Переменные.

Практически любая программа оперирует с некоторыми данными. Это может быть·как реально большая база данных, так и одно-единственное значение - принципиальной разницы здесь нет. Важно то, что данные в программе должны как-то сохраняться. Мы пока что будем "сохранять" данные с помощью переменных. В отношении переменных важны два момента:

1. *у переменной есть имя*
2. *переменная ссылается на некоторое значение*

В Python тип переменной явно не указывается, он определяется по значению на которое ссылается переменная.

Название переменных может быть практически любым (комбинация букв, цифр, символов подчеркивания), например:

*dlinna\_okruzhnosti = 24*

*a127 = 14*

*ssd = 21*

Важно чтобы название переменной не совпадало со списком ключевых слов:

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

Вообще тип данных важен, по меньшей мере, с двух позиций: тип данных определяет объем памяти, выделяемый для хранения этих данных, а также находится в тесной связи с теми операциями, которые допустимы с данными. Во многих языках программирования переменные объявляется с указанием типа. Вот в этом случае переменную удобно представлять как корзину, на которой· написано название переменной, а значение переменной - то, что внутри корзины. Таких корзин в программе может быть много. Каждая корзина отождествляется с какой-то конкретной переменной, в каждой корзине что-то "лежит" (значение соответствующей переменной). Размер корзины определяется типом переменной, с которой мы эту корзину отождествляем. Когда мы считываем значение переменной, то просто "смотрим", что лежит в корзине. Когда меняем значение переменной, то вынимаем из корзины то, что в ней было, и помещаем туда новое содержимое.

В Python все несколько иначе. Переменные не имеют типа. Но тип есть у данных, которые "запоминаются" с помощью переменных. Поэтому концепция обычных корзин теряет свою актуальность. Актуальной становится концепция корзин, к которым привязаны веревочки. Переменная - это веревочка с биркой (название переменной) , а корзина - это данные , на которые ссылается переменная. То есть те ­перь название переменной - это не бирка на корзине, а бирка на веревке. На корзинах бирок нет. Как и ранее, размер корзины определяется характером ее содержимого (тип данных). Но доступ к корзине у нас есть только через веревочку. Веревочки все одинаковые и отличаются лишь названием на бирке. Что мы можем сделать в такой ситуации: мы можем потянуть за веревочку и посмотреть, что находится в корзине, к которой привязана веревочка. Это аналог считывания значения переменной. Также мы можем поменять содержимое корзины или вообще отвязать от веревочки корзину и привязать веревочку к другой корзине. Это аналог изменения значения переменной. Более того, мы можем привязать несколько веревочек к одной и той же корзине. На языке программирования это означает, что несколько переменных ссылаются на одно и то же значение. И такое в Python тоже возможно.

Основные операторы языка Python.

Операторы являются конструкциями, которые могут манипулировать значением операндов. Рассмотрим выражение 4 + 5 = 9. Здесь, 4 и 5 называются операндами и + называется оператором.

Типы операторов:

**Язык Python** поддерживает следующие типы операторов:

* Арифметические операторы
* Сравнение (реляционные) операторы
* Операторы присваивания

**Арифметические операторы в Python**

Предположим, переменная содержит **а** значение 10, а переменная **b** содержит значение 21, то:

|  |  |  |
| --- | --- | --- |
| **оператор** | **Описание** | **пример** |
| + Сложение | Добавление значений по обе стороны от оператора. | а + b = 31 |
| – Вычитание | Вычитание правого операнда из левого операнда. | а – b = -11 |
| \* Умножение | Умножение значения по обе стороны от оператора | a \* b = 210 |
| / Деление | Делит левый операнд на правый операнд | b / а = 2,1 |
| % Модуль | Делит левый операнд на правый операнд и возвращает остаток | b % а = 1 |
| \*\* | Выполняет возведение в степень | а \*\* b = 10 в степени 20 |
| // деление с округлением | После деления оставляется только целая часть числа | 9 // 2 = 4 и 9,0 // 2,0 = 4,0  -11 // 3 = -4 -11,0 // 3 = -4,0 |

**Операторы сравнения в Python**

Операторы сравнения значений по обе стороны от них и решить, соотношение между ними. Их также называют реляционные операторы.

Предположим, переменная **а** содержит значение 10, а переменная **b** имеет значение 20, то:

|  |  |  |
| --- | --- | --- |
| **оператор** | **Описание** | **пример** |
| == | Если значения двух операндов равны, то условие становится истинным. | (а == b) не верно. |
| != | Если значения двух операндов не равны, то условие становится истинным. | (а ! = b) истинно. |
| > | Если значение левого операнда больше значения правого операнда, то условие становится истинным. | (а > b) не верно. |
| < | Если значение левого операнда меньше значения правого операнда, то условие становится истинным. | (а < b) истинно. |
| > = | Если значение левого операнда больше или равно значению правого операнда, то условие становится истинным. | (а >= b) не верно. |
| <= | Если значение левого операнда меньше или равно значению правого операнда, то условие становится истинным. | (а <= b) истинно. |

**Операторы присваивания в Python**

Предположим, переменная содержит **а** значение 10, а переменная **b** имеет значение 20, то:

|  |  |  |
| --- | --- | --- |
| **оператор** | **Описание** | **пример** |
| = | Назначает значения с правой стороной операндов левой стороне операнда | с = а + b присваивает значение a + b в c |
| + = Добавить и | Добавляет правый операнд к левому операнду и присвоить результат левого операнда | с + = а эквивалентно c = c + a |
| – = вычесть и | Вычитает правый операнд из левого операнда и присваивает результат левого операнда | с – = а эквивалентно c = c – a |
| \* = умножить и | Умножает правый операнд на левый операнд и присваивает результат левого операнда | с \* = а эквивалентно c = c \* a |
| / = разделить и | Делит левый операнд на правый операнд и присваивает результат левого операнда | с / = а эквивалентно c = c / ac /= a эквивалентно с = с / а |
| % = Модуль и | Принимает модуль с помощью двух операндов и присваивает результат левого операнда | c% = а эквивалентно c = c % a |
| \*\* = Экспонент и | Выполняет вычисление экспоненту от операторов и присваивает значение левого операнда | с \*\* = а эквивалентно c = c \*\* a |
| // = Floor Division | Выполняет деление операторов с округлением и присваивает значение левого операнда | с // = а эквивалентно c = c // a |

Ветвление. Условный оператор

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

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

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

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

if логическое\_выражение {

выражение 1;

выражение 2;

…

}

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

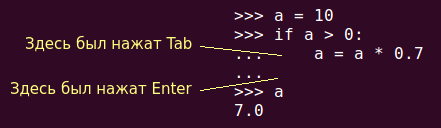
Конструкция if логическое\_выражение называется **заголовком условного оператора**. Выражения внутри фигурных скобок – **телом условного оператора**. Тело может содержать как множество выражений, так и всего одно или даже быть пустым.

Пример использования условного оператора в языке программирования Python:

**if** n < 100:

b = n + a

В Питоне вместо фигурных скобок используется двоеточие. Обособление вложенного кода, т. е. тела оператора, достигается за счет отступов. В программировании принято делать отступ равным четырем пробелам. Однако также можно использовать клавишу табуляции (Tab) на клавиатуре. Большинство сред программирования автоматически делают отступ, как только вы поставите двоеточие и перейдете на новую строку. Однако при работе в интерактивном режиме надо делать отступы вручную.



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

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

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

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

b = 0

a = 50

n = 98

**if** n < 100:

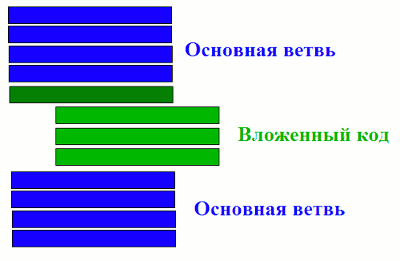
b = n + a

**print**(b)

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

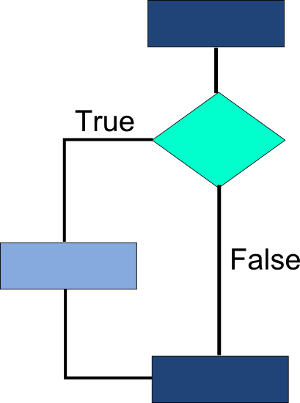
Поскольку переменная n равна 98, а это меньше 100, то b станет равной 148-ми. Это значение будет выведено на экран. Если переменная n изначально была бы связана, например, со значением 101, то на экран был бы выведен 0. При n, равной 101, логическое выражение в заголовке условного оператора вернуло бы ложь. Значит, тело не было бы выполнено, и переменная b не изменилась бы.

Структуру программы можно изобразить следующим образом:

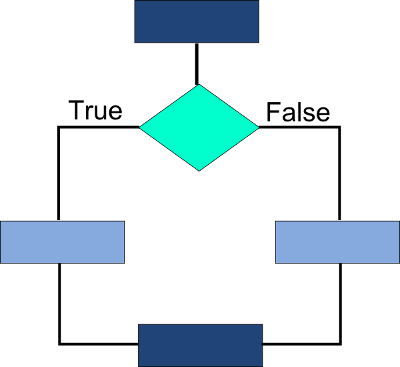


Основная ветка программы выполняется всегда, а вложенный код лишь тогда, когда в темно-зеленой строчке, обозначающей заголовок условного оператора, случается истина.

Для небольших программ иногда чертят так называемые блок-схемы, отражающие алгоритм выполнения. В языке блок-схем определенные конструкции обозначаются своими фигурами. Так блок действий обозначается прямоугольником, а логическое выражение – ромбом. Для кода выше блок-схема может выглядеть так:



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



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

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

if логическое\_выражение {

выражение 1;

выражение 2;

…

}

else {

выражение 3;

…

}

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

Пример кода с веткой else на языке программирования Python:

tovar1 = 50

tovar2 = 32

**if** tovar1+ tovar2 > 99 :

**print**("99 рублей недостаточно")

**else**:

**print**("Чек оплачен")

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

a = ?

**if** a:

a = 1

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

a = 5 > 0

**if** a:

**print**(a)

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

Третий пример:

**if** a > 0 **and** a < b:

**print**(b - a)

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

**if** 0 < a < b:

**print**(b - a)

Пример множественного ветвления:

old = int(input('Ваш возраст: '))

**print**('Рекомендовано:', end=' ')

**if** 3 <= old < 6:

**print**('"Заяц в лабиринте"')

**elif** 6 <= old < 12:

**print**('"Марсианин"')

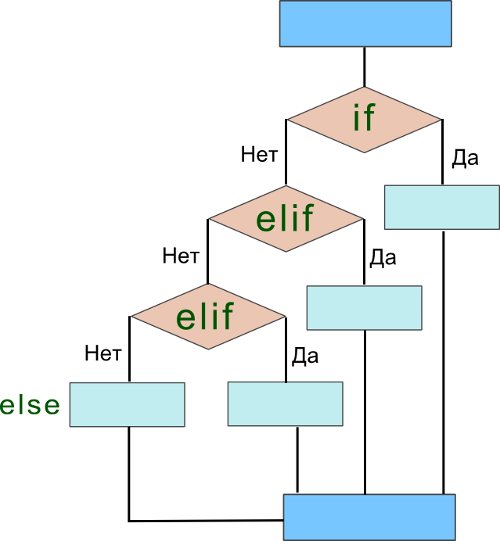
**elif** 12 <= old < 16:

**print**('"Загадочный остров"')

**elif** 16 <= old:

**print**('"Поток сознания"')

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



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

**Ошибки и исключения**

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

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

>>> 1a = 10

File "<stdin>", line 1

1a = 10

^

SyntaxError: invalid syntax

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

В Python не говорят о семантических ошибках, говорят об **исключениях**.

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

>>> a = 0

>>> **print**(a + b)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'b' is not defined

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

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

>>> a = 0

>>> **if** a == 0:

... **print**(a)

... **print**(a + b)

...

0

Traceback (most recent call last):

File "<stdin>", line 3, in <module>

NameError: name 'b' is not defined

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

>>> int("Hi")

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: invalid literal for int() with base 10: 'Hi'

>>> 8 + "3"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

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

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

Деление на ноль вызывает исключение ZeroDivisionError:

>>> 1/0

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero