**Тема: Основы языка Python**

**Цель: Рассмотрение среды разработки, типов данных, операторов и функций вывода и ввода данных**

**ОГЛАВЛЕНИЕ**

[Введение](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#vv)[1 Среда разработки](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#1)[2 Использование специальных методов](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#2)[3 Понятие объекта](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#3)[4 Встроенные типы данных](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4)[5 Ввод и вывод данных](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#5)[6 Операторы](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#6)[7 Приоритет операций](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#7)[8 Встроенные идентификаторы](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#8)

**Введение**

Python – мощный и простой в использовании язык программирования, разработанный Гвидо ван Россумом (Guido van Rossum) в 1991 году. Название языка произошло не от вида пресмыкающихся. Автор назвал язык в честь популярного британского комедийного телешоу 1970-х годов «Летающий цирк Монти Пайтона». Тем не менее название языка чаще ассоциируют именно со змеей, нежели с фильмом – пиктограммы файлов в Microsoft Windows изображают змеиные головы.  
Отличительными особенностями языка Python являются:

* простота использования – с точки зрения профессионального программиста легкость языка Python обеспечивает высокую производительность труда: программы на Python простые и требуют меньше времени на разработку и отладку;
* Python – объектно-ориентированный язык, имеющий преимущество перед C# и Java в том, что для этих языков ООП-подход выполняется неукоснительно. Это делает короткие программы избыточно сложными. В Python заложена другая модель – решение о том, использовать или нет ООП-приемы принимает программист в зависимости от типа и сложности программ. Python совмещает функциональность и гибкость;
* легкость интеграции с другими языками программирования, например, C, C++ или Java – программа на Python может использовать сильные стороны этих языков (скажем, быстродействие C, C++) не поступаясь простотой разработки;
* кроссплатформенность – Python может работать на всех платформах: от миниатюрного Palm до суперкомпьютера Cray. На ПК Python поддерживают операционные системы Windows, Linux, Macintosh и др.;
* Python – это бесплатная система с открытым исходным кодом: можно, например делать копии Python, в том числе и модифицированные, можно даже их продавать.

Таким образом, Python – это язык программирования, известный своей простотой и элегантностью исходного кода, эффективностью, а также поддержкой выдающегося сообщества программистов. Благодаря наличию невероятного количества качественных библиотек, Python широко используется в разных областях от применения в NASA и CERN, процессинга электронных платежей и до создания популярных онлайн игр.

**1 Среда разработки**

В комплект поставки Python входит интегрированная среда разработки IDLE (Integrated DeveLopment Environment), в которой редактирование программы выполняется намного удобнее, чем в простом текстовом редакторе или терминале. IDLE также имеет встроенную систему отладки, позволяющую запускать программу построчно, что облегчает процесс поиска ошибок.   
IDLE написана на языке Python с использованием GUI-инструментария tkinter, поэтому легко запускается в любой операционной системе, для которой существует реализация Python.  
Графический пользовательский интерфейс GUI (Graphical User Interface) – это разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений. В отличие от интерфейса командной строки (типа старого DOS системы Windows), в GUI пользователь имеет свободный доступ ко всем элементам интерфейса экрана и может ими управлять с помощью клавиатуры или мыши  
Отличительные черты среды IDLE Python:

* запрограммирована на коде Python с использованием GUI-инструментария tkinter;
* кроссплатформенна: работает на Windows и Unix;
* имеет многооконный текстовый редактор с функцией многократной отмены, подсветкой синтаксиса Python и многими другими свойствами;
* использует отладчик с возможностью задавать точки прерывания, осуществлять просмотр и пошаговое выполнение кода.

Таким образом IDLE Python является интерактивным интерпретатором. Интегрированная среда разработки имеет два основных режима работы:

* интерактивный
* сценарный

**1.1 Программирование в интерактивном режиме**

Для запуска интерактивной сессии необходимо выбрать IDLE (Python GUI) в меню Пуск или на панели задач. На экране появится окно Shell (Python 3.6.0 Shell), содержащее:

* панель инструментов с полями (меню): File, Edit, Shell, Debug, Options, Windows и Help
* информацию:
  + о версии языка: Python 3.6.0 (v3.6.0:41df79263a11;
  + о дате ее создания: Dec 23 2016, 07:18:10;
  + об используемой разрядности: [MSC v.1900 32 bit (Intel)] on win32;
* приглашение к работе в виде трех символов >>>.

Интерактивный режим можно использовать в качестве калькулятора: например, набрав после приглашения 2+3 и нажав Enter, получаем на экране:  
  
>>> 2+3  
  
Но основным назначением интерактивного режима является непосредственное выполнение операторов языка Python. Это может быть использовано для проверки работоспособности отдельных фрагментов программы.  
Например, следующий фрагмент программы имеет две строки:  
  
>>> # Вывод данных на экран  
>>> print "Hello,world!"  
  
Первая сторока начинается с символа # и является комментарием, который не обрабатывается интерпретатором. Комментарий продолжается до конца строки (т.е в языке Python все комментарии являются однострочными, при этом не допускается использование перед символом # кавычек.   
Вторая строка осуществляет вывод сообщения в окно среды разработки. После нажатия клавиши Enter программа начнет выполняться. На экран выводится сообщение о найденной синтаксической ошибке:  
  
>>> SyntaxError: Missing parentheses in call to 'print'

Ошибка возникла из-за того, что для вывода на экран сообщения был использован оператор print, который применяется только в версиях языка Python до 3.x. Версии 3.x обратно не совместимы с версиями 2.x. Об отличиях версии 3.0 от 2.x см. <http://docs.python.org/3/whatsnew/3.0.html>).  
Для версий 3.x языка Python вместо оператора print необходиммо использовать встроенную (built-in) функцию print() (описание встроенных функций см. в <http://docs.python.org/3/library/functions.html>, а также в [подразделе 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#8)). Описание функции print() приведено в [подразделе 5.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#5.1). В данном примере функция вызывается с единственным аргументом, который указывает данные, выводимые на экран:  
  
>>> # Вывод данных на экран  
>>> print ( "Hello,world!" )  
  
После запуска программы на экране появляется сообщение:  
  
Hello,world!  
  
Как видим, среда разработки осуществляет синтаксическую подсветку текста на экране: комментарии и сообщения об ошибке имеют красный цвет, вызовы функций – сиреневый, строки – зеленый, результаты вывода на экран – синий, ключевые (служебные) слова – оранжевый, остальные символы и слова – черный.  
Необходимо отметить, что в отличие от большинства других языков программирования язык Python для разделения операторов использует не точку с запятой, а перевод строки.

**1.2 Использование встроенной системы помощи**

Помимо справочной информации о языке Python, которую можно получить из внешних источников (см. [Введение](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#vv)), интерпретатор имеет внутреннюю систему оказания помощи, реализованную в виде встроенной функции help([object]), которая используется в интерактивном режиме. Если ни один аргумент функции не задан:   
  
>>> help (),  
  
на экран выводится текст, поясняющий работу функции (здесь приводится только часть текста: его первая и последняя строки), а также приглашение к интерактивному режиму помощи (help>):  
  
Welcome to Python 3.3! This is the interactive help utility.  
  
. . .  
  
or summary contain a given string such as "spam", type "modules spam".   
  
help>   
  
Приведем сокращенный перевод этого текста.

*После приветствия указывается о необходимости изучения документации по языку Python и сообщается как получить помощь. Для этого после выведенного на экран приглашения для ввода (help>) необходимо ввести имя любого модуля (англ. module), ключевого слова (англ. keyword), символа (англ. symbol) или темы (англ. topics). Чтобы получить список доступных модулей, ключевых слов, символов или тем языка Python, необходимо ввести соответственно "modules", "keywords", "symbols" или "topics". Для выхода из интерактивного режима помощи в интерактивный режим интерпретатора неоюбходимо ввести "quit".*

Например, для получения полного списка ключевых слов языка необходимо указать "keywords":  
  
help> "keywords"  
  
Here is a list of the Python keywords. Enter any keyword to get more help.

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

help>   
  
Теперь можно получить информацию о любом ключевом слове, набрав его имя после приглашения help>. Для возврата в интерактивный режим интерпретатора необходимо набрать "quit":  
  
help> quit  
  
После этого следует небольшой текст, сообщающий о том, что осуществлен переход к интерпретатору языка Python.   
Если необходимо получить справку о некотором объекте, нужно вызвать функцию help() с указанием имени объекта в качестве ее аргумента. Например, для получения справочной информации о функции print() необходимо вызвать help("print"):  
  
>>> help ( "print" )  
  
На экране появляется информация о назначении функции и ее аргументах:   
  
Help on built-in function print in module builtins:   
  
print(...)  
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)  
  
Prints the values to a stream, or to sys.stdout by default.  
Optional keyword arguments:  
file: a file-like object (stream); defaults to the current sys.stdout.  
sep: string inserted between values, default a space.  
end: string appended after the last value, default a newline.   
flush: whether to forcibly flush the stream.   
  
На русском языке эта информация приведена в [подразделе 5.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#5.1) при описании функции print().  
Укажем, что система помощи работает для модулей, классов и функций, только если те были снабжены строками документации (см. объявления функции и класса соответственно в подразделах [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.1) лаб. раб. №6 и [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB11/lab.htm#1.1) лаб. раб. №11).   
Отметим, что для получения информации о возможностях и режимах работы самой среды IDLE необходимо воспользоваться пунктом "IDLE help" меню "help" панели инструментов (см. [подраздел 1.3.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#1.3.1)).   
Кроме встроенной, для языка Python существуют и улучшенные интерактивные оболочки, напримр, IPython и bpython.

**1.3 Программирование в сценарном режиме**

При работе в интерактивном режиме система отвечает мгновенно. С одной стороны это хорошо, так как сразу можно видеть результат. Но для создания программ, которые сохранялись бы в памяти и могли быть запущены позднее, этот режим не подходит. Для этой цели в среде разработки предусмотрен сценарный режим (см. [подраздел 1.3.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#1.3.2) ). В нем можно создавать, редактировать, загружать и сохранять программу.

**1.3.1 Описание панели инструментов**

Панель инструментов имеет следующие меню, предназначенные для задания различных режимов работы среды разработки IDLE:

* меню File (окна "Shell" и "Editor"):
  + New File (Ctrl+N) – создает окно "Editor" для редактирования нового файла(может быть создано несколько таких окон);
  + Open (Ctrl+O) – открывает существующий файл;
  + Recent Files – открывает список используемых файлов (нужный файл необходимо отметить щелчком мышки);
  + Open Module... (Alt+M) – открывает существующий модуль (поиск ведется в sys.path);
  + Class Browser (Alt+C) – показывает функции, классы и методы в виде дерева в текущем окне "Editor";
  + Path Browser – показывает sys.path директорий, модулей, функций, классов и методов в виде дерева;
  + Save (Ctrl+S) – сохраняет содержимое текущего окна в ассоциированном с ним файле, если он один. Окна, которые были изменены после открытия и последние сохраненные окна отмечаются звездочкой (\*) соответственно перед и после наименования окна. Если ассоциированного файла нет, выполняется команда Save As;
  + Save As... (Ctrl+Shift+S) – сохраняет содержимое окна в указанном файле. Этот файл становится ассоциированным файлом для окна;
  + Save Copy As... (Alt+Shift+S) – сохраняет содержимое окна в другом файле без изменения ассоциированного файла;
  + Print Window (Ctrl+P) – печатает содержимое окна на принтере, заданном по умолчанию;
  + Close (Alt+F4) – закрывает текущее окно (если содержимое окна не сохранено, спрашивает, нужно ли его сохранять);
  + Exit (Ctrl+Q) – закрывает текущие окна и выходит из среды IDLE (если имеются окна с несохраненным содержимым, спрашивает, надо ли его сохранять);
* меню Edit (окна "Shell" и "Editor") – содержит описание команд редактора;
* меню Shell (только окно "Shell"):
  + View Last Restart (F6) – осуществляет скролирование окна "Shell" до последнего рестарта;
  + Restart Shell (Ctrl+F6) – осуществляет рестарт для очистки окружения;
  + Interrupt Execution (Ctrl+C) – прекращает выполнение программы;
* меню Debug (только окно "Shell") – содержит описание команд отладчика;
* меню Format (только окно "Editor") – выполняет форматирование текста программы;
* меню Run (только окно "Editor"):
  + Check Module (Alt+X) – проверяет синтаксис только что открытого модуля в окне "Editor". Если модуль не сохранен, спрашивает, надо ли сохранять. Если обнаружена ошибка, указывается в окне "Editor" ее приблизительное местоположение;
  + Run Module (F5) – осуществляет проверку модуля (как в пункте Check Module). Если ошибок не обнаружено, выполняет модуль. Результаты выполнения помещаются в окно "Shell", которое остается активным. В это время можно использовать полученные результаты;
* меню Options (окна "Shell" и "Editor") – выполняет конфигурирование среды IDLE (задает шрифт, его размер, цвет и др.);
* меню Window (окна "Shell" и "Editor"):
  + Zoom height (Alt+2) – измененяет высоту окна;
  + "Python 3.6.0 Shell" – возвращает к окну "Shell";
* меню Help (окна "Shell" и "Editor"):
  + about IDLE – содержит общие сведения о среде IDLE;
  + IDLE help – описывает режимы работы и возможности среды IDLE;
  + Python Docs (F1) – указывает ссылки на документацию языка Python 3.6.0;
  + Turtle Demo – демонстрирует возможности модуля turtle, который разработан как средство обучения детей программированию.

**1.3.2 Работа в сценарном режиме**

В сценарный режим можно перейти из интерактивного, выбрав в меню File пункт New Window (Cntr+N) Появляется новое окно "Edit". В нем необходимо создать текст программы, например,   
  
# -\*- coding utf-8 -\*-  
print ( "Hello, world!" )

Затем, выбрав в меню File пункт Save As, задать путь к файлу, который будет содержать программу, указав расширение .py, например, 1.py, и нажать Сохранить.  
Если первой строкой кода является строка указанного в примере вида, т.е. начинающаяся с символов # -\*- . . . , она указывает кодировку, в которой был создан текст программы – в данном случае в кодировке UTF-8 (от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-битный») — одна из общепринятых и стандартизированных кодировок текста, которая позволяет хранить символы Юникода, используя переменное количество байт (от 1 до 6).   
В версиях Python 3.x все символы по умолчанию представляются в кодировке Юникода , в отличие от версий Python 2.x, в которых такой кодировкой яляляется ASCII. Поэтому требуется преобразовать тексты всех внешних файлов, в том числе и программ, в кодировку UTF-8. Например, если текст программы был создан в кодировке cp-1252 (window-1252), то необходимо в первой строке кода указать:  
  
# -\*- coding cp-1252 -\*- .  
  
Ддя внешних файлов, созданных текстовыми редакторами в кодировке UTF-8 (в том числе при разработке программ с использованием интегрированной среды разработки IDLE) задание в первой строке кодировки:  
  
# -\*- coding utf-8 -\*-

не является обязательным (предполагается заданным по умолчанию) и в последующих примерах использоваться не будет.   
Отметим, что в тех случаях, когда в программе необходимо указать путь к интерпретатору, например при использовании скриптов на языке Python в качестве серверных приложений, первая строка кода должна указывать путь к интерпретатору (см. [подраздел 2.2.3](https://py-khpi.github.io/!nfo/doc/lab/LAB10/lab.htm#2.2.3) лаб. раб. №10), а кодировка текста (при необходимости) будет указываться во второй строке.   
Для запуска программы с предварительной проверкой нужно выбрать пункт Run Module в меню Run или нажать F5. В окне Shell добавляется строка RESTART и результат работы программы:   
  
>>> ================================ RESTART ================================  
Hello,world!  
  
Для последующего запуска этой программы из интерактивного режима необходимо, выбрав в меню File пункт Open, загрузить программу 1.py в окно "Edit" и запустить ее на выполнение, как было указано выше.  
Если по результатам выполнения программы необходимо внести в нее изменения, то они вносятся в окне "Edit" и сохраняются в памяти путем выбора в меню File пункта Save.   
Однако, если попытаться запустить эту программу не из среды разработки Python, а по двойному щелчку мышки по файлу с расширением .py, то на экране промелькнет консольное окно и исчезнет. Чтобы удержать результат работы программы на экране, необходимо добавить в программу функцию input():  
  
print ( "Hello, world!" )   
input ( "Для завершения программы нажмите Enter" )   
  
Описание функции input() приведено в [подразделе 5.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#5.2).  
Отметим, что одновременное использование интерактивного и сценарного режимов при создании и отладке программы создает предпосылки для наиболее эффективной и качественной работы.

**2 Использование специальных методов**

В языке Python знаки подчеркивания (англ. underscores) помимо обычного использования в идентификаторах подобно другим языкам программирования (см. [подраздел 6.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#6)) имеют и специальное назначение:

* одиночный знак подчеркивания, стоящий в начале имени объекта указывает на его ограниченное (англ. weak) использование. Например, при работе оператора:   
    
  >>> from M import \*  
    
  к программе будут подключены все объекты модуля M кроме тех, чьи имена начинаются с одиночного знака подчеркивания (подключение модулей см. в [подразделе 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#1.1) лаб. раб. №2). Другим примером является применение начального знака подчеркивания доя наименования атрибутов, предназначенных для использования внутри описания класса, но разрешенных для даступа из клиентского кода (см. [подраздел 2.3](https://py-khpi.github.io/!nfo/doc/lab/LAB11/lab.htm#2.3) лаб. раб. №11);
* одиночный знак подчеркивания, стоящий в конце имени используется для предотвращения совпадения с ключевым словом. Например, для обозначения переменной, в которой произходит накапливание суммы, вместо имени sum (название встроенной функции) можно использовать имя sum\_;
* двойной знак подчеркивания, стоящий в начале имени атрибута, означает его использование только внутри описания класса и запрет доступа из клиентского кода (см. [подраздел 2.3](https://py-khpi.github.io/!nfo/doc/lab/LAB11/lab.htm#2.3) лаб. раб. №11);
* двойные знаки подчеркивания, стоящие как в начале, так и в конце идентификатора, служит для обозначения специальных (англ. special) или, как их еще называют, магических (англ. magic) методов.Такое обозначение специальных методов сделано с целью избежать коллизий в адресном пространстве. Поэтому пользователи не должны использовать знаки подчеркивания таким образом.

Специальные методы во многом составляют ядро яз ыка Python, поскольку:

* используются при описании встроенных типов (классов);
* формируют протоколы:
  + контейнера (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.1) лаб. раб. №3);
  + итератора (см. [подраздел 1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.2) лаб. раб. №3);
  + последовательностей (см. [подраздел 1.4.1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.4.1) лаб. раб. №3);
* вызываются при выполнении арифметических и побитовых операций (см. подразделы [4.1.4](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.4) и [4.1.6](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.6));
* вызываются при выполнении встроенных функций для работы с числами (см. [подраздел 4.1.5](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.5));
* вызываются при выполнении операторов присваивания (см. таблицу 1 в [подразделе 6.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#6));
* вызываются при выполнении операций сравнения (см. [подраздел 6.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#6.2));
* могут быть также применены при описании пользовательских классов (см. [раздел 1](https://py-khpi.github.io/!nfo/doc/lab/LAB11/lab.htm#1) лаб. раб. №11).

**3 Понятие объекта**

Все данные, которые использованы в программах на языке Python, а также отношения между данными, являются объектами. Поэтому понятия "объект" и "данные" являются синонимами. Каждый объект имеет следующие свойства:

* идентичность;
* тип;
* значение.

**3.1 Идентичность объектов**

Идентичность (identity) является неизменяемым свойством объекта в течение времени его существования, задается при его создании и представляет собой уникальное целое число. В реализации CPython указывает на адрес объекта в памяти. Значение идентичности объекта может быть получено с помощью встроенной функции id() (см. описание встроенных функций в [подразделе 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#id)). Например, объект, представленный числом 25, имеет значение идентичности, равное 505894688:  
  
>>> id (25)  
505894688.  
  
Проверить идентичность двух объектов можно с помощью операторов is и is not. Операция is возвращает значение True при их идентичности:   
  
>>> s=5  
>>> s is 5  
True  
  
и значение False при отсутствии идентичности:  
  
>>> s is '5'  
False  
  
Операция is not возвращает противоположное значение:  
  
>>> s is not '5'  
True

**3.2 Типы объектов**

Python является языком со строгой динамической типизацией. Строгая типизация означает следующее:

* любой объект данных (переменная, константа, выражение) в языке всегда имеет строго определенный тип, который фиксируется или на момент компиляции (статическая типизация) или определен во время выполнения (динамическая типизация);
* допускается присваивание переменной только значения, имеющего строго тот же тип данных, что и переменная (эти же ограничения действуют и при передаче параметров функции);
* каждая операция требует параметров строго определенных типов;
* неявные преобразования типов не допускаются;

Каждый объект языка Python имеет свой тип (type), который является неизменяемым свойством и определяет операции, поддерживаемые объектом, и его возможные значения. Поскольку данные являются объектами, а любой объект принадлежит классу (class), который определяет его свойства, то понятия "класс" и "тип" являются синонимами.  
В языке Python все является объектом, в том числе тип объекта и его класс. Для определения типа объекта используется встроенная функция type(), которая, если указан один аргумент (см. описание встроенных функций в [подразделе 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#type)), возвращает его тип (класс) в форме <class 'тип данных'>, например, целое число  
  
>>> type (-55)  
<class 'int'>.  
  
имеет тип int (см. описание целых чисел в [подразделе 4.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1)). Проверим утверждение, что тип объекта сам является объектом:   
  
>>> type ( type (-55))  
<class 'type'>.  
  
Определить класс (тип) объекта также можно, используя атрибут \_\_class\_\_:  
  
>>> 2.3e-1.\_\_class\_\_  
<class 'float'>  
  
(см. описание чисел с плавающей точкой в [подразделе 4.1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.2)).  
При описании языка Python также используется термин "instance", под которым понимается конкретный образец или экземпляр класса, поэтому этот термин является синонимом слова "объект".

**3.3 Значения объектов**

Объекты, в зависимости от того, могут быть изменены их значения или нет, делятся на изменяемые объекты (mutable objects) и неизменяемые объекты (immutable objects). Для определения, явояется ли данный объект изменяемым или нет, можно использовать функцию hash() (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#hash) данной лаб. раб. и [подраздел 1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.3) лаб. раб. №3).   
Неизменяемые объекты могут создаваться и уничтожаться, но не могут менять число, последовательность и значения своих элементов. Все рассматриваемые в данной лабораторной работе объекты (целые числа, числа с плавающей точкой, комплексные числа и данные логиченского типа) являются неизменяемыми обхектами.   
Изменяемые объекты могут изменять значения своих элементов, могут вставлять новые элементы в начале, в середине и в конце последовательности своих элементов. Примером изменяемого типа являются списки (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2) лаб. раб. №3).  
Для задания значений объектам могут быть использованы как литералы (любые разрешенные для данного типа значения, представленные в синтаксически правильной для языка Python форме), так и конструкторы соответствующено типа (см. разделы [4.1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.1), [4.1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.2), [4.1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.3) и [4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.2)).   
Объекты явно не уничтожаются, но после того, как они становятся недоступными, происходит удаление их из памяти системными средствами сборки мусора.

**4 Встроенные типы данных**

Python имеет большое количество встроенных типов данных (для версии 3.6.0 они приведены по адресу <http://docs.python.org/3/library/stdtypes.html>). Наиболее важные из них:

* числа (см. [подраздел 4.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1));
* логический тип (см. [подраздел 4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.2));
* текстовые последовательности (строки) (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2) лаб. раб. №2);
* списки (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2) лаб. раб. №3);
* кортежи (см. [раздел 1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#1) лаб. раб. №4);
* диапазоны (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#2) лаб. раб. №4);
* двоичные последовательности (см. [раздел 3](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3) лаб. раб. №4);
* множества (см. [раздел 1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1) лаб. раб. №5);
* словари (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2) лаб. раб. №5);
* генераторы (см. [подраздел 3](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#3) лаб. раб. №5);
* файловые типы (см. [раздел 1](https://py-khpi.github.io/!nfo/doc/lab/LAB8/lab.htm#1) лаб. раб. №8);
* объект, не имеющий значение (Null object) – возращается функциями, которые не имеют явно заданного оператором return значения. Не поддерживает никаких операций. Имеется только один такой объект, он именуется встроенной константой None (см. [подраздел 8.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#8.2) ). В логическом контексте имеет значение False;
* объект NotImplemented – возвращается при выполнении операций, когда в них указываются типы данных, которые эти операции не поддерживают (см. использование метода \_\_add\_\_() в [подразделе 4.1.4](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.4)). Имеется только один такой объект, он именуется встроенной константой NotImplemented (см. [подраздел 8.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#8.2) ). В логическом контексте NotImplemented имеет значение True.

Кроме них существуют и многие другие типы данных. В языке Python все является объектом, поэтому в нем имеются также и такие типы, как модуль, функция, класс, метод и даже скомпилированный код.

**4.1 Числа**

Числа (англ. numeric types) являются неизменяемыми объектами и подразделяются на:

* целые числа (см. [подраздел 4.1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.1));
* числа с плавающей точкой (см. [подраздел 4.1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.2));
* комплексные числа (см. [подраздел 4.1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.3)).

**4.1.1 Целые числа**

Целые числа со знаком (англ. integers) принадлежат типу (классу) int. Они имеют формально неограниченный размер (фактически их размер ограничен конечным объемом памяти компьютера). Значения целых чисел, во-первых, могут быть заданы в виде целочисленных литералов:

* в десятичной форме:  
  >>> -125;  
  -125
* в десятичной форме с использованием знака подчеркивания:  
  >>> 1\_000\_000; # Эта форма введена в версии 3.6.0 для улучшения читаемости   
  1000000
* в двоичной форме (имеют префикс 0b/0B):  
  >>> 0b110011  
  51;
* в восмеричной форме (имеют префикс 0o/0O):  
  >>> 0o25  
  21;
* в 16-ричной форме (имеют префикс 0x/0X):  
  >>> 0Xa5  
  165.

Во-вторых, целые числа могут быть созданы с помощью конструктора класса – встроенной функции int(x [,base]), которая:

* если аргумент не задан – возращает значение 0;
* если аргумент задан в виде целого числа – возращает его значение (в этом случае проще использовать литерал);
* если аргументом является число с плавающей точкой – оно преобразуется в целое путем отбрасывания дробной части, например:  
    
  >>> int (1.9)  
  1  
  >>> int (-1.1)  
  -1;
* если аргументом является строковый литерал, он должен содержать только цифры и буквы, соответствующие заданной системе счисления. По умолчанию задана десятичная система. С помощью второго необязательного аргумента base можно задать систему счисления по основанию от 2 до 36. При этом строка может содержать цифры от 0 до 9 и латинские буквы от a/A до z/Z:   
    
  >>> int ( 'z' ,36)  
  35.  
    
  Можно также использовать литералы по основанию 2, 8 и 16, указывая их с префиксами 0b/0B, 0o/0O или 0x/0X, так же как числовые литералы задаются в коде программы. В этом случае значение base должно быть 0:  
    
  >>> int ( '0x10' ,0)  
  16;
* если аргумент имеет логический тип (см. [подраздел 4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.2)) – возвращает 1 для значения True и 0 для значения False.

Отметим, что при использовании вместо встроенной функции int() специального метода x.\_\_int\_\_(), x должен быть числом:  
  
>>> 0.026e3.\_\_int\_\_()  
26  
>>> '3' .\_\_int\_\_(),  
  
иначе выводится сообщение об ошибке:  
  
AttributeError: 'str' object has no attribute '\_\_int\_\_';  
  
Для представления целых чисел в виде строк (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2) лаб. раб. №2) можно использовать следующие встроенные функции (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#8)):

* [bin()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#bin) – возвращает целочисленный аргумент в двоичной форме:   
    
  >>> bin (25)  
  '0b11001';
* [oct()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#oct) – возвращает целочисленный аргумент в 8-ричной форме:   
    
  >>> oct (25)  
  '0o31';
* [hex()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#hex) – возвращает целочисленный аргумент в 16-ричной форме:   
    
  >>> hex (25)  
  '0x19'.

Дополнительную информацию о целых числах можно получить, использовав метод int\_info модуля sys (см. [подраздел 1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#1.3) лаб. раб. №2).

**4.1.2 Числа с плавающей точкой**

Числа с плавающей точкой (англ. floating point numbers) или вещественные числа имеют тип (класс) float и содержат целую и дробную части, разделенные точкой. Создаваться числа с плавающей точкой могут, во-первых, в виде литерала, который имеет две формы:

* в виде десятичной дроби:   
    
  >>> 1.2  
  1.2  
    
  Точка ставится даже в тех случаях, когда дробной части нет.  
    
  >>> 1.  
  1.0
* в виде экспоненты {мантисса}e/E{порядок}:   
    
  >>> 1.2e-2  
  0.012

При выводе на экран вещественные числа преобразуются в форму десятичной дроби. Задаваться вещественные числа могут только в десятичной системе счисления.   
Во-вторых, числа с плавающей точкой могут быть созданы с помощью конструктора класса – втроенной функции float(), которая:

* если аргумент не задан, возращает значение 0.0;
* если аргумент задан в виде числа с плавающей точкой, возвращает значение аргумента;
* если аргументом является строка, она преобразуется в число с плавающей точкой:  
    
  >>> float ( '1.25E2' )  
  125.0.

При использовании специального метода x.\_\_float\_\_(), x должен быть числом.  
Дополнительную информацию о вещественных числах можно получить, использовав метод float\_info модуля sys (см. [подраздел 1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#1.3) лаб. раб. №2).

**4.1.3 Комплексные числа**

Комплексные числа (англ. complex numbers) имеют тип (класс) complex и создаются, во-первых, в виде литерала формы a+bj, где a задает вещественную часть числа, а b – мнимую (a и b могут быть числами любого типа, включая пип complex), например:  
  
>>> comp=2+3j+1.33e-1  
>>> comp  
(2.133+3j).  
  
Во-вторых, компрлексное число может быть создано с помощью встроенной фукнции – конструктора complex([real[, imag]]), которая создает комплексное число с реальной частью real и мнимой частью imag. Если первый параметр является строкой, он интерпретируется как комплексное число и функция должна вызываться без второго параметра. Второй параметр не должен быть строкой. Каждый аргумент может быть любого числового типа, включая комплексный. Если мнамая часть опущена, ее значение по умолчанию приравнивается нулю и функция выполняется подобно функциям int() и float(). Если опущены оба аргумента, возвращается значение 0j.  
Отметим, что когда осуществляется преобразование из строки, строка не должна содержать разделительных символов вокруг внутренних операторов + или -. Например, вызов функции complex('4+3j'):   
  
>>> comp1= complex ( '4+3j' )  
>>> comp1  
(4+3j)  
  
создает правильное комплексное число, а вызов функции complex('4 + 3j') приведет к возникновению исключения:  
  
>>> comp1= complex ( '4 + 3j' )  
ValueError: complex() arg is a malformed string,  
  
т.е.аргументом функции complex() является неправильно сформированая строка.  
Класс complex поддерживает свойства:

* real – указывает вещественную часть числа  
    
  >>> comp1.real  
  4.0;
* imag – указывает мнимую часть числа  
    
  >>> comp1.imag  
  3.0

и метод conjugate(), который возвращает сопряженное комплексное число  
  
>>> comp.conjugate()  
(4-3j)

**4.1.4 Операции с числами**

Все числа (за исключением комплексных чисел) поддерживают следующие операции (в скобках указаны соответствующие им специальные методы, если операция имеет два операнда, то приводится также метод с обратным использованием операндов:

* x+y – выполняется сложение чисел x и y, если одно из чисел имеет тип float, то тип результата также будет float  
  (x.\_\_add\_\_(y) и y.\_\_radd\_\_(x)) :  
    
  >>> 2+3.0e-1  
  2.3;  
    
  Отметим, что при использовании специальных методов для целых чисел форма:  
    
  >>> 2.\_\_add\_\_(2+3.0e-1)  
    
  дает синтаксическую ошибку:  
    
  SyntaxError: invalid syntax,  
    
  поскольку фрагмент "2." интерпретируется как число с плавающей точкой. Поэтому значение целого числа необходимо взять в скобки:  
    
  >>> (2).\_\_add\_\_(3.0e-1)  
  NotImplemented;  
    
  В этом случае уже не появляется сообщение об синтаксической ошибке. Однако вместо результата выполненной операции сложения на экран выводится значение объекта NotImplemented. Это говорит о том, что для указанных типов операндов метод .\_\_add\_\_() не применим, поскольку требует, в отличие от операции сложения, чтобы оба операнда были целыми числами:  
    
  >>> (2).\_\_add\_\_(3)  
  5;
* x-y – выполняется вычитание чисел x и y, если одно из чисел имеет тип float, то тип результата также будет float  
  (x.\_\_sub\_\_(y) и y.\_\_rsub\_\_(x)) :  
    
  >>> 5-12.75  
  -7.75;
* x\*y – выполняется умножение чисел x и y, если одно из чисел имеет тип float, то тип результата также будет float  
  (x.\_\_mul\_\_(y) и y.\_\_rmul\_\_(x)) :  
    
  >>> 0.77e1\*3.456  
  26.6112;
* x/y – выполняется деление чисел x и y, возвращается значение с типом float, даже если и делимое, и делитель имеют тип int  
  (x.\_\_truediv\_\_(y) и y.\_\_rtruediv\_\_(x)) :  
    
  >>> 2/3  
  0.6666666666666666,  
    
  как это видно из результата, точность чисел с плавающей точкой равна 15 десятичным знакам в дробной части;  
    
  >>> 12/3  
  4.0;
* x//y – выполняется целочисленное деление x на y, результат – целое число (округление до ближайшего целого и для положительных, и для отрицательных чисел происходит в сторону минус бесконечности  
  (x.\_\_floordiv\_\_(y) и y.\_\_rfloordiv\_\_(x)) :  
    
  >>> 17//4  
  4;  
  >>> -17//4  
  -5;
* x%y – выполняется деление x на y по модулю (определяется остаток от целочисленного деления)  
  (x.\_\_mod\_\_(y) и y.\_\_rmod\_\_(x)):  
    
  >>> 17%4   
  1;
* x\*\*y – выполняется операция возведения x в степень y (результатом операции 0\*\*0 будет 1, как это принято в языках программирования)   
  (x.\_\_pow\_\_(y) и y.\_\_rpow\_\_(x)):  
    
  >>> 2\*\*10  
  1024;  
  >>> 2\*\*(0.5)   
  1.4142135623730951;
* -x – указывается отрицательное значение x  
  (x.\_\_neg\_\_();
* +x – x не меняется  
  (x.\_\_pos\_\_()).

При выполнении каждого оператора вызывается соответствующий ему метод. Это позволяет для одного и того же оператора задавать разные действия в зависимости от типа операндов.  
Отметим, что в выражении операции выполняются в порядке их приоритета: наивысший приоритет имеет операция \*\*, затем следуют операции \*, /, // и %, еще более низкий приоритет имеют операции сложения и вычитания. При равном приоритете операции выполняются слева направо, например:  
  
>>> 5+6/2\*4\*\*2  
53.  
  
Однако операции возведения в степень, идущие подряд, выполняются в обратном порядке – справа налево. Поэтому результатом выражения 2\*\*3\*\*2 будет 512, а не 64:  
  
>>> 2\*\*3\*\*2  
512.  
  
Для изменения порядка выполнения операций в выражении необходимо использовать скобки.

**4.1.5 Функции для работы с числами**

При работе с числами могут быть использованы также следующие встроенные функции (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#8). В скобках указан соответствующий им специальный метод:

* [abs(x)](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#abs) – возвращает абсолютное значение параметра x  
  (x.\_\_abs\_\_()) :  
    
  >>> abs (-7)  
  7,  
    
  для комплексных чисел, заданных как a+bj, функция считается по формуле (a\*\*2+b\*\*2)\*\*(1/2):  
    
  >>> abs (4+3j)  
  5.0;
* [divmod(x,y)](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#divmod) – имеет два параметра: x задает делимое, y – делитель, возвращает кортеж (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab2.htm#2.1) лабораторной работы №2), элементами которого являются соответственно результаты операций // и % над параметрами  
  x.\_\_divmod\_\_(y) :  
    
  >>> divmod (17,3)  
  (5, 2);
* [pow(x,y)](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#pow) – возвращает результат возведения первого параметра в степень, заданную вторым параметром (функция аналогична операции \*\*)  
  (x.\_\_pow\_\_(y) или x.\_\_rpow\_\_(y)):  
    
  >>> pow (3,4)  
  81;  
  >>> (3).\_\_rpow\_\_(4)  
  64;
* [round(x[, n])](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#round) – возвращает число с плавающей точкой, округленное до n цифр после десятичной точки. Если аргумент n опущен или равен 0 – возвращает целое число. Для встроенных типов, поддерживающих функцию round(), значения округляются до числа, ближайшего к 10 в степени минус n. Если два значения одинаково близки, то округление идет в направлении четного числа  
  (x.\_\_round\_\_(n) или x.\_\_rround\_\_(n)):  
    
  >>> round (1.333,2)  
  1.33  
  >>> round (2.5)  
  2  
  >>> round (3.5)  
  4.

Отметим, что поведение функции round() для чисел с плавающей точкой может вызвать удивление: например:  
  
>>> round (2.675, 2)  
  
дает  
  
2.67  
  
вместо ожидаемых 2.68. Это связано с тем обстоятельством, что большинство десятичных дробей не может быть точно представлено в форме числа с плавающей точкой.  
Помимо встроенных функций при работе с целыми числами можно воспользоваться текже следующими методами класса int:

* int.bit\_length() – возвращает число битов, необходимых для представления целого числа в двоичной форме, исключая знак и начальные нули:  
    
  >>> n=2014  
  >>> n.bit\_length()  
  11.

**4.1.6 Побитовые операции над целыми числами**

Python поддерживает также побитовые операции над целыми числами. Пусть операнды a и b имеют следующие значения:  
  
>>> a=60 # 0b00111100  
>>> b=13 # 0b00001101  
  
Для работы с этими операндами могут быть использованы следующие операторы (в скобках указаны соответствующие им специальные методы):

* & – выполняет побитовую операцию И   
  (a.\_\_and\_\_(b) и b.\_\_rand\_\_(a)):  
    
  >>> a&b  
  12 # 0b00001100;
* | – выполняет побитовую операцию ИЛИ  
  (a.\_\_or\_\_(b) и b.\_\_ror\_\_(a)) :  
    
  >>> a|b  
  61 # 0b00111101;
* ^ – выполняет побитовую операцию ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)  
  (a.\_\_xor\_\_(b) и b.\_\_rxor\_\_(a)):  
    
  >>> a^b  
  49 # 0b00110001;
* ~ – выполняет унарную операцию побитового инвертирования  
  (a.\_\_invert\_\_() :  
    
  >>> ~a  
  -61 # 0b110000011;
* >> – выполняет сдвиг левого операнда вправо на число разрядов, заданное правым опрерандом  
  (a.\_\_rshift\_\_(b) и b.\_\_rrshift\_\_(a)) :  
    
  >>> a>>2  
  15 # 0b00001111;
* << – выполняет сдвиг левого операнда влево на число разрядов, заданное правым опрерандом  
  (a.\_\_lshift\_\_(b) и b.\_\_rlshift\_\_(a)) :  
    
  >>> a<<2  
  240 # 0b11110000.

**4.2 Логический тип**

Данные логического типа являются подклассом целых чисел bool(int) и могут принимать одно из двух значений: истина (англ. true) или ложь (англ. false). Язык Python имеют константы True и False, которые используются для непосредственного присваивания логических значений в виде литералов (все встроенные константы языка Python приведены в [подразделе 8.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#8.2)). Другим вариантом задания логических значений является использование конструктора класса – встроенной фукнкции bool([x]) (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#bool)).   
Результатом вычисления выражений также может быть логическое значение. В определенных местах программы, например, в операторе if, интерпретатор языка ожидает, что результатом вычисления выражения будет логическое значение. Такие места программы называют логическим контекстом. Практически всякое выражение можно использовать в логическом контексте. Python в любом случае попытается осуществить его проверку на истинность, используя следующие правила:

* любое число, не равное 0, непустая строка, непустой объект и NotImplemented – принимают значение True;
* числа, равные 0 (0 и 0.0), пустая строка, пустые объекты и значение None – принимают значение False;
* операции сравнения применяются к структурам данных рекурсивно;
* операции сравнения возвращают True или False;
* логические операторы and и or возвращают истинный или ложный объект-операнд:
  + x or y – если x имеет значение False, тогда y, иначе x (в этом случае y не вычисляется);
  + x and y – если x имеет значение False, тогда x (y не вычисляется), иначе y;
  + not x – если x имеет значение False, тогда True, иначе False.

Логические операции приведены в порядке возрастания приоритета:  
  
>>> True and True or False and False  
True  
>>> True and (True or False) and False  
False  
>>> not True or True  
True  
>>> not (False and False)  
False.  
  
Операция not имеет более низкий приоритет, чем любая нелогическая операция, например, not a==b интерпретируется как not (a==b).

**4.3 Проверка типа объекта**

В [подразделе 3.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#3.2) приведено описание и использование функции type(), которая выполняет вывод типа (класса) указанного объекта. Иногда возникает необходимость не выводить значение типа объекта, а определить, принадленжит ли данный объект определенному типу. Для этого можно использовать встроенную функцию isinstance() (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#isinstance) ), которая имеет два аргумента – в первом указывается проверяемый тип (значение), во втором – тип (класс). Функция isinstance() возвращает значение True, если объект имеет указанный тип, и значение False – в противном случае:  
  
>>> isinstance (1.0, int)  
False.  
>>> isinstance ( int (1.0), int)  
True.  
  
Из результатов примера видно, что значение 1.0 не является литералом целого числа, а после произведенного преобразования с помощью конструктора int() – уже имеет целочисленный тип.

**5 Ввод и вывод данных**

В примерах при описании функционирования среды разработки языка Python были использованы встроенные функции print() и input(). Сейчас приведем их более детальное описание.

**5.1 Функция print()**

Функция print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False) осуществляет вывод данных objects в потоковый файл. При этом:

* все неименованные аргументы value, ... преобразуются в строки;
* между ними вставляются строки-разделители, заданные именованым аргументом sep (по умолчанию – пробелы, т.е. sep=' ');
* в конце вывода добавляется строка завершения, заданная именованым аргументом end, (по умолчанию – символ новой строки, т.е. end='\n');
* файл вывода задается с помощью именованого аргумента file (по умолчанию file='sys.stdout'). Он должен иметь метод write(string);
* необходимость буферизации обычно определяется файлом, но если именованый аргумент flush (введенный в версии Python 3.3), равен True, то принудительно выполняется потоковый режим (по умолчанию flush=False).

Все именованные аргументы должны быть строками. Если именнованный аргумент не задан или имеет значение None, принимается значение, указанное для него по умолчанию. Если ни один из неименнованных аргументов не указан, будет осуществлен вывод строки end .   
Пример вывода двух значений без использования ключевых слов sep и end:   
  
x=25   
y=2.5E-2   
print ( 'x' ,x)   
print ( 'y' ,y)   
>>> ================================ RESTART ================================  
x 25  
y 0.025  
>>>

и с их использованием:  
  
x=25   
y=2.5E-2   
print ( 'x' ,x,sep= '=' ,end= ' ' )   
print ( 'y' ,y,sep= '=' ,end= '' )   
>>> ================================ RESTART ================================  
x=25 y=0.025

**5.2 Функция input()**

Функция input([prompt]) – при наличии необязательного аргумента prompt выводит его в стандантный поток вывода (англ. standard output). Затем считывет данные из стандартного устройства ввода (англ. standard intput) и преобразует их в строки, убирая замыкающие символы новой строки (англ. trailing newline).   
Отметим, что функция input() объединила в себе две функции, которые используются во второй версии языка Python – собственно функцию input() и функцию raw\_input(), которая не поддерживается версиями 3.x языка Python.   
В примере показан ввод двоичного целого числа и числа с плавающей точкой:   
  
x= int ( input ( "Введите двоичное число " ),2)  
y= float ( input ( "Введите число с плавающей точкой " ))  
  
Кроме того, осуществлено преобразование введенных значений с помощью соответствующих конструкторов в целый тип и тип с плавающей точкой. Это необходимо делать в случаях, когда введенные значения будут использоваться для выполнения операций с числами, поскольку при вводе все значения преобразуются в строки.

**6 Операторы**

**6.1 Оператор присваивания**

Оператор присваивания имеет следующую форму: v=expr, где

* v – имя переменной;
* expr – выражение, в общем случае представляющее собой совокупность литералов разных типов, переменных, вызовов функций, соединенных между собой знаками операций.

Переменные в языке Python не требуют явного объявления и для них не указывается изначально тип данных. Python относится к языкам программирования с динамической типизацией переменных. Переменные появляются, когда им присваивают значение, и автоматически уничтожаются при выходе из области видимости. Поэтому при работе оператора присваивания переменная получает не только значение выражения, но и его тип.  
Имена (идентификаторы) (англ. names, identifiers) переменных языка Python (), так же как и во многих других языках программирования (Pascal, C, Java, JavaScript и др.) могут содержать только латинские буквы, цифры и знаки подчеркивания и начинаться или с буквы, или со знака подчеркивания. Отметим, что в качестве идентификаторов не следует использовать зарезервированные (ключевые) слова (англ. keywords) языка Python. Список ключевых слов можно получить, например, с помощью встроенной системы помощи (см. [подраздел 1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#1.2)). Имена в языке Python зависят от регистра и не имеют ограничений по числу символов.   
При выполнении оператора присваивания интерпретатором осуществляются следующие действия:

* вычисляется значение выражения, указанного справа от знака равенства, и определяется тип выражения;
* полученное значение заносится в свободные ячейки памяти;
* создается переменная, указанная слева от знака равенства, которая, являясь ссылкой на данные (объект), занесенные в память, получает значение и тип этих данных, например:  
    
  >>> n=7  
  >>> n  
  7  
  >>> type (n)  
  <class 'int'>.

При выполнении оператора del указанная переменная перестает ссылаться на данные в памяти и, таким образом, теряет свое значение:  
  
>>> del n  
>>> n  
NameError: name 'n' is not defined  
  
Данные в памяти, на которые не указывает ни одна ссылка, в последствии очищаются автоматически с помощью системных средств сборки мусора.  
При присваивании переменной значения другой переменной создается еще одна ссылка на существующие данные:  
  
>>> s1= 'some value'  
>>> s2=s1  
>>> s2  
'some value',  
  
и при удалении первой ссылки вторая сохраняет свое значение:  
  
>>> del s1  
>>> s1  
NameError: name 's1' is not defined  
>>> s2  
'some value'.  
  
Python допускает множественное присвоение:  
  
>>> s5=s4=s3=s2  
>>> s5  
'some value'.  
  
Числа и строки в языке Python являются неизменяемыми типами данных. Это означает следующее: для переменных указанных типов нельзя изменить значение данных, хранящихся в памяти, на которые они ссылаются. При присваивании переменной нового значения   
  
>>> n=7  
>>> id (n)  
505894400  
>>> n=25  
>>> id (n)  
505894688.  
  
происходит следующее – прежнее значение не изменится (оно будет в последствии автоматически удалено системой сборки мусора), а новое значение будет занесено в новую область памяти и теперь на него будет указывать ссылка n, что подтверждает встроенная функция id() (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#id)), которая показывает, что значение адреса переменной n стало другим. Т.е. происходит не изменение значения объекта, а создание нового объекта с новыми данными и адресом.  
Оператор присваивания имеет несколько расширенных вариантов, приведенных в табл. 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 1 – Расширенные формы оператора присваивания | | | |
| **Оператор** | **Пример использования** | **Эквивалентная форма** | **Соответствующий метод** |
| += | x+=y | x=x+y | x.\_\_iadd\_\_(y) |
| -= | x-=y | x=x-y | x.\_\_isub\_\_(y) |
| \*= | x\*=y | x=x\*y | x.\_\_imul\_\_(y) |
| /= | x/=y | x=x/y | x.\_\_itruediv\_\_(y) |
| //= | x//=y | x=x//y | x.\_\_ifloordiv\_\_(y) |
| %= | x%=y | x=x%y | x.\_\_imod\_\_(y) |
| \*\*= | x\*\*=y | x=x\*\*y | x.\_\_ipow\_\_(y) |
| <<= | x<<=y | x=x<<y | x.\_\_ilshift\_\_(y) |
| >>= | x>>=y | x=x>>y | x.\_\_irshift\_\_(y) |
| &= | x&=y | x=x&y | x.\_\_iand\_\_(y) |
| |= | x|=y | x=x|y | x.\_\_ior\_\_(y) |
| ^= | x^=y | x=x^y | x.\_\_ixor\_\_(y) |

В языке Python операторы обычно отделяются друг от друга символами перевода на новую строку, т.е. каждый оператор занимает одну строку программы. Мы этого до сих пор придерживались и будет поступать так в дальнейшем. Но синтаксисом языка также разрешается помещать несколько операторов на строке, разделяя их точкой с запятой:   
  
>>> z1=5; z2=3.7; z3=1.5e-1.

Также разрешается располагать оператор на нескольких строках, разделяя отдельные части оператора символами обратного слэша:   
  
>>> x=z1+ \  
z2+ \  
z3 \  
>>> x  
8.85.

**6.2 Условный оператор**

Условный оператор имеет следующую форму:  
  
if <условие> :  
<блок>

В условии указывается выражение, имеющее логический тип. Блок представляет собой одну или несколько строк программы (операторов), которые задаются с отступом. Визуальный блок в языке Python является логическим блоком и соответствует конструкции {...} в JavaScript и других C-подобных языках. Если условие имеет значение True, то выполняются строки, входящие в блок. В противном случае блок не выполняется, а управление передается следующей после условного оператора строке.  
Операторы сравнения, когорые могут использоваться в условии, приведены в табл. 2.

|  |  |  |
| --- | --- | --- |
| Таблица 2 – Операторы сравнения | | |
| **Оператор сравнения** | **Выполняемые действия** | **Метод, соответствующий оператору** |
| x==y | если операнды x и y равны, условие получает значение True, иначе – False | x.\_\_eq\_\_(y) |
| x!=y | если операнды x и y не равны, условие получает значение True, иначе – False | x.\_\_ne\_\_(y) |
| x>y | если значение x больше значения y, условие получает значение True, иначе – False | x.\_\_gt\_\_(y) |
| x>=y | если значение x больше значения y или равно ему, условие получает значение True, иначе – False | (x.\_\_ge\_\_(y)) |
| x<y | если значение x меньше значения y, условие получает значение True, иначе – False | x.\_\_lt\_\_(y) |
| x<=y | если значение x меньше значения y или равно ему, условие получает значение True, иначе – False | x.\_\_le\_\_(y) |

Отметим, что в версиях Python 2.x для сравнения использовались: оператор сравнения <> (эвивалентный оператору !=), встроенная функция cmp() и соответствующий ей метод \_\_cmp\_\_(), которые убраны в версиях 3.x.  
При использовании оператора сравнения вызывается соответствующий ему метод, определенный для типа (класса) сравнивающихся объектов. Поэтому использование оператора сравнения:   
  
>>> x=7  
>>> x==4  
False  
>>> x==7  
True  
  
равноценно вызову требуемого метода:  
  
>>> x.\_\_eq\_\_(4)  
False  
>>> x.\_\_eq\_\_(7)  
True.  
  
Благодаря этому для одного и того же оператора сравнения могут выполняться разные действия в зависимости от типа его операндов.  
Пример использования условного оператора:   
  
>>> x= input ( 'Введите число' )  
Введите число 12  
>>> if int (x)>=0:  
print ( 'Число положительное' )  
  
Число положительное.  
  
После ввода первой строки программы и нажатия Enter на экран выводится сообщение: Введите число. После ввода числа (12) и ввода первой строки условного оператора среда разработки делает отступ для ввода блока. Вводится первая строка блока и дважды нажимается Enter (признаком завершения блока является пустая строка). На экране появляется результат работы этого фрагмента программы: Число положительное.   
Python поддерживает множественное использование операций сравнения:  
  
>>> a,b,c=2,3,4   
>>> if a<b<c: #a<b and b<c   
print ( 'OK' )  
  
OK.  
  
В другой форме условного оператора для задания альтернативного варианта добавляется служебное слово else:  
  
if <условие> :  
<блок1>  
else :  
<блок2>.  
В зависимости от значения условия будет выполняться или блок1 (True), или блок2 (False). Следующий пример запускается в сценарном режиме:  
  
x= input ( 'Введите число' )  
if int (x)>=0:  
print ( 'Число положительное' )  
else :  
print ( 'Число отрицательное' ).  
  
После ввода этого фрагмента программы в окне редактирования выбирается режим Run module и программа запускается на выполнение. Результат работы программы помещается в окно итерактивного режима:  
  
>>> ================================ RESTART ================================  
>>>  
Введите число -1  
Число отрицательное.  
  
В языке Python при задании альтернативных значений вместо фрагмента программы, занимающего четыре строки:   
  
>>> if X:  
A=Y  
>>> else :  
A=Z  
можно применить более короткий вариант, связанный с использованием трехместного выражения if/else:   
  
>>> A=Y if X else Z.  
  
Python поддерживает также третью форму условного оператора, позволяющую задать несколько альтернативных вариантов:  
  
if <условие1> :  
<блок1>  
elif <условие2>:  
<блок2>  
. . .  
elif <условиеN>:  
<блокN>  
else :  
<блокN+1>.  
В качестве примера рассмотрим задачу определения оценки по полученным баллам:  
  
points= input ( 'Введите число баллов' )  
points= int (points)  
if points>80:  
mark= 'Відмінно'  
elif points>66:  
mark= 'Добре'  
elif points>50:  
mark= 'Задовільно'  
else :  
mark= 'Незадовільно'  
print (mark)  
>>> ================================ RESTART ================================  
>>>  
Введите число баллов 80  
Добре.

**6.3 Оператор цикла while**

Оператор цикла while имеет следующую форму:  
  
while <условие> :  
<блок>.

Пока условие имеет логическое значение True, будут циклически выполняться действия, указанные в блоке. Когда условие примет заначение False, управление будет передано оператору, следующему за блоком. Если условие изначально не выполняется, то действия, предусмотренные блоком, не будут выполнены вовсе.  
В качестве примера использования оператора цикла while приведем программу нахождения чисел Фибоначчи. Эти числа характеризуются тем, что каждое следующее число Фибоначчи равно сумме двух предыдущих. Программа находит и выводит на экран все числа Фибоначчи, не превышающие 50:  
  
>>> # Определение чисел Фибоначчи  
>>> a,b=0,1  
>>> while b<=50:  
print (b,end= ' ' )  
a,b=b,a+b  
1 1 2 3 5 8 13 21 34.  
  
В теле оператора while также могут быть использованы операторы break и continue:

* break – выполняет внеочередное завершение цикла;
* continue – выполняет внеочередное завершение текущей итерации:

>>> с=0  
>>> while True :  
с+=1  
if с>9:  
break  
if c==3 or c==5:  
continue  
print (c,end= ' ' )  
1 2 4 6 7 8 9.

**6.4 Оператор pass**

Оператор pass не делает ничего. Он может использоваться, когда синтаксически требуется присутствие оператора, но от программы не требуется действий. Например:  
  
>>> while True :  
pass # Ожидание прерывания c клавиатуры (Ctrl+C).  
  
После нажатия класиши Ctrl+C режим ожидания заканчивается в выводом:  
  
KeyboardInterrupt.  
  
Этот оператор часто используется для создания минималистичных классов, к примеру исключений (exceptions), или для игнорирования нежелательных исключений.

**7 Приоритет операций**

В таблице 3 приведены операции в порядке возрастания их приоритета. Для унарных операций x обозначает операнд.

|  |  |
| --- | --- |
| Таблица 3 – Приоритет операций | |
| **Операция** | **Описание** |
| or | логическое ИЛИ |
| and | логическое И |
| not x | логическое НЕ |
| in, not in | проверка принадлежности |
| is, is not | проверка идентичности |
| <,<=,>,>=, !=,== | операторы сравнения |
| | | побитовое ИЛИ |
| ^ | побитовое исключающее ИЛИ |
| & | побитовое И |
| <<, >> | побитовые сдвиги |
| +, - | сложение и вычитание |
| \*, /, //, % | \*, /, % умножение, деление, целочисленное деление, остаток |
| +x, -x | унарный плюс и смена знака |
| ~x | побитовое НЕ |
| \*\* | возведение в степень |
| x.атрибут | ссылка на атрибут |
| x[индекс] | взятие элемента по индексу |
| x[от:до] | выделение среза (от и до) |
| f(аргумент,...) | вызов функции |
| ( ... ) | скобки или кортеж |
| [ ... ] | список или списковое включение |
| {кл:зн, ...} | словарь пар ключ-значение |

Порядок вычислений операндов определяется такими правилами:

* операнд слева вычисляется раньше операнда справа во всех бинарных операциях, кроме возведения в степень;
* цепочка сравнений вида a < b < c ... y < z фактически равносильна: (а < b) and (b < c) and ... and (y < z);
* перед фактическим выполнением операции вычисляются нужные для нее операнды. В большинстве бинарных операций предварительно вычисляются оба операнда (сначала левый), но операции or и and, а также цепочки сравнений вычисляют такое количество операндов, которое достаточно для получения результата. В невычисленной части выражения в таком случае могут даже быть неопределенные имена. Это важно учитывать, если используются функции с побочными эффектами;
* аргументы функций, выражения для списков, кортежей, словарей и т.п. вычисляются слева-направо, в порядке следования в выражении.

В случае неясности приоритетов желательно применять скобки. Несмотря на то, что одни и те же символы могут использоваться для разных операций, приоритеты операций не меняются.

**8 Встроенные идентификаторы**

**8.1 Встроенные функции**

В предыдущих разделах были описаны и использованы некоторые встроенные функции языка Python. Встроенные (buit-in) означает, что их можно непосредственно использовать в программе без подключения дополнительных модулей. В табл. 4 перечислены в алфавитном порядке имена всех встроенных функций языка Python 3.4. Ниже приводится их описание. Встроенные функции, которые были использованы в лабораторных работах, имеют соответствующие ссылки.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 4 – Встроенные функции языка Python 3.4 | | | | |
| **Built-in functions** | | | | |
| [abs()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#abs) | [dict()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#dict) | [help()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#help) | [min()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#min) | [setattr()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#setattr) |
| [all()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#all) | [dir()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#dir) | [hex()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#hex) | [next()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#next) | [slice()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#slice) |
| [any()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#any) | [divmod()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#divmod) | [id()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#id) | [object()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#object) | [sorted()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#sorted) |
| [ascii()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#ascii) | [enumerate()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#enumerate) | [input()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#input) | [oct()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#oct) | [staticmethod()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#staticmethod) |
| [bin()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#bin) | [eval()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#eval) | [int()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#int) | [open()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#open) | [str()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#str) |
| [bool()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#bool) | [exec()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#exec) | [isinstance()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#isinstance) | [ord()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#ord) | [sum()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#sum) |
| [bytearray()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#bytearray) | [filter()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#filter) | [issubclass()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#issubclass) | [pow()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#pow) | [super()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#super) |
| [bytes()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#bytes) | [float()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#float) | [iter()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#iter) | [print()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#print) | [tuple()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#tuple) |
| [callable()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#callable) | [format()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#format) | [len()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#len) | [property()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#property) | [type()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#type) |
| [chr()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#chr) | [frozenset()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#frozenset) | [list()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#list) | [range()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#range) | [vars()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#vars) |
| [classmethod()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#classmethod) | [getattr()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#getattr) | [locals()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#locals) | [repr()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#repr2) | [zip()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#zip) |
| [compile()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#compile) | [globals()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#globals) | [map()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#map) | [reversed()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#reversed) | [\_\_import\_\_()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#import) |
| [complex()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#complex) | [hasattr()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#hasattr) | [max()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#max) | [round()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#round) |  |
| [delattr()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#delattr) | [hash()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#hash) | [memoryview()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#memoryview) | [set()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#set) |  |

* **abs(x)** – возвращает абсолютное значение числа (см. [подраздел 4.1.5](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.5) );
* **all(iterable)** – возвращает значение True, если все значения итерабельного объекта имеют значение True или итерабельный объект пуст (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2.1) лаб. раб. №3);
* **any(iterable)** – возвращает значение True, если хотя бы одно значение итерабельного объекта равно True. Если итерабельный объект пуст, возвращает False (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2.1) лаб. раб. №3);
* **ascii(object)** – так же, как функция repr(), возвращает строку, содержащую печатаемое представление объекта, но в отличие от repr() не возвращает изображения не ASCII символов, представляя их с помощью ESC-последовательностей \x, \u или \U:  
    
  >>> repr ( 'ы' )  
  "'ы'"  
  >>> ascii ( 'ы' )  
  "'\\u044b'";
* **bin(x)** – преобразует целое число в двоичную строку. Если x не является целым числом, т.е. не является объектом типа int, то для него должен быть определен метод \_\_index\_\_() для преобразования в int (см. [подраздел 4.1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1));
* **bool([x])** – преобразует значение к логическому типу, используя стандартную процедуру определения истинности (см. [подраздел 4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.2) лаб. раб. №1). Если x ложно или опущено, – возвращает False, в остальных случаях возвращает True;
* **bytearray([source[, encoding[, errors]]])** – возвращает новый объект типа bytearray (см. [подраздел 3.2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.2) лаб. раб. №4);
* **bytes([source[, encoding[, errors]]])** – возвращает новый объект типа bytes (см. [подраздел 3.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.1) лаб. раб. №4);
* **callable(object)** – возвращает True, если аргумент можно вызвать по имени, и False – если нет (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.1) лаб. раб. №6). Если возвращается значение True, остается возможность неудачного вызова, но если возвращается False, вызов никогда не будет успешным. Отметим, что экземпляры классов являются вызываемые (англ. callable), если их классы имеют метод \_\_call\_\_();
* **chr(code)** – возвращает строку из одного символа, который соответствует числовому значению аргумента, заданному в ASCII или Unicode коде (см. [подраздел 2.5](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2.5) лаб. раб. №2). Например, chr(97) возвратит строку 'a'. Зта функция является обратной по отношению к функции ord(). (см. [подраздел 2.5](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2.5) лаб. раб.ж №2). Допустимый диапазон значений составляет от 0 др 1114111 (0x10FFFF). При превышении диапазона будет вызвано исключение ValueError;
* **classmethod(function)** – возвращает для функции, указанной аргументом, метод класса;
* **compile(source, filename, mode, flags=0, dont\_inherit=False, optimize=-1)** – компилирует исходный код, указанный аргументом source, в код или объект AST (см. [подрездел 4.1](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#4.1) лаб. раб. №6). Объекты кода могут быть выполнены с помощью функций [exec()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#exec) или [eval()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#eval);
* **complex([real[, imag]])** – создает комплексное число (см. [подраздел 4.1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.3));
* **delattr(object, name)** аргументами являются объект и строка. Строка должна быть именем одного из атрибутов объекта. Функция удаляет указанный атрибут объекта, например, delattr(x, 'foobar') эквивалентно del x.foobar;
* **dict(\*\*kwarg), dict(mapping, \*\*kwarg)**   
  **dict(iterable, \*\*kwarg)** – создаeт новый словарь (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1) лаб. раб. №5);
* **dir([object])** – если оргумент не указан, возвращает список имен текущей области видимости. Если аргумент указан – возвращает список имен (атрибутов) объекта (см. [подраздел 1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#1.2) лаб. раб. №4 и [подраздел 2.4](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#2.4) лаб. раб. №6);
* **divmod(a, b)** – возращает частное и остаток от деления двух не комелексных числа (см. [подраздел 4.1.5](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.5));
* **enumerate(iterable, start=0)** – возвращает пронумерованный (англ. enumerate) объект. В качестве аргумента iterable может быть указаны последоваетльность, итерабельный объект или любой другой объект, поддерживающий итерацию. Например, если первым аргументом функции enumerate() укажем список (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2) лаб. раб. №3) времен года, а нумерацию начнем с 1 (необязательный второй аргумент), то функция возвратит пронумерованный список времен года в виде кортежей (см. [раздел 1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#1) лаб. раб. №4):  
    
  >>> seasons=[ 'зима' , 'весна' , 'лето' , 'осень' , ]  
  >>> list ( enumerate (seasons, start=1))  
  [(1, 'зима'), (2, 'весна'), (3, 'лето'), (4, 'осень')];
* **eval(expression, globals=None, locals=None)** – осуществляет грамматический разбор выражения и вычисляет его значение, аргументами являются: строка expression, задающая выражение, и необязательные словари globals и locals, указывающие пространства имен соответственно глобальных и локальных переменных (см. [подрездел 4.3](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#4.3) лаб. раб. №6) ;
* **exec(object[, globals[, locals]])** – эта функция поддерживает динамическое выполнение кода языка Python. Аргумент object должен быть строкой или объектом типа code (см. [подраздел 4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#4.2) лаб. раб. №6);
* **filter(function, iterable)** – возвращает итератор, созданный из элементов итерабельного типа, заданных аргументом iterable, путем их фильтрации с помощью функции function (см. [подраздел 1.5.1](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.5) лаб. раб. №6);
* **float([x])** – преобразует строку или число в число с плавающей точкой (см. [подраздел 4.1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.2));
* **format(value[, format\_spec])** – преобразует значение value в форматированное представление, управляемое спецификацией формата (format\_spec). Интерпретация формата зависит от типа значения. Однако имеется стандартный синтаксис форматирования, который может быть использован большинством встроенных типов данных;
* **frozenset([iterable])** – возвращает неизменяемое множество, аргументом является итерабельный объект (см. [раздел 1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1) лаб. раб. №5);
* **getattr(object, name[, default])** – возвращает значение атрибута объекта object, имеющего имя, заданное строкой name. getattr(x, 'foobar') эквивалентно x.foobar. Если имя атрибута не указано, возвращается значение по умолчанию (default). Если и оно не указано, вырабатывается исключение AttributeError;
* **globals()** – возращается словарь, представляющий таблицу текущих глобальных имен. Это всегда словарь текущего модуля, т.е.это модуль, который определен внутри функции или метода, не модуль, из которого были вызваны функция или метод (см. подразделы [2.4](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#2.4), [4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#4.2) и [4.3](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#4.3) лаб. раб. №6);
* **hasattr(object, name)** – возвращает значение True, если строка name, является именем одного из атрибутов объекта object, и False – если нет (реализована путем вызова функции getattr(object, name)) и анализа, вызовет ли это исключение AttributeError или нет;
* **hash(object)** – возвращает хэш-значение (англ. hash value) объекта (если он его имеет). Хэш-значения являются целыми числами и используются для более быстрого сравнения ключей словаря при его просмотре (см. [подраздел 1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.3) лаб. раб. №3);
* **help([object])** – вызывает встроенную систему помощи (см. [подраздел 1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#1.2));
* **hex(x)** – преобразует целое число в шестнадцатиричную строку. Если оргумент x не является объектом класса int, то должен быть определен метод \_\_index\_\_(), возвращающий целое значение. Отметим, что для получения шестнадцатиричной строки, представляющей число с плавающей точкой, необходимо воспользоваться методом float.hex() (см. [подраздел 4.1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.1));
* **id(x)** – возвращает “идентичность” (англ. identity) объекта в виде числа, которое является уникальным и постоянным в течение времени его жизни. Два объекта с непересекающимися временами жизни могут иметь одинаковые значения идентичности. Отметим, что при реализации CPython идентичность представляет собой адрес объекта в памяти (см. подразделы [3.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#3.1) и [6.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#6));
* **input([prompt])** – при наличии необязательного аргумента prompt выводит его в стандантный поток вывода. Затем функция считывет данные из стандартного устройства ввода, преобразует их в строки, убирая замыкающие символы новой строки (см. [подраздел 5.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#5.2));
* **int(x=0)**  
  **int(x, base=10)** – преобразует x в целое число или возвращает 0, если аргумент не был задан (см. [подраздел 4.1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.1) лаб. раб. №1);
* **isinstance(object, classinfo)** – возвращает True, если объект, указанный аргументом object является экземпляром класса, заданного аргументом classinfo, или его подклассом (прямым, непрямым или виртуальным), и False в противном случае. Если object не является объектом указанного типа, то функция всегда возвращает False. Если аргумент classinfo не является классом (типом объекта), он может быть кортежом типов объектов, или может рекурсивно содержать другие такие кортежи (другие последователности типов не допускаются). Если же classinfo не является ни типом, ни кортежом типов или кортежей, то вырабатывается исключение TypeError (см. использование функции в подразделах [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.1), [1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.2) и [1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.3) лаб. раб. №3, в разделе [2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#2), в подразделах [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#1.1) и [3.3](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.3) лаб. раб. №4, в разделе [1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1), в подразделах [2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1) и [3.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#3.1) лаб. раб. №5);
* **issubclass(class, classinfo)** – возвращает значение True, есои class является подклассом (прямым, непрямым или виртуальным) аргумента classinfo. Класс считается подвклассом самого себя. Аргумент classinfo может быть кортежем объектов-классов, в котором каждый элемент должен быть проверен. В случае неудачи возникает исключение TypeError;
* **iter(object[, sentinel])** – возвращает объект-итератор. Первый аргумент интерпретируется по разному в зависимости от присутствия второго агрумента.   
  Без второго аргумента объект может быть или объектом-контейнером с поддержкой протокола итерации (метода \_\_iter\_\_()), или он должен поддерживать протокол последовательности (метод \_\_getitem\_\_() с целочисленным аргументом, начинающимся с 0. Если он не поддерживает ни один из этих протоколов, возникает ошибка типа (TypeError).  
  Если второй аргумент sentinel, задающий граничное значение, указан, тогда объект должен быть вызываемым объектом (англ. callable object). Итератор, созданный в этом случае будет вызывать объект без аргументов для каждого вызова своего метода \_\_next\_\_(). Если возвращаемое значение равно значению аргумента sentinel, будет вызвано исключение StopIteration. В остальных случаях будет возвращено значение (см. [подраздел 1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.2) лаб. раб. №3);
* **len(s)** – возвращает длину (число элементов) объекта. Аргументом может быть последовательность (см. [подраздел 2.5](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2.5) лаб. раб. №2, подразделы [1.4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.4.2) и [2.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2.2) лаб. раб. №3, [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#2) и подразделы [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#1.1), [3.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.1) и [3.2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.2) лаб. раб. №4), множество (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.1) лаб. раб. №5) и словать (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1) лаб. раб. №5);
* **list([iterable])** – возвращает значение в виде списка (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2) лаб. раб. №3);
* **locals()** – изменяет и возвращает словарь, представляющий таблицу текущих локальных имен. Свободные переменные возвращаются функцией locals(), когда она вызывается в блоке функции, но не в блоке класса. Отметим, что содержимое словаря не должно модифицироваться, изменения могут не повлиять на значения локальных и свободных переменныоъ, используемых интерпретатором (см. подразделы [2.4](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#2.4), [4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#4.2) и [4.3](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#4.3) лаб. раб. №6);
* **map(function, iterable)** – возвращает итератор, сформированный из элементов iterable с помощью функции function (см. [подраздел 1.5.2](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.5.2) лаб. раб. №6);
* **max(iterable, \*[, key, default])**   
  **max(arg1, arg2, \*args[, key])** – возвращает самый большой элемент итерабельного типа или наибольший из двух или более аргументов. Если используется один позиционный оргумент, он должен быть инерабельным объектом. Возвращается его наибольший элемент (см. [подраздел 2.5](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2.5) лаб. раб. №2, подразделы [1.4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.4.2) и [2.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2.2) лаб. раб. №3, [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#2) и подразделы [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#1.1), [3.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.1) и [3.2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.2) лаб. раб. №4), множество (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.1) лаб. раб. №5) и словарь (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1) лаб. раб. №5). Если указываются два или больше позиционных аргументов, возвращается наибольший из них. Функция может иметь два необязательных именных аргумента. Аргумент key указывает функцию упорядочевания с одним аргументом, подобную тем, которые используются для метода list.sort() (см. [подраздел 1.4](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.4) лаб. раб. №6). Аргумент по умолчанию – объект, который будет возвращен, если итерабельный объект будет пустым. Если итерабельный объект будет пустым и аргумент по умолчанию не будет указан, возникнет исключение ValueError. Если окажется несколько максимальных значений, функция возвратит, первое встретившееся. Эти правила распространяются и на другие программные средства сортировки, такие как sorted(iterable, key=keyfunc, reverse=True)[0] и heapq.nlargest(1, iterable, key=keyfunc). Новое в версии 3.4 – именованный аргумент, задающий значение по умолчанию;
* **memoryview(object)** – преобразует объект object в объект класса memoryview, поддерживает протокол буфера (англ. buffer protocol);
* **min(iterable, \*[, key, default])**   
  **min(arg1, arg2, \*args[, key])** – возвращает самый маленький элемент итерабельного типа или наименьший из двух или более аргументов. Если используется один позиционный оргумент, оy должен быть инерабельным объектом. Возвращается его наименьший элемент (см. [подраздел 2.5](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2.5) лаб. раб. №2, подразделы [1.4.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.4.2) и [2.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2.2) лаб. раб. №3, [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#2) и подразделы [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#1.1), [3.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.1) и [3.2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#3.2) лаб. раб. №4), множество (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.1) лаб. раб. №5) и словарь (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1) лаб. раб. №5). Если указываются два или больше позиционных аргументов, возвращается наименьший из них. Функция может иметь два необязательных именных аргумента. Аргумент key указывает функцию упорядочевания с одним аргументом, подобную тем, которые используются для метода list.sort() (см. [подраздел 1.4](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.4) лаб. раб. №6). Аргумент по умолчанию – объект, который будет возвращен, если итерабельный объект будет пустым. Если итерабельный объект будет пустым и аргумент по умолчанию не будет указан, возникнет исключение ValueError. Если окажется несколько миниимальных значений, функция возвратит, первое встретившееся. Эти правила распространяются и на другие программные средства сортировки, такие как sorted(iterable, key=keyfunc, reverse=True)[0] и heapq.nlargest(1, iterable, key=keyfunc). Новое в версии 3.4 – именованный аргумент, задающий значение по умолчанию;
* **next(iterator[, default])** – возвращает следующий элемент из итератора путем вызова его метода \_\_next\_\_(). Если указано значение по умолчанию, оно возращается когда нет следующего элемента, в остальных случаях вызывается исключение StopIteration (см. [подраздел 1.2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.2) лаб. раб. №3);
* **object()** – возвращает новый объект без особенностей (featureless object). Объект object является базовым для всех классов (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB11/lab.htm#1.1) лаб. раб. №11). Имеет методы, общие для экземпляров всех классов языка Python. Эта функция не принимает никаких аргументов. Отметим, что object не имеет метода \_\_dict\_\_, так что нельзя присваивать произвольные атрибуты экземплярам класса object;
* **oct(x)** – преобразует целое число в восьмеричную строку. Результатом является правильное выражение языка Python. Если x не является объектом класса int, т.е. целым числом, он должен иметь метод \_\_index\_\_() для преобразания значения к целочисленному типу (см. [подраздел 4.1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.1));
* **open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)** – открывает файл и возвращает соответствующий файловый объект (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB8/lab.htm#1.1) лаб. раб. №8);
* **ord(c)** – аргументом указывается строка, представляющая один символ в кодировке Unicode, возвращает целое число, являющееся кодом этого символа. Например, ord('a') возвращает число 97 и ord('\u2020') возвращает 8224. Эта функция является обратной функции chr() (см. [подраздел 2.5](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2.5) лаб. раб. №2);
* **pow(x, y[, z])** – возвращает x в степени y. Если указано z, возвращает x в степени y по модулю z (вычисляется более эффективно чем pow(x, y) % z). Форма с двумя аргументами эквивалентна использованию оператора x\*\*y. Аргументы должны быть числами. При использовании чисел разных типов применяются правила приведения типов. Для целых чисел результат имеет тот же тип, что и оргументы (после приведения). Но это выполняется только если второй аргумент не является отрицательным числом. В этом случае все аргументы преобразуются в вещественные числа (числа с плавающей точкой) и результат будет иметь такой же тип. Например, 10\*\*2 возвращает 100, но 10\*\*-2 возращает 0.01. Если второй аргумент отрицателен – третьего аргумента не должно быть. Если z присутствует, x и y должны быть целыми числами и y не должнен быть отрицательным числом (см. [подраздел 4.1.5](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.5));
* **print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)** – осуществляет вывод данных objects в потоковый файл (см. [подраздел 5.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#5.1));
* **property(fget=None, fset=None, fdel=None, doc=None)** – возвращает атрибут property;
* **range(stop)range(start, stop[, step])** – возвращает виртуальную последовательность чисел от start до stop - 1 с шагом step (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#2) лаб. раб. №4);
* **repr(object)** – возвращает строку, в большинстве случаев содержащую представление объекта, предназначенное для работы интерпретатора (см. [подраздел 2.5](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2.5) лаб. раб. №2);
* **reversed(seq)** – возвращает обратный итератор. Аргумент seq должен быть объектом, имеющим метод \_\_reversed\_\_() или поддерживающим протокол последовательности (меотоды \_\_len\_\_() и \_\_getitem\_\_() с целочисленным аргументом, начинающимся с 0;
* **round(number[, ndigits])** – возвращает число с плавающей точкой, округленное до ndigits цифр после десятичной точки (см. [подраздел 4.1.5](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4.1.5));
* **set([iterable])** – возвращает изменяемое множество, аргументом является итерабельный объект (см. [раздел 1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1) лаб. раб. №5);
* **setattr(object, name, value)** – выполняет действия, противоположные функции getattr(). Атрибутами являются объект object, строка name и произвольное значение value. Строка может быть или именем существующего атрибута, или новым атрибутом. Функция присваивает значение атрибуту, проверяя, позволяет ли объект сделать это. Например, setattr(x, 'foobar', 123) эквивалентно x.foobar = 123;
* **slice(stop)**  
  **slice(start, stop[, step])** – возвращает объект типа slice, представленный индексами, указанными диапазоном range(start, stop, step). Аргументы start и step по умолчанию имеют значение None. slice-объекты имеют данные только для чтения, атрибуты start, stop и step просто вовзращают значения (или значения по умолчанию). Эти объекты не имеют явной функциональности. Но могут быть использованы в Numerical Python и других расширениях;
* **sorted(iterable[, key][, reverse])** – возвращает новый отсортированный список из элементов аргумента iterable. Имеет два необязательных оргумента, которые должны быть указаны как именованные. Аргумент key указывает функцию одного аргумента, которая используепся для получения сравниваемого элемента key из каждого элемепнта списка: key=str.lower. Значение по умолчанию – None (прямое сравнение элементов). Аргумент reverse имеет логическое значение. Если он имеет значение True – элементы списка сортируются, как будто каждое сравнение является обратным (см. [подраздел 1.4](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.4) лаб. раб. №6);
* **staticmethod(function)** – возвращает статический метод для функции. Для объявления статического метода необходимо использовать декоратор @staticmethod (см. [2.2](https://py-khpi.github.io/!nfo/doc/lab/LAB11/lab.htm#2.2) лаб. раб. №11):   
    
  >>> class C :  
  @ staticmethod  
  def f (arg1, arg2, ...)
* **str(object='')**  
  **str(object[, encoding[, errors]])** – возвращает представление заданного объекта в виде строки (см. [подраздел 2.5](https://py-khpi.github.io/!nfo/doc/lab/LAB2/lab.htm#2.5) лаб. раб. №2);
* **sum(iterable[, start])** – возвращает сумму элементов итерабельного объекта iterable, полученную в результате их суммирования слева направо. Значение необязательного аргумента start, задающего начальное значение суммы, по умолчанию равно 0. Элементы итерабельного объекта обычно являются числами. Найдем сумму чисел, заданных кортежем:  
    
  >>> sum ((2.5, 4,1.25e1))  
  19.0  
    
  Для некоторых классов есть неплохие альтернативы функции sum(): для конкатенации строк можно использовать join(sequence), для сложения чисел с плавающей точкой повышенной точности – math.fsum(), для конкатенации нескольких итерабельных объектов – itertools.chain();
* **super([type[, object-or-type]])** – возвращает прокси объект (англ. proxy object), который делегирует запросы метода базовому классу (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB12/lab.htm#2) лаб. раб. №12);
* **tuple([iterable])** – возвращает кортеж (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB4/lab.htm#1.1) лаб. раб. №4);
* **type(object)**   
  **type(name, bases, dict)** – при задании одного аргумента object функция возвращает наименование типа объекта (см. подразделы [3.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#3.2) , [2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2.1) лаб. раб. №3, [2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1) и [3.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#3.1) лаб. раб. №5, [4.1](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#4.1) лаб. раб. №6, [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB11/lab.htm#1.1) лаб. раб. №11. Для проверки типа объекта рекомендуется использовать встроенную функцию [isinstance()](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#isinstance).   
  При задании трех аргументах функция type(name, bases, dict) возвращает новый класс. Это существенно для динамической формы оператора класса. Строка name указывает имя класса и становится атрибутом \_\_name\_\_, кортеж bases перечисляет базовые классы и становится атрибутом \_\_bases\_\_ и словарь dict задает адресное пространство атрибутов класса и становится атрибутом \_\_dict\_\_ (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB11/lab.htm#1.1) лаб. раб. №11);
* **vars([object])** – возвращает значение атрибута \_\_dict\_\_ для модуля, класса, экземпляра класса или других объектов, имеющих атрибут \_\_dict\_\_. Такие объекты как модули, экземпляры классов имеют изменяемые атрибуты \_\_dict\_\_, в то время как другие объекты могут иметь ограничения на изменение значения (например, классы используют dictproxy для предотвращения изменения словаря). Без аргумента функция vars() действует так же как функция locals(). Отметим, что словарь locals полезен только для чтения, поскольку изменения в locals dictionary игнорируются (см. [2.4](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#2.4) лаб. раб. №6);
* **zip(\*iterable)** – возвращает итератор, элементами которого являются кортежи, где i-й кортеж содержит i-е элементы каждого аргумента последовательностей или итераторов iterable (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1) лаб. раб. №5);
* **\_\_import\_\_(name, globals=None, locals=None, fromlist=(), level=0)** – это функция с расширенными функциональными возможностями не для ежедневноого программирования. Вызывается при выполнении оператора import, который может быть заменен ею с целью изменения семантики оператора. В других случаях делать это нежелательно.  
  Функция \_\_import\_\_() загружает (импортирует) модуль с именем name, при этом необязательные аргументы globals и locals определяют, как интерпретировать имя name в контексте пакета (англ. package). Аргумент fromlist укахывает имена объектов или подмодулей, которые должны быть импортированы из модуля name. Стандартное применение не использует аргумент locals вообще, а globals только для определения пакета в контексте оператора import. Аргумент level указывает, нужно ли использовать абсолютное или относительное импортирование. Значение 0 (задается по умолчанию) означает выполнение только абсолютного импортирования. Положительное значение этого аргумента показывает число родительских каталогов для поиска относительно каталога модуля, вызывающего \_\_import\_\_(). Пример использования:  
    
  >>> temp=\_\_import\_\_( "collections" , fromlist=[ "Iterable" , "Sequence" ])  
  >>> isinstance ( "WWW" , temp.Sequence)  
  True  
  >>> isinstance (2,temp.Iterable)  
  False

**8.2 Встроенные константы**

Python имеет следующие встроенные константы:

* **False** – ложное значение логического типа bool (см. [подраздел 6.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#6.2));
* **True** – истинное значение логического типа bool (см. [подраздел 6.2](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#6.2));
* **None –** значение типа NoneType. None часто используется для представления отсутствия значения, например, когда значение по умолчанию не передается в функцию (см. [раздел 4](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4));
* **NotImplemented** – специальное значение, которое может быть возвращено при сравнении с использованием методов \_\_eq\_\_(), \_\_lt\_\_() и им подобных для указания того, что сравнение не может быть применено, учитывая тип данных второго операнда (см. [раздел 4](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#4));
* **Ellipsis –** то же самое, что и **. . .**. Специальное значение, используемое в основном в срезах с расширенным синтаксисом для определенных пользователем типов данных-контейнеров. Вне среза:  
  >>> ...  
  Ellipsis;
* **\_\_debug\_\_** – константа принимает истинное значение, если Python начал работу не по опции **-O** в командной строке:  
  >>> \_\_debug\_\_  
  True