит незамедлительно при вызове функции `print()` . При значении `False` (по умолчанию) вывод в файл будет произведен только после выхода из цикла.

Пример:

```
name = input("Введите Ваше имя: ")
print(f"Привет, {name}!")
```

Пример:

```
name = input("Введите Ваше имя: ")
print("Привет, ", name, "!", sep="")
```

§ 2.11 Типы операторов

> **Оператором** является элемент выражения, который указывает на то, какое действие необходимо произвести между элементами.

То есть, в выражении `21 - 4` символ `-` является оператором, указывающим на то, что нужно произвести вычитание. `21` и `4` при этом называются **операндами**.

2.11.1 Арифметические операторы

Оператор	Описание	Пример использования	Результат
`+`	Оператор суммы	`print(5 + 8)`	13
`-`	Оператор разности	`print(31 - 2)`	29
`*`	Оператор произведения	`print(12 * 9)`	108
`/`	Оператор деления	`print(6 / 4)`	1.5
`%`	Оператор получения остатка от деления	`print(6 % 4)`	2
`**`	Оператор возведения в степень	`print(9 ** 2)`	81
`//`	Оператор целочисленного деления	`print(6 // 4)`	1
`abs(a)`	Модуль числа	`abs(-4)`	4
`divmod(a,b)`	Получение пары чисел (`a // b` , `a % b`)	`divmod(25, 3)`	(8, 1)
`pow(a, b[, c])`	`a` в степени `b` . Если указано число `c` , тогда вычисляется остаток от деления на число `c`	`pow(4, 3, 5)`	5

```
В [6]:
```

```
pow(4, 3, 5)
```

```
Out[6]:
```

4

2.11.2 Операторы сравнения

Оператор	Описание	Пример использования	Результат
`==`	Проверяет, равны ли операнды между собой. Если они равны, то выражение становится истинным	`print(5 == 5)`	True
`!=`	Проверяет, равны ли операнды между собой. Если они не равны, то выражение становится истинным	`print(12 != 12)`	False
`>`	Проверяет, больше ли левый операнд чем правый, если больше, то выражение становится истинным	`print(432 >500)`	False

Оператор	Описание	Пример использования	Результат
<	Проверяет, меньше ли левый операнд чем правый, если меньше, то выражение становится истинным	print(5 < 51)	True
>=	Проверяет, больше левый операнд, чем правый, или равен ему. Если больше или равен, то выражение становится истинным	print(6 >=44)	False
<=	Проверяет, меньше левый операнд, чем правый, или равен ему. Если меньше или равен, то выражение становится истинным	print(32 <= 232)	True

In [3]:

```
6 != 6.0
```

Out[3]:

```
False
```

2.11.3 Операторы присваивания

Оператор	Описание	Пример использования	Результат
=	Присваивает значение правого операнда левому	var = 51	var = 51
+=	Прибавляет значение правого операнда к левому и присваивает левому. a += b эквивалентно записи a = a + b	var += 4	var = 55
-=	Отнимает значение у левого операнда правое и присваивает левому. a -= b эквивалентно записи a = a - b	var -= 2	var = 49
*=	Умножает значение левого операнда на правое и присваивает левому. a *= b эквивалентно записи a = a * b	var *= 10	var = 510
/=	Делит значение левого операнда на правое и присваивает левому. a /= b эквивалентно записи a = a / b	var /= 4	var = 12.75
%=	Делит значение левого операнда по остатку на правое и присваивает левому. a %= b эквивалентно записи a = a % b	var %= 10	var = 1
**=	Возводит значение левого операнда в степень правого и присваивает левому. a **= b эквивалентно записи a = a ** b	var **= 2	var = 2601
//=	Целочисленно делит значение левого операнда на правое и присваивает левому. a //= b эквивалентно записи a = a // b	var //= 4	var = 13

В Python можно поменять значения у двух переменных без использования временной переменной командой: `a,b = b,a` (значение переменных меняется местами).

In [26]:

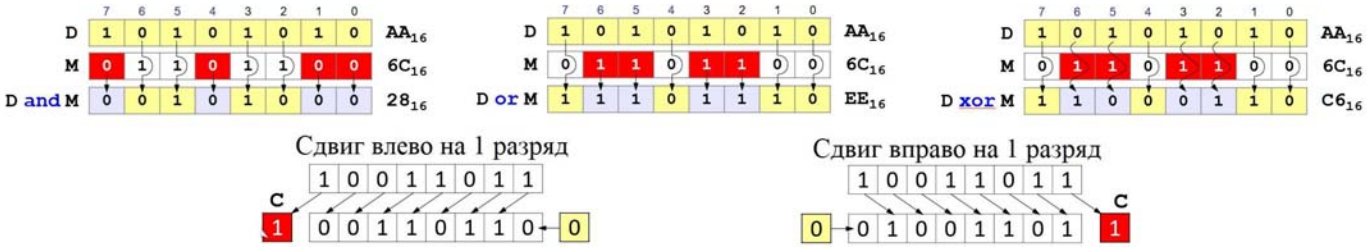
```
x = 23
x //= 2
x
```

Out[26]:

```
11
```

2.11.4 Побитовые операторы

Побитовые операторы работают с данными в двоичной системе счисления. Например число 13 в двоичной системе будет равно 1101



Логический (побитовый) сдвиг влево на 1 разряд увеличивает целое положительное число вдвое.
Логический (побитовый) сдвиг вправо на 1 делит целое положительное число нацело на 2.

Оператор	Описание	Пример использования / Результат
&	Бинарный "И" оператор, копирует бит в результат только если бит присутствует в обоих операндах	`0&0=0; 1&0=0; 0&1=0; 1&1=1; 101 & 110 = 100`
|	Бинарный "ИЛИ" оператор копирует бит, если тот присутствует в хотя бы в одном операнде	`0|0=0; 1|0=1; 0|1=1; 1|1=1; 101 | 110 = 111`
^	Бинарный "Исключительное ИЛИ" оператор копирует бит только если бит присутствует в одном из операндов, но не в обоих сразу	`0^0=0; 1^0=1; 0^1=1; 1^1=0; 101 ^ 110 = 011`
~	Побитовая операция "НЕ". Для числа `a` соответствует `-(a+1)`	`~1 = -10; ~0 = -1; ~101 = -102`
>>	Побитовый сдвиг вправо. Значение левого операнда "сдвигается" вправо на количество бит указанных в правом операнде	`100 >> 2 = 001`
<<	Побитовый сдвиг влево. Значение левого операнда "сдвигается" влево на количество бит указанных в правом операнде	`100 << 2 = 10000`

In [32]:

```
print(0b1010<<3)
```

```
80
```

In [29]:

```
print(0b10101010&0b1101100)
```

```
40
```

2.11.5 Логические операторы

Оператор	Описание	Пример использования	Результат
`and`	Логический оператор "И". Условие будет истинным если оба операнда истина	`True and False`	`False`
`or`	Логический оператор "ИЛИ". Если хотя бы один из операндов истинный, то и все выражение будет истинным	`False or False`	`False`
`not`	Логический оператор "НЕ". Изменяет логическое значение операнда на противоположное	`not True`	`False`

В [21]:

```
False and True
```

Out[21]:

```
False
```

2.11.6 Операторы членства

> **Операторы членства** участвуют в поиске данных в некоторой последовательности.

Оператор	Описание	Пример использования	Результат
in	Возвращает истину, если элемент присутствует в последовательности, иначе возвращает ложь	print('he' in 'hello')	True
not in	Возвращает истину если элемента нет в последовательности	print(12 not in [1, 2, 4, 56])	True

В [22]:

```
55 in ['55', 23, 18]
```

Out[22]:

```
False
```

2.11.7 Операторы тождественности

> **Операторы тождественности** помогают сравнить размещение двух объектов в памяти компьютера.

Оператор	Описание	Пример использования	Результат
is	Возвращает истину, если оба операнда указывают на один объект	a = 2; b = 2; a is b	True
is not	Возвращает ложь, если оба операнда указывают на один объект	a = 1; c = 2; a is not c	True

В [31]:

```
ls_1 = ls_2 = [2, 4, 'a']
ls_1 is ls_2
```

Out[31]:

```
True
```

In [32]:

```
ls_1[0] = 5
print(ls_1)
print(ls_2)
```

```
[5, 4, 'a']
[5, 4, 'a']
```

In [33]:

```
ls_3 = [5, 6, 'b']
ls_4 = [5, 6, 'b']
ls_3 is ls_4
```

Out[33]:

```
False
```

2.11.8 Приоритет операций

В таблице представлены операции Python в порядке убывания приоритета выполнения

Операция	Описание действия
**	Возведение в степень
~ + -	Комплементарные операции
* / % //	Умножение, деление, деление по модулю, целочисленное деление
+ -	Сложение и вычитание
>> <<	Побитовый сдвиг вправо и влево
&	Бинарное И
^ |	Бинарное ИСКЛЮЧАЮЩЕЕ ИЛИ и бинарное ИЛИ
< <= > >=	Операции сравнения (меньше, меньше или равно,больше, больше или равно)
<>== ! =	Операции сравнения на равенство
= %= /= //= -= += *= **=	Присваивание
is is not	Тождество и не тождество
in not in	Принадлежит и не принадлежит множеству
not or and	Логическое отрицание, сложение и умножение

2.11.9 Ленивые логические выражения

Следующая особенность Python – это **ленивые логические выражения**.

Подробно разберем их на примере.

Пример 1. У нас есть переменная x, которая равняется 12 и переменная y, которая равняется `false`. И есть логическое выражение `x or y`.

В [13]:

```
x = 12
y = False

print(x or y)
```

```
12
```

В результате работы этого выражения мы видим, что на экране не `true`, как мы могли бы ожидать, а число 12. Что здесь происходит? Python начинает интерпретировать логическое выражение, видит, что x является истинным, и понимает, что ему не нужно выполнять оставшуюся часть логического выражения. Поэтому он у уже не будет проверять, x – это истина, оператор `or` стоит, значит нас устраивает то, что мы можем остановиться в этот момент и результатом выполнения выражения будет как раз значение x.

Пример 2. Пусть есть переменная x, которая равна 12, и переменная z, которая равна строке `boom`. В результате работы логического выражения `x and z` мы получаем как результат работы логического выражения строку `boom`.

В [14]:

```
x = 12
z = 'boom'

print(x and z)
```

```
boom
```

Происходит все то же самое. Python выполняет логическое выражение, до тех пор, пока оно имеет смысл, и результатом является последнее значение.

§ 2.12 Модуль random

Python порождает случайные числа на основе формулы, так что они не на самом деле случайные, а, как говорят, **псевдослучайные**.

Модуль `random` позволяет генерировать случайные числа. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import random
```

Оператор	Описание	Пример использования	Результат
`random.random()`	Возвращает псевдослучайное число из диапазона `[0.0; 1.0)`	`random.random()`	0.8382604480558495

Оператор	Описание	Пример использования	Результат
`random.seed(<Параметр>)`	Настраивает генератор случайных чисел на новую последовательность. По умолчанию используется системное время. Если значение параметра будет одиноким, то генерируется одинокое число		
`random.uniform(beg, end)`	Возвращает псевдослучайное вещественное число из диапазона [beg; end]	`random.uniform(0, 20)`	18.092324756265473
`random.randint(beg, end)`	Возвращает псевдослучайное целое число из диапазона [beg; end]	`random.randint(10, 25)`	19
`random.choince(<Последовательность>)`	Возвращает случайный элемент из любой последовательности (строки, списка, кортежа)	`random.choice('@#%&_')`	&
`random.randrange(beg, end, step)`	Возвращает случайно выбранное число из последовательности	random.randrange(2, 64, 4)	26
`random.shuffle(<Список>)`	Перемешивает последовательность (изменяется сама последовательность). Поэтому функция не работает для неизменяемых объектов	`List = [1,2,3,4,5,6,7,8,9]; random.shuffle(List); List`	`[6, 7, 1, 9, 5, 8, 3, 2, 4]`

Пример. Сгенерировать случайный пароль для пользователя. Длина пароля - от 12 до 20 символов.

В [43]:

```
import random

digit = '0123456789'                    # строка из цифр
small = 'qwertyuiopasdfghjklzxcvbnm'    # строка из строчных букв
big = small.upper()                     # строка из прописных букв

stroka = digit + small + big            # соединяем все строки в одну

ls = list(stroka)       # преобразуем получившуюся строку в список
random.shuffle(ls)      # перемешиваем список
len_psw = random.randint(12, 20)       # слычайно определим длину пароля
psw = ''.join([random.choice(ls) for x in range(len_psw )]) # извлекаем из списка len_psw

print(psw)  # посмотрим пароль
```

```
eT95TjuCkHnC7B
```

§ 2.13 Модуль math

Для использования стандартных математических функций необходимо подключить модуль `math` :

`from math import *` (или просто `import math`)

При вызове соответствующей функции необходимо указать `<имя модуля>.<имя функции>` Основные функции представлены в таблице:

Таблица. Стандартные математические функции

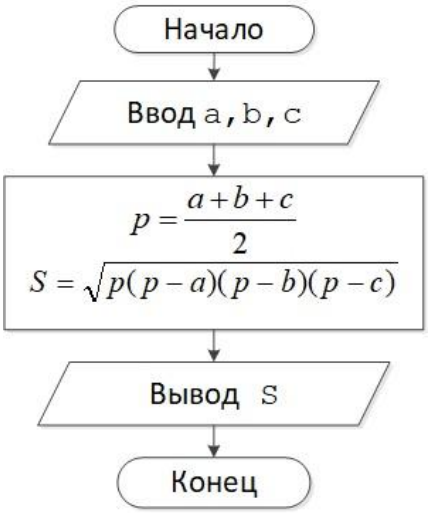
Синтаксис функции	Синтаксис функции
`math.ceil(X)`	Округление значения X до целого вверх
`math.copysign(X, Y)`	Возвращает число, имеющее модуль такой же, как и у числа X, а знак – как у числа Y
`math.fabs(X)`	Модуль X
`math.factorial(X)`	Факториал числа X, т.е. $X!$
`math.floor(X)`	Округление значения X до целого вниз
`math.fmod(X, Y)`	Остаток от деления X на Y
`math.frexp(X)`	Возвращает мантиссу и экспоненту числа. $X \rightarrow A * 2^B$. Применение: `A,B = math.frexp(X)`
`math.ldexp(X, I)`	Возвращает значение $X * 2^I$. Функция, обратная функции `math.frexp()`
`math.fsum(a1,…, an)`	Сумма всех членов последовательности $a_1, a_2, \ldots, a_n$. Эквивалент встроенной функции `sum()` , но `math.fsum()` более точна для чисел с плавающей точкой
`math.isfinite(X)`	Является ли X числом
`isinf(X)`	Возвращает `True` , если X является бесконечностью, иначе — `False`
`math.isnan(X)`	Возвращает `True` , если X является `NaN` (*Not a Number* – не число), иначе — `False`
`math.modf(X)`	Возвращает дробную и целую часть числа X. Оба числа имеют тот же знак, что и X. Например,
`math.modf(20.1) → (0.1, 20)`	
`math.trunc(X)`	Усекает значение X до целого
`math.exp(X)`	e^X
`math.expm1(X)`	$e^X - 1$. При $X \rightarrow 0$ точнее, чем $math.exp(X) - 1$
`math.log(X, [base])`	Логарифм X по основанию $base$. Если $base$ не указан, то вычисляется натуральный логарифм
`math.log1p(X)`	Натуральный логарифм $(1 + X)$. При $X \rightarrow 0$ точнее, чем $math.log(1 + X)$
`math.log10(X)`	Логарифм X по основанию 10
`math.log2(X)`	Логарифм X по основанию 2
`math.pow(X, Y)`	X^Y
`math.sqrt(X)`	Квадратный корень из X. Альтернатива: $X * *(1/2)$. Но $X * *(1/2)$ вернет комплексное число для отрицательного X, а $sqrt(X)$ выдаст ошибку
`math.acos(X)`	Арккосинус X радианах
`math.asin(X)`	АрксинусX. В радианах
`math.atan(X)`	Арктангенс X. В радианах
`math.atan2(Y, X)`	Арктангенс Y/X. В радианах. С учетом четверти, в которой находится точка (X, Y)

Синтаксис функции	Синтаксис функции
`math.cos(X)`	Косинус X (X указывается в радианах)
`math.sin(X)`	Синус X (X указывается в радианах)
`math.tan(X)`	Тангенс X (X указывается в радианах)
`math.hypot(X, Y)`	Вычисляет гипотенузу треугольника с катетами X и Y ($math.\ sqrt(\mathrm{x} * \mathrm{x} + \mathrm{y} * \mathrm{y})$)
`math.degrees(X)`	Конвертирует радианы в градусы
`math.radians(X)`	Конвертирует градусы в радианы
`math.cosh(X)`	Вычисляет гиперболический косинус
`math.sinh(X)`	Вычисляет гиперболический синус
`math.tanh(X)`	Вычисляет гиперболический тангенс
`math.acosh(X)`	Вычисляет обратный гиперболический косинус
`math.asinh(X)`	Вычисляет обратный гиперболический синус
`math.atanh(X)`	Вычисляет обратный гиперболический тангенс
`math.erf(X)`	Функция ошибок $erf(X)$
`math.erfc(X)`	Дополнительная функция ошибок $(1 - math.\ erf(X))$
`math.gamma(X)`	Гамма-функция $\Gamma(X)$
`math.lgamma(X)`	Натуральный логарифм гамма-функции X
`math.pi`	$\pi = 3,1415926...$
`math.e`	$e = 2,718281...$

§ 2.14 Контрольное задание. Программирование линейных алгоритмов

Задание. Определить площадь треугольника по трем сторонам по формуле Герона $S = \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$, где $p = \frac{a+b+c}{2}$.

Блок-схема



In [56]:

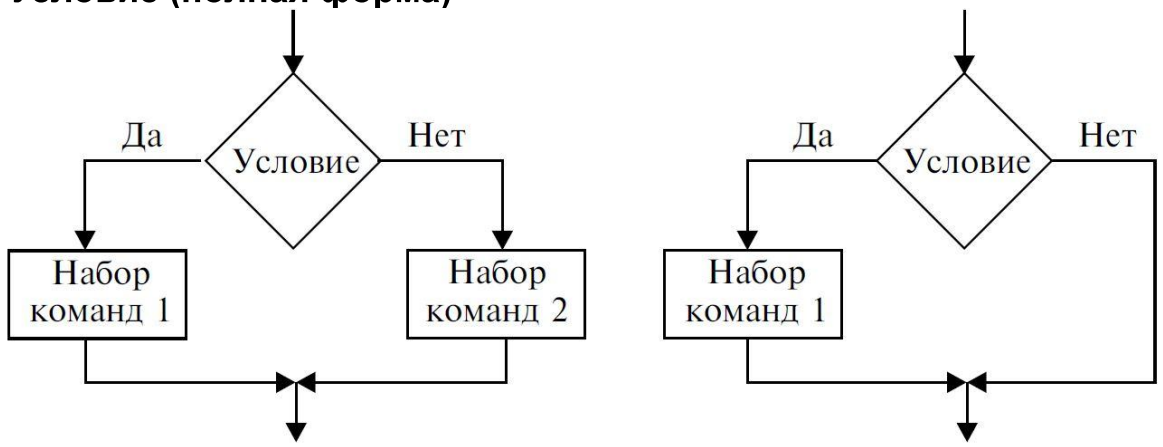
```
from math import cos, sin, sqrt

a, b, c = map(float, input("Введи a, b, c: ").split())
p = (a + b + c) / 2
s = sqrt(p * (p - a) * (p - b) * (p - c))
print("Площадь треугольника со сторонами {0:0.2f}, {1:0.2f}, {2:0.2f} равна: {3:0.3f}".fo
```

```
Введи a, b, c: 6 6 6
Площадь треугольника со сторонами 6.00, 6.00, 6.00 равна: 15.588
```

§ 2.15 Организация условного перехода

Рассмотрим использование условного оператора `if` . Вот его шаблон:

2.15.1 Условие (полная форма)



Синтаксис:

```
if условие:
    что делать, если оно выполнено
[else:
    что делать, если оно не выполнилось]
```

Для обозначения необязательной части конструкции языка используются квадраьные скобки [. . .].

Пример. Является ли введенное слово длинным (слово считается длинным, его в нем более 10 букв).

In [47]:

```
word = input('Введите слово: ')

if len(word) > 10:
    print('Это слово длинное.')
else:
    print('Это слово короткое.')
```

```
Введите слово: квинтиллион
Это слово длинное.
```

Набор команд вида

```
if a > b:
    M = a
else:
    M = b
```

достаточно прост, однако все равно занимает целых 4 строки. Для таких случаев В Python присутствует **тернарный оператор** в виде конструкции следующего вида (в некоторых случаях это удобно):

```
M = a if a > b else b
```

Пример. Переменной будет присвоена строка *Argentina* в том случае, если условие выполнено, иначе будет присвоена строка *Jamaica*.

В [5]:

```
score_1 = 5
score_2 = 0
winner = 'Argentina' if score_1 > score_2 else 'Jamaica'
print(winner)
```

```
Argentina
```

Пример. Является ли введенная переменная x положительной, отрицательной или она равна нулю.

В [23]:

```
x = int(input('Введите x: '))

if x > 0:
    print('Результат положителен')
else:
    if x < 0:
        print('Результат отрицателен')
    else:
        print('Результат равен нулю')
```

```
Введите x: 0
Результат равен нулю
```

Булевы операторы `and` и `or` в условиях выполняются слева направо до тех пор, пока не станет ясен общий результат выражения (аналогично тому, как это сделано, например, в C++).

Пример. Определите, является ли год високосным.

Пояснение: год считается високосным, если его номер кратен 4, но не кратен 100, а также если он кратен 400.

В [51]:

```
# Вариант 1 (if - else)

year = int(input("Введи год: "))

if year % 4 == 0 and year % 100 != 0 or year % 400 == 0:
    print("Год високосный")
else:
    print("Год не високосный")
```

```
Введи год: 2100
Год не високосный
```

Эту задачку **можно решить короче**, используя функцию `isleap` из модуля стандартной библиотеки `calendar`

В [2]:

```
# Вариант 2 (покороче)

import calendar

year = int(input("Введи год: "))

if calendar.isleap(year):
    print("Год високосный")
else:
    print("Год не високосный")
```

```
Введи год: 2000
Год високосный
```

В условиях **можно использовать множественное сравнение**. По этой причине в следующем примере ошибки не возникнет:

В [35]:

```
height = float(input('Укажите свой рост (м): '))
weight = int(input('Укажите свой вес (кг): '))

imt = weight/ height/ height

if 18.5 <= imt <=24.99:
    print('Норма')
elif imt < 18.5:
    print('Недостаточная масса тела')
else:
    print('Избыточная масса тела')
```

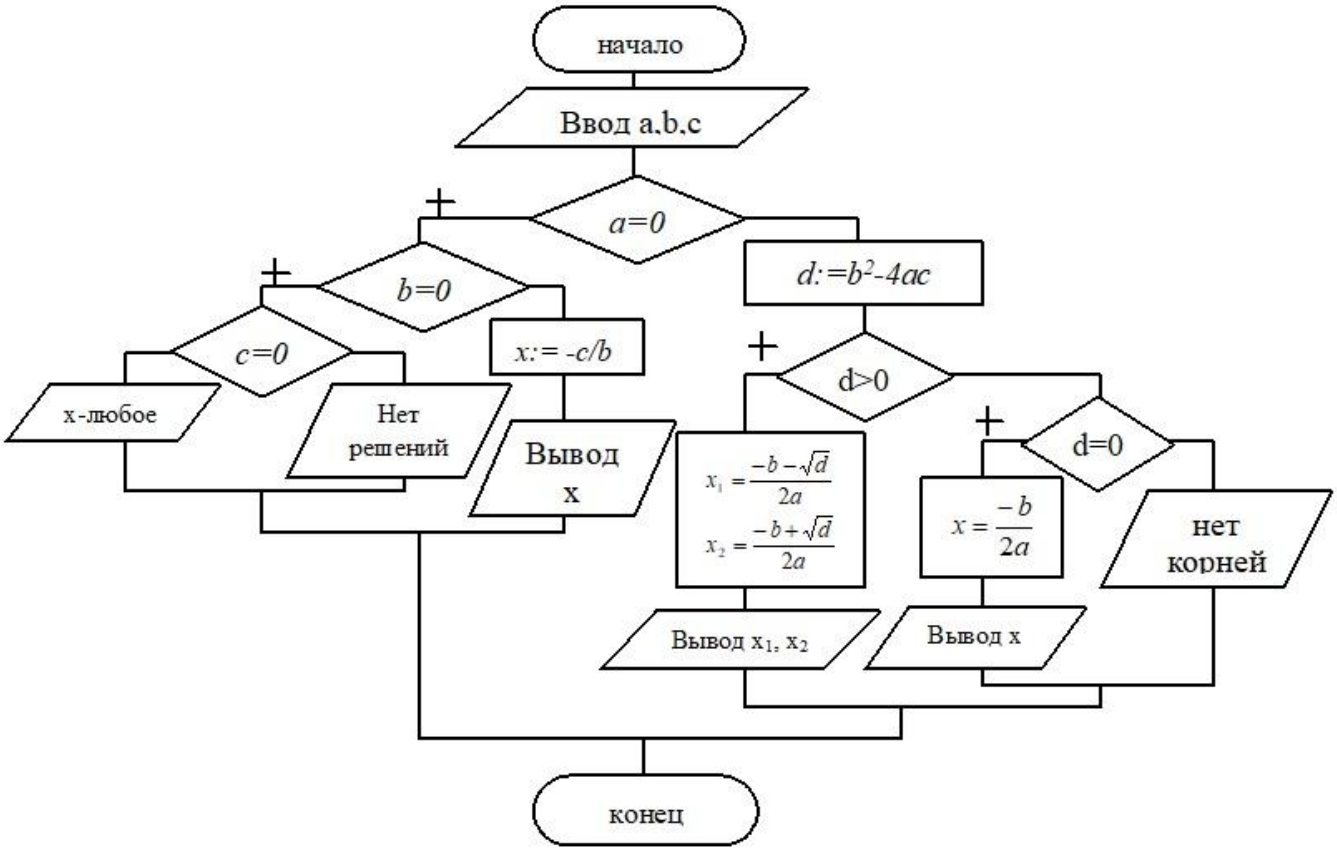
```
Укажите свой рост (м): 1.67
Укажите свой вес (кг): 70
Избыточная масса тела
```

2.15.2 Контрольное задание. Программирование разветвляющихся

алгоритмов

Задача. Решить квадратное уравнение $ax^2 + bx + c = 0$

Блок-схема



In [44]:

```
a, b, c = map(float, input('Введите коэффициенты квадратного уравнения: ').split())

if a == 0:
    if b == 0:
        if c == 0:
            print('x - любое')
        else:
            print('Нет решений')
    else:
        x = -c / b
        print('Корень равен %.2f.' % x)
else:
    d = b**2 - 4 * a * c
    if d > 0:
        x1 = (-b - d**0.5) / (2 * a)
        x2 = (-b + d**0.5) / 2 / a
        print("Корени уравнения: %.2f и %.2f." % (x1, x2))
    elif d == 0:
        x = -b / 2 / a
        print("Корень равен %.2f." % x)
    else:
        print('Нет корней')
```

```
Введите коэффициенты квадратного уравнения: 1 -3 -8
Корени уравнения: -1.70 и 4.70.
```

2.15.3 Условие (каскадное ветвление)

Синтаксис:

```
if условие:
    что делать, если оно выполнено
elif условие:
    что делать, если оно выполнено
else:
    что делать, если не выполнилось ни одно условие выше
```

Всё, что происходит внутри условия, отделяется отступом.

Пример. По номеру месяца определить время года.

В [67]:

```
num = int(input('Введи номер месяца: '))

if num in [12, 1, 2]:
    print('Зима')
elif 3 <= num <= 5:
    print('Весна')
elif 6 <= num <= 8:
    print('Лето')
elif 9 <= num <= 11:
    print('Осень')
else:
    print('Ошибка ввода месяца!')
```

```
Введи номер месяца: 2
Зима
```

Задача. В старояпонском календаре принят 12-летний цикл. Годы внутри цикла носят названия животных: крысы, коровы, тигра, зайца, дракона, змеи, лошади, овцы, обезьяны, курицы, собаки, свиньи. Составить программу, которая по номеру года определяет название соответствующего животного по старояпонскому календарю.

In [24]:

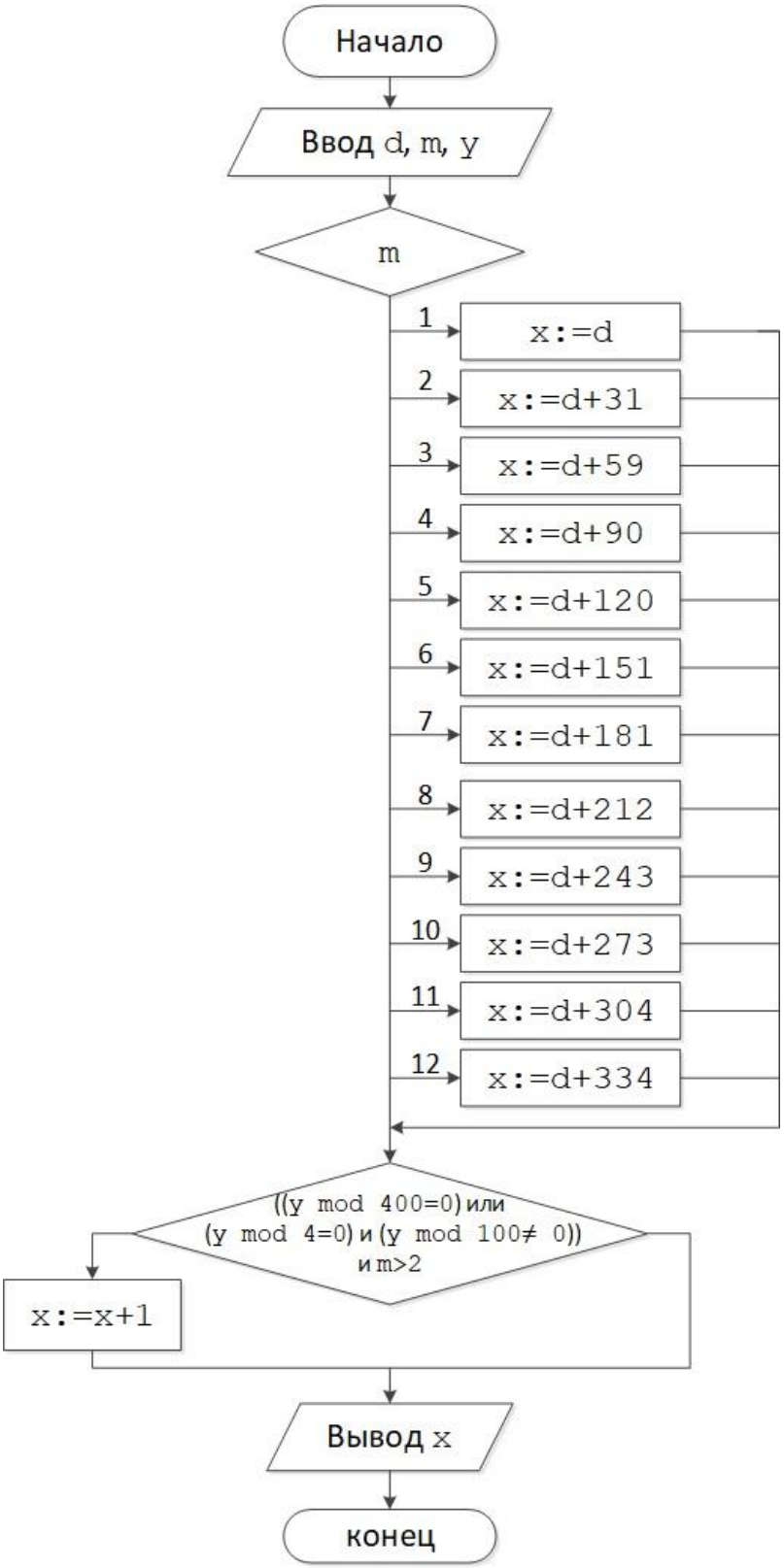
```
year = int(input('Введите год: '))

y = year % 12

print('%d год - это год ' % year, end='')
if y == 0:
    print('обезьяны')
elif y == 1:
    print('курицы')
elif y == 2:
    print('собаки')
elif y == 3:
    print('свиньи')
elif y == 4:
    print('крысы')
elif y == 5:
    print('коровы')
elif y == 6:
    print('тигра')
elif y == 7:
    print('зайца')
elif y == 8:
    print('дракона')
elif y == 9:
    print('змеи')
elif y == 10:
    print('лошади')
elif y == 11:
    print('овцы')
```

```
Введите год: 2014
2014 год - это год лошади
```

2.15.4 Контрольное задание. Программирование множественного выбора

Задача. Вычислить количество дней, прошедших с начала года по заданным числу, месяцу и году.

Блок-схема


Начало
Ввод d, m, y
m
1
x:=d
2
x:=d+31
3
x:=d+59
4
x:=d+90
5
x:=d+120
6
x:=d+151
7
x:=d+181
8
x:=d+212
9
x:=d+243
10
x:=d+273
11
x:=d+304
12
x:=d+334
((y mod 400=0) или
(y mod 4=0) и (y mod 100≠ 0))
и m>2
x:=x+1
Вывод x
конец

In [1]:

```
d, m, y = map(int, input('Введите дату в формате dd.mm.yyyy: ').split('.'))

if m == 1:
    x = d
elif m == 2:
    x = d + 31
elif m == 3:
    x = d + 59
elif m == 4:
    x = d + 90
elif m == 5:
    x = d + 120
elif m == 6:
    x = d + 151
elif m == 7:
    x = d + 181
elif m == 8:
    x = d + 212
elif m == 9:
    x = d + 243
elif m == 10:
    x = d + 273
elif m == 11:
    x = d + 304
elif m == 12:
    x = d + 334
if (y % 4 == 0 and y % 100 != 0 or y % 400 == 0) and m > 2:
    x += 1
print("От начала года прошло %d дней." % x)
```

```
Введите дату в формате dd.mm.yyyy: 31.12.2020
От начала года прошло 366 дней.
```

§ 2.16 Циклы

2.16.1 Цикл с предусловием (while)

Синтаксис цикла **while**:

```
while условие:
    команды
[else:
    команды]
```

Все, что происходит внутри цикла, отделяется отступом.

Обратите внимание, конструкции Питона, которые заканчиваются двоеточием, сигнализируют о том, что далее ожидается блок — последовательность строк, выделенных отступом. **Не забывайте про двоеточие**!

Пример. Найти сумму и произведение цифр, из которых состоит строка.

В [2]:

```
# Решение
N_str = input("Введи число: ")
if N_str.isdigit():
    N = int(N_str)
    summa = 0
    product = 1
    while N > 0:
        summa += N % 10
        product *= N % 10
        N //= 10
    print("Сумма цифр в числе %s равна %d, произведение равно %d." % (N_str, summa, produ
else:
    print("Входная строка имела неверный формат")
```

```
Введи число: 456
Сумма цифр в числе 456 равна 15, произведение равно 120.
```

2.16.2 Цикл с параметром (for)

Синтаксис цикла **for**:

```
for i in диапазон_изменений_i:
    команды
[else:
    команды]
```

Все, что происходит внутри цикла, отделяется отступом.

Пример. Какие годы между 2000 и 2700 являются високосными. Перечислите все эти годы через запятую.

B [6]:

```python
# Решение
great_years = []

for year in range(2000, 2701):
    if (year % 4 == 0 and year % 100 != 0) or year % 400 == 0:
        great_years.append(str(year))

print(', '.join(great_years))
```

```
2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036, 2040, 2044, 204
8, 2052, 2056, 2060, 2064, 2068, 2072, 2076, 2080, 2084, 2088, 2092, 2096, 2
104, 2108, 2112, 2116, 2120, 2124, 2128, 2132, 2136, 2140, 2144, 2148, 2152,
2156, 2160, 2164, 2168, 2172, 2176, 2180, 2184, 2188, 2192, 2196, 2204, 220
8, 2212, 2216, 2220, 2224, 2228, 2232, 2236, 2240, 2244, 2248, 2252, 2256, 2
260, 2264, 2268, 2272, 2276, 2280, 2284, 2288, 2292, 2296, 2304, 2308, 2312,
2316, 2320, 2324, 2328, 2332, 2336, 2340, 2344, 2348, 2352, 2356, 2360, 236
4, 2368, 2372, 2376, 2380, 2384, 2388, 2392, 2396, 2400, 2404, 2408, 2412, 2
416, 2420, 2424, 2428, 2432, 2436, 2440, 2444, 2448, 2452, 2456, 2460, 2464,
2468, 2472, 2476, 2480, 2484, 2488, 2492, 2496, 2504, 2508, 2512, 2516, 252
0, 2524, 2528, 2532, 2536, 2540, 2544, 2548, 2552, 2556, 2560, 2564, 2568, 2
572, 2576, 2580, 2584, 2588, 2592, 2596, 2604, 2608, 2612, 2616, 2620, 2624,
2628, 2632, 2636, 2640, 2644, 2648, 2652, 2656, 2660, 2664, 2668, 2672, 267
6, 2680, 2684, 2688, 2692, 2696
```

B [10]:

```python
for year in range(2000, 2701):
    if (year % 4 == 0 and year % 100 != 0) or year % 400 == 0:
        print(year, end=' ')
```

```
2000 2004 2008 2012 2016 2020 2024 2028 2032 2036 2040 2044 2048 2052 2056 2
060 2064 2068 2072 2076 2080 2084 2088 2092 2096 2104 2108 2112 2116 2120 21
24 2128 2132 2136 2140 2144 2148 2152 2156 2160 2164 2168 2172 2176 2180 218
4 2188 2192 2196 2204 2208 2212 2216 2220 2224 2228 2232 2236 2240 2244 2248
2252 2256 2260 2264 2268 2272 2276 2280 2284 2288 2292 2296 2304 2308 2312 2
316 2320 2324 2328 2332 2336 2340 2344 2348 2352 2356 2360 2364 2368 2372 23
76 2380 2384 2388 2392 2396 2400 2404 2408 2412 2416 2420 2424 2428 2432 243
6 2440 2444 2448 2452 2456 2460 2464 2468 2472 2476 2480 2484 2488 2492 2496
2504 2508 2512 2516 2520 2524 2528 2532 2536 2540 2544 2548 2552 2556 2560 2
564 2568 2572 2576 2580 2584 2588 2592 2596 2604 2608 2612 2616 2620 2624 26
28 2632 2636 2640 2644 2648 2652 2656 2660 2664 2668 2672 2676 2680 2684 268
8 2692 2696
```

In [14]:

```
from calendar import isleap

for year in range(2000, 2701):
    if isleap(year):
        print(year, end=' ')
```

```
2000 2004 2008 2012 2016 2020 2024 2028 2032 2036 2040 2044 2048 2052 2056 2
060 2064 2068 2072 2076 2080 2084 2088 2092 2096 2104 2108 2112 2116 2120 21
24 2128 2132 2136 2140 2144 2148 2152 2156 2160 2164 2168 2172 2176 2180 218
4 2188 2192 2196 2204 2208 2212 2216 2220 2224 2228 2232 2236 2240 2244 2248
2252 2256 2260 2264 2268 2272 2276 2280 2284 2288 2292 2296 2304 2308 2312 2
316 2320 2324 2328 2332 2336 2340 2344 2348 2352 2356 2360 2364 2368 2372 23
76 2380 2384 2388 2392 2396 2400 2404 2408 2412 2416 2420 2424 2428 2432 243
6 2440 2444 2448 2452 2456 2460 2464 2468 2472 2476 2480 2484 2488 2492 2496
2504 2508 2512 2516 2520 2524 2528 2532 2536 2540 2544 2548 2552 2556 2560 2
564 2568 2572 2576 2580 2584 2588 2592 2596 2604 2608 2612 2616 2620 2624 26
28 2632 2636 2640 2644 2648 2652 2656 2660 2664 2668 2672 2676 2680 2684 268
8 2692 2696
```

Пример. Из заданного имени вывести последовательно буквы и их порядковый номер.

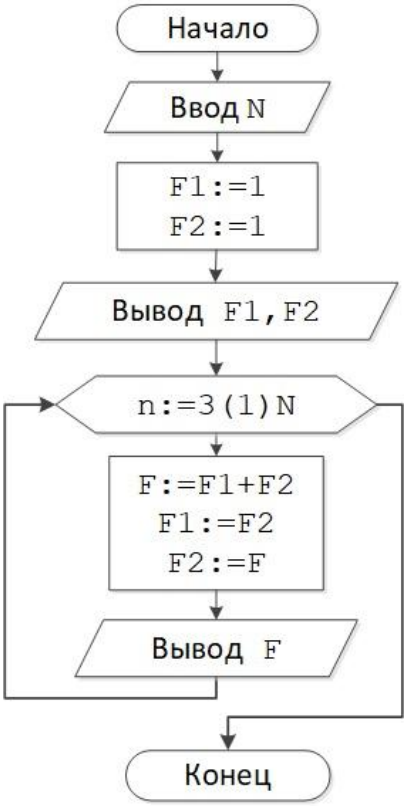
In [7]:

```
name = 'Сергей'
for i, letter in enumerate(name):
    print('Буква',  i+1, 'в этом имени -', letter)
```

```
Буква 1 в этом имени - С
Буква 2 в этом имени - е
Буква 3 в этом имени - р
Буква 4 в этом имени - г
Буква 5 в этом имени - е
Буква 6 в этом имени - й
```

Пример. Вывести N первых чисел Фибоначчи: 1, 1, 2, 3, 5, 8, 13, . . .

Блок-схема



В [8]:

```
f = None
f1, f2 = 1, 1
N = int(input('N = '))
print('1 =>', f1)
print('2 =>', f2)
for n in range(3, N + 1):
    f = f1 + f2
    f1, f2 = f2, f
    print(n, '=>', f)
```

```
N = 5
1 => 1
2 => 1
3 => 2
4 => 3
5 => 5
```

Также как и цикл `while` , `for` имеет необязательный блок `else` , операторы которого выполнятся, когда цикл закончится. Если использовать `break` в цикле `for - else` , то при прерывании цикла код после `else` выполняться не будет.

Иногда бывает нужно узнать, когда цикл закончился, например, когда он очень долго выполняется.

Пример. Использование `else` в операторе `for` .

In [6]:

```
for x in range(6):
    print(x)
else:
    print("Цикл завершен!")
```

```
0
1
2
3
4
5
Цикл завершен!
```

2.16.3 Операторы continue и break. Оператор else

В `for` и `while` допускается использование традиционных операторов `break` и `continue` для преждевременного выхода из цикла.

Оператор `continue`

Оператор `continue` позволяет начать следующий проход цикла, минуя оставшиеся инструкции.

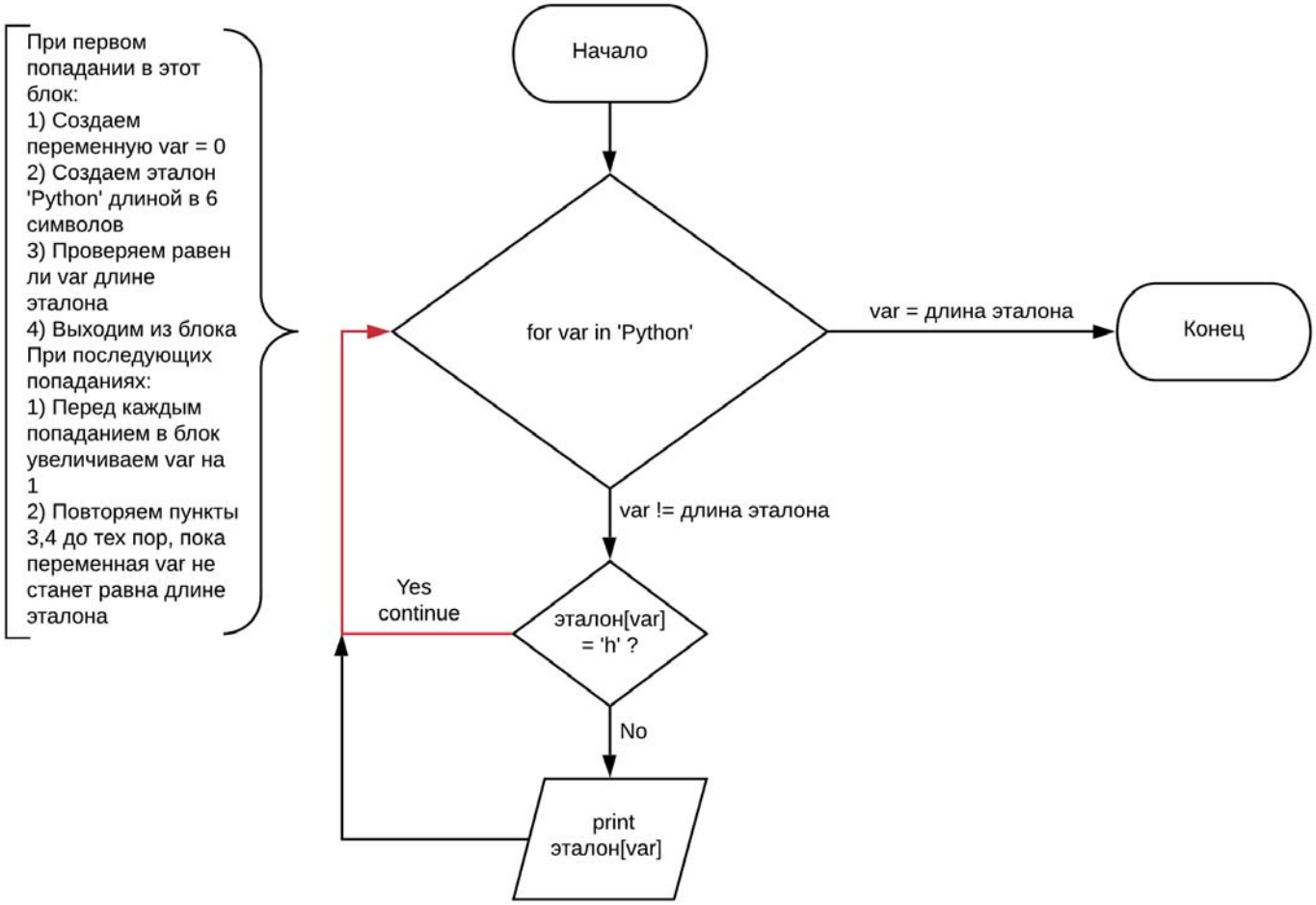
Пример. В строке 'Python' вывести поочередно все буквы, кроме буквы 'h'.

In [19]:

```
for c in 'Python':
    if c == 'h':
        continue
    print(c)
```

```
P
y
t
o
n
```

На примере выше мы перебираем последовательность символов и когда наша переменная хранит в себе символ `h`, мы используем оператор `continue`, чтобы пропустить дальнейшую инструкцию `print(var)`.



На блок-схеме выше представлена ситуация, когда используется оператор `continue`. В текстовой сноске примерно описан вариант реализации цикла `for`. Основное, на что стоит обратить внимание, это красная линия, которая переводит на блок `for` без каких-либо действий.

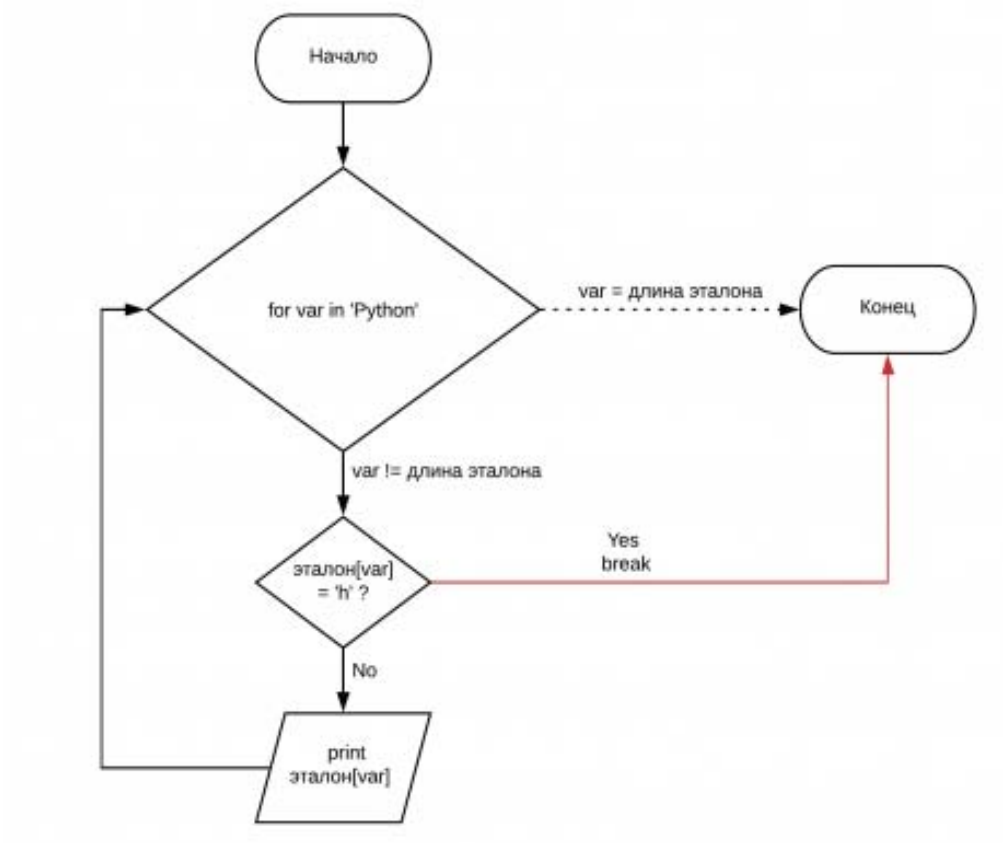
Оператор `break`

Оператор `break` досрочно прерывает цикл. Повторите пример ниже и посмотрите, что получится.

In [18]:

```
for c in 'Python':
    if c == 'h':
        break
    print(c)
else:
    print('Цикл завершен')
```

```
P
y
t
o
n
Цикл завершен
```



В данном случае оператор `break` ведет нас сразу к окончанию работы цикла. При этом получается, что выход совершается экстренный, т.к. мы не переберем все элементы эталона, о чем говорит пунктирная стрелка.

Оператор `else`

Оператор `else` проверяет цикл на экстренный выход (`break`). Если экстренного выхода не было, т.е. оператор `break` не был выполнен, блок инструкций вложенный в оператор `else` - выполняется.

In [18]:

```
for var in 'Python':
    if var == 'a':
        break
else:
    print('Символа a нет в слове Python')
```

```
Символа a нет в слове Python
```

Операторы `continue` , `break` и `else` работают с циклами `for` и `while` .

Пример (совместного использования `break` и `else`). Найти минимальное число, у которого сумма цифр равна s, а произведение - p.

В [24]:

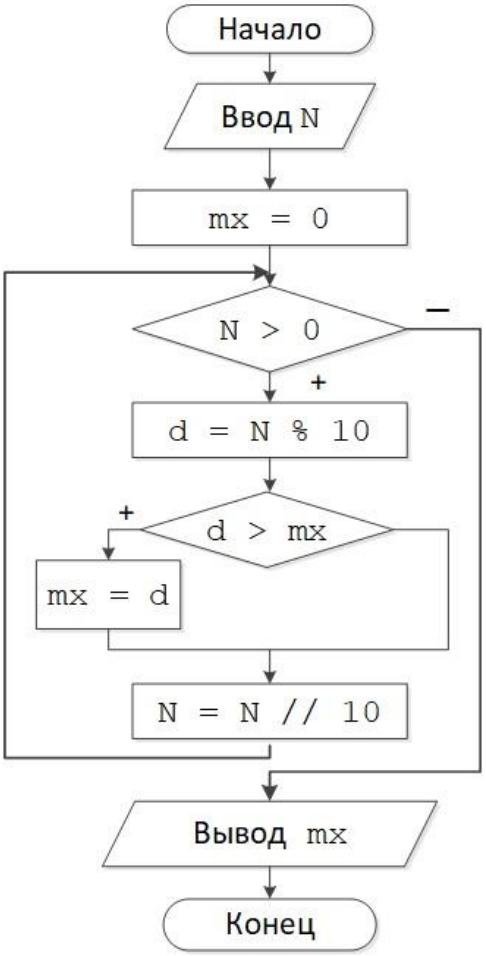
```
lim = 1_000_000
s = 10
p = 30

for n in range(1, lim + 1):
    N = n
    summa = 0
    product = 1
    while N > 0:
        summa += N % 10
        product *= N % 10
        N //= 10
    if summa == s and product == p:
        print(n)
        break
else:
    print('На отрезке [1, %d] таких чисел нет.' % lim)
```

```
235
```

2.16.4 Контрольное задание. Программирование циклических алгоритмов (цикл while)

Задача. Дано натуральное число N, определить в нем максимальную цифру.

Блок-схема



В [19]:

```
N = int(input("Введи число: "))
print("Максимальная цифра в числе %d равна:" % N, end='')
mx = 0
while N > 0:
    d = N % 10
    if d > mx:
        mx = d
    N //= 10
print(mx)
```

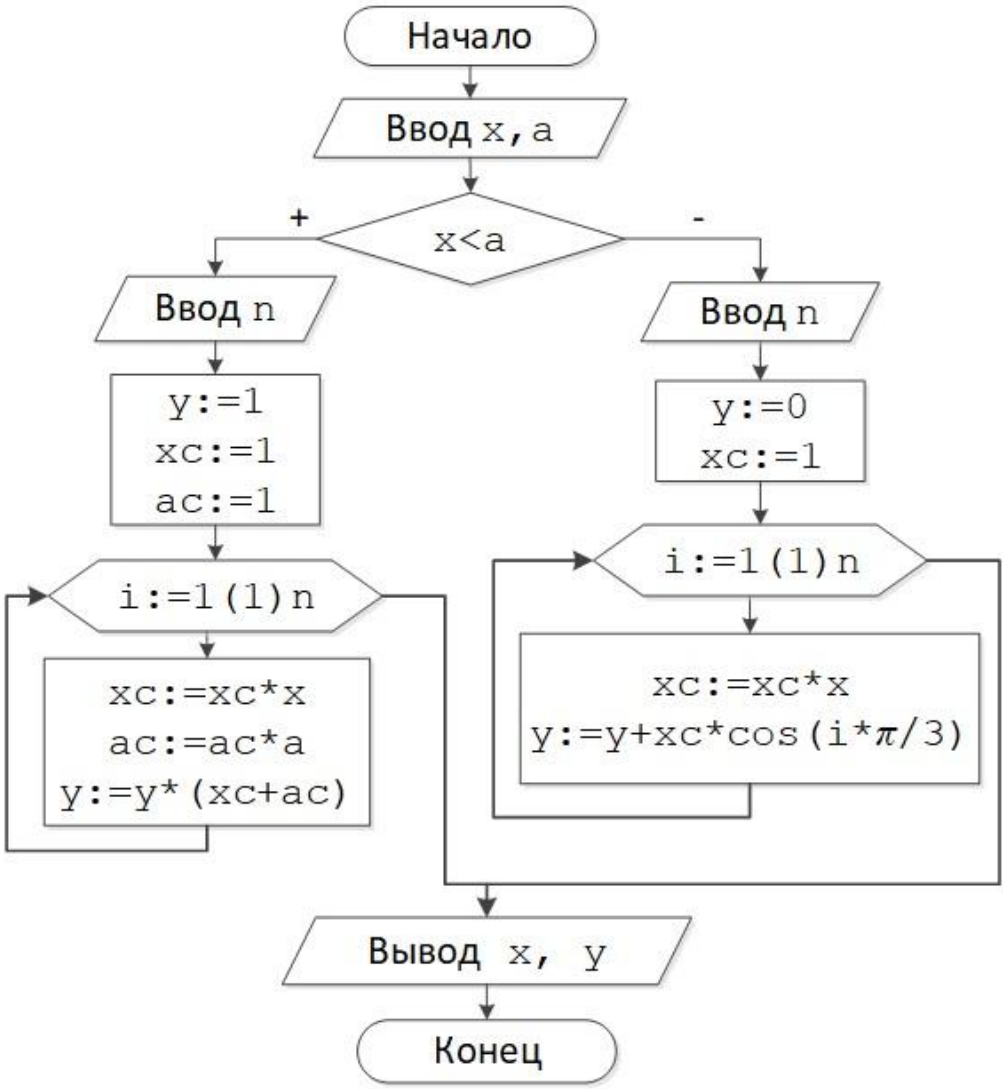
```
Введи число: 45964
Максимальная цифра в числе 45964 равна:9
```

2.16.5 Контрольное задание. Программирование циклических алгоритмов (цикл for)

Задача. Вычислить значение функции y, заданной выражением (при нахождении степени **НЕ** использовать готовый оператор возведения в нужную степень, а найти её накоплением):

$$y = \begin{cases} \sum_{k=1}^{10} x^k \cos\frac{k\pi}{3}, & \text{если } x \geq a; \\ \prod_{m=1}^{6} (a^m + x^m), & \text{если } x < a. \end{cases}$$

Блок-схема


Начало
Ввод x, a
x<a
+
-
Ввод n
y:=1
xc:=1
ac:=1
i:=1(1)n
xc:=xc*x
ac:=ac*a
y:=y*(xc+ac)
Ввод n
y:=0
xc:=1
i:=1(1)n
xc:=xc*x
y:=y+xc*cos(i*π/3)
Вывод x, y
Конец

В [4]:

```
from math import cos, pi

x = float(input('Введите x: '))
a = float(input('Введите a: '))

if x < a:
    n = int(input('Введите количество множителей: '))
    y = 1
    xc = 1
    ac = 1
    for i in range(1, n + 1):
        xc *= x
        ac *= a
        y *= xc + ac
else:
    n = int(input('Введите количество слагаемых: '))
    y = 0
    xc = 1
    for i in range(1, n + 1):
        xc *= x
        y += xc * cos(i * pi/3)

print('При x = %.2f a = %.2f значение y = %.3f.' % (x, a, y))
```

```
Введите x: 1.2
Введите a: 0.5
Введите количество слагаемых: 10
При x = 1.20 a = 0.50 значение y = -7.269.
```

Результаты контрольного расчета (MathCAD):

$$y(x,a) := \text{if}\left[x < a,\ \prod_{m=1}^{6}\left(a^m + x^m\right),\ \sum_{k=1}^{10}\left(x^k \cdot \cos\left(k \cdot \frac{\pi}{3}\right)\right)\right]$$

$y(-1.5, 0.2) = -4390.482$ $\quad$ $y(1.3, 1.3) = -15.575$ $\quad$ $y(1.2, 0.5) = -7.269$