Д. Ю. Федоров

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

PYTHON

УЧЕБНОЕ ПОСОБИЕ ДЛЯ СПО

Рекомендовано Учебно–методическим отделом среднего профессионального образования в качестве учебного пособия для студентов образовательных учреждений среднего профессионального образования

Книга доступна в электронной библиотеке biblio-online.ru, а также в мобильном приложении «Юрайт.Библиотека»

Москва ▪ Юрайт ▪ 2019

УДК 004.42(075.32)
ББК 32.973-018я723
Ф33

***Автор*:**

Федоров Дмитрий Юрьевич — старший преподаватель кафедры вычислительных систем и программирования факультета информатики и прикладной математики Санкт-Петербургского государственного экономического университета.

***Рецензенты*:**

Чудаков О. Е. — профессор, доктор технических наук, профессор кафедры специальных информационных технологий Санкт-Петербургского университета МВД России;

Трофимов В. В. — доктор технических наук, профессор, заслуженный деятель науки Российской Федерации, заведующий кафедрой информатики Санкт-Петербургского государственного экономического университета.

Федоров, Д. Ю.

Ф33 Программирование на языке высокого уровня Python : учеб. пособие для СПО / Д. Ю. Федоров. — М. : Издательство Юрайт, 2019. — 126 с. — (Серия : Профессиональное образование).

ISBN 978-5-534-05118-6

В учебном пособии рассматриваются теоретические основы современных технологий и методов программирования, практические вопросы создания программ, а также основные алгоритмические конструкции и их реализация на языке высокого уровня Python.

Соответствует актуальным требованиям Федерального государственного образовательного стандарта среднего профессионального образования и профессиональным требованиям.

Для студентов образовательных учреждений среднего профессионального образования.

УДК 004.42(075.32)
ББК 32.973-018я723

ISBN 978-5-534-05118-6

Оглавление

Предисловие

Современное общество является информационным, в нем все большее число людей занято получением, переработкой и использованием информации с применением компьютерных технологий. Компьютеры используются практически во всех областях человеческой деятельности: в профессиональной, учебной, досуговой сферах.

Дальнейшее развитие информационного общества требует разработки большого количества качественных программных продуктов, обеспечивающих удовлетворение растущих потребностей людей. В этих условиях весьма актуальным становится овладение современными технологиями программирования. Исследования в области управления персоналом показывают, что профессия программиста будет одной из самых востребованных в XXI в.

Целью и задачами учебного пособия «Программирование на языке высокого уровня Python» является систематизация и упорядочение сведений о технологиях разработки современных программных продуктов. В книге рассматриваются основные алгоритмические конструкции и их реализация на языке высокого уровня Python. Рассмотрение теоретических основ программирования сопровождается большим количеством примеров, иллюстрирующих приемы создания программ, а также заданиями для самостоятельного выполнения, позволяющими сформировать у обучающихся практические навыки программирования.

Пособие предназначено для студентов образовательных учреждений среднего профессионального образования, а также для всех, интересующихся современными технологиями программирования.

В результате изучения материалов пособия обучающиеся должны освоить:

трудовые действия

• владение основными принципами объектно-ориентированного подхода при создании программ;

• спецификой работы с переменными различных типов данных;

необходимые умения

• выполнять стандартные операции над данными различного типа;

• применять стандартные алгоритмические структуры для обработки данных;

необходимые знания

• структуру программы;

• основные типы данных, их особенности;

• стандартные модули языка.

Глава 1
ЗНАКОМСТВО С ЯЗЫКОМ ПРОГРАММИРОВАНИЯ PYTHON

Python — высокоуровневый язык программирования общего назначения с динамической типизацией, автоматическим управлением памятью, поддержкой многопоточных вычислений и удобными структурами данных. На рис. 1.1 перечислены области применения языка программирования Python.

Прежде чем переходить к выполнению программ на языке Python, рассмотрим, как запускаются программы на компьютере (рис. 1.2). Выполнение программ осуществляется операционной системой (Microsoft Windows, GNU/Linux и пр.). В задачи операционной системы входит выделение ресурсов (оперативной памяти и пр.) для программы, запрет или разрешение на доступ к устройствам ввода/вывода и т.д.

Для запуска программ, написанных на языке программирования Python, необходима программа-интерпретатор[1] (*виртуальная*

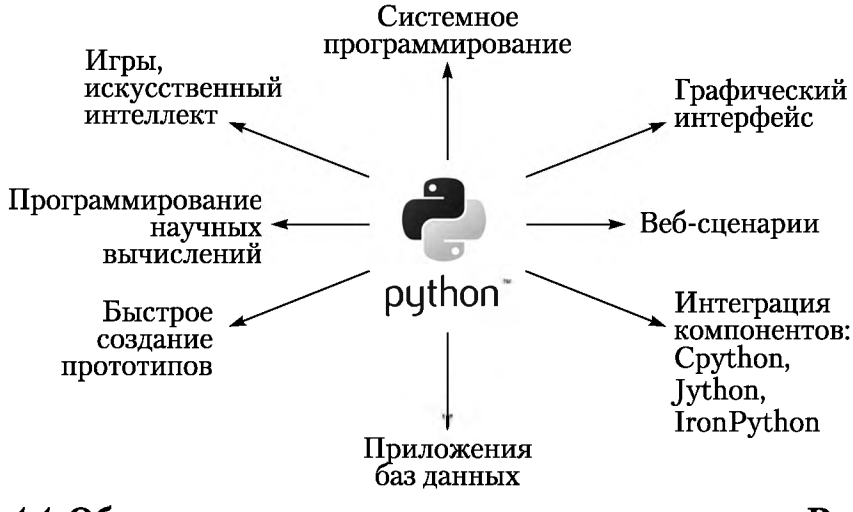


Рис. 1.1. **Области применения языка программирования Python**

[1] Python является интерпретируемым языком программирования (команды выполняются шаг за шагом), в отличие от компилируемых, где текст программы переводится в эквивалентный машинный код.

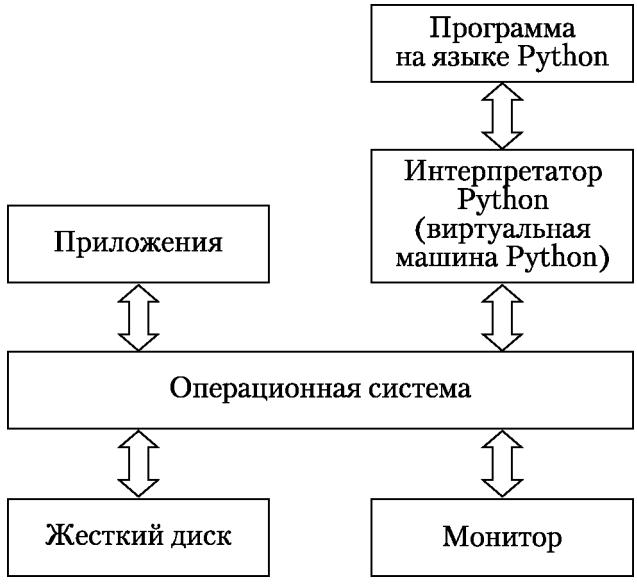


Рис. 1.2. **Схема запуска программ**

машина) Python. Данная программа скрывает от Python-программиста все особенности операционной системы, поэтому, написав программу на Python в системе Windows, ее можно запустить, например, в GNU/Linux и получить схожий результат.

Скачать и установить интерпретатор Python[1] можно совершенно бесплатно с официального сайта http://python.org. Для работы нам понадобится интерпретатор Python версии 3.6 или выше. В процессе установки рекомендуется указать путь **C:/Python36-32**.

После установки программы-интерпретатора запустите интерактивную графическую среду IDLE[2] и дождитесь появления приглашения для ввода команд:

```
Type "copyright", "credits" or "license()" for more
information.
>>>
```

[1] Процесс установки зависит от операционной системы.

[2] Выбор среды IDLE обусловлен тем, что она входит в стандартную поставку интерпретатора Python и является свободно распространяемой.

Глава 2
ИНТЕЛЛЕКТУАЛЬНЫЙ КАЛЬКУЛЯТОР

В самом начале обучения Python можно рассматривать как интерактивный интеллектуальный калькулятор. В интерактивном режиме IDLE найдем значения математических выражений. После завершения набора выражения нажмите клавишу <*Enter*> для завершения ввода и последующего выполнения указанных выражений:

```
>>> 3.0 + 6
9.0
>>> 4 + 9
13
>>> 1 - 5
-4
>>> _ + 6
2
```

Нижним подчеркиванием в предыдущем примере обозначается последний полученный результат.

Если совершить ошибку при вводе команды, то интерпретатор Python сообщит об этом:

```
>>> a
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

В математических выражениях в качестве операндов могут использоваться как *целые числа*[1] (1, 4, –5), так и *вещественные*[2] (в программировании их еще называют *числами с плавающей точкой*): 4.111, –9.3. Математические операции, доступные над числами в Python[3], представлены в табл. 2.1.

---

[1] А также комплексные числа и логические значения (**True**, **False**).

[2] Объяснение правил хранения вещественных чисел в компьютере выходит за рамки учебника, сравните в следующем примере результаты вычислений:

```
>>> 2/3 + 1
1.6666666666666665
>>> 5/3
1.6666666666666667
>>>
```

[3] Любопытно, что в Python выражение **(b * (a // b) + a % b)** эквивалентно **a**.

Таблица 2.1

Математические операторы в Python

Оператор	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление (в результате вещественное число)
//	Деление с округлением вниз
**	Возведение в степень
%	Остаток от деления

```
>>> 5/3
1.6666666666666667
>>> 5 // 3
1
>>> 5 % 3
2
>>> 5 ** 67
677626357803440271254658000543713569641113281 25
```

Если один из операндов является вещественным числом, то в результате вычислений получится вещественное число.

При вычислении математических выражений Python придерживается приоритета операций (следует математическим соглашениям):

```
>>> -2 ** 4
-16
>>> -(2**4)
-16
>>> (-2) ** 4
16
```

В случае сомнений в порядке применения математических операторов и для упрощения чтения выражений будет полезным обозначить приоритет в виде круглых скобок.

Выражаясь в терминах программирования, только что мы познакомились с *числовым типом данных* (целочисленным типом **int** и вещественным типом **float**), т.е. множеством числовых значений и множеством математических операций, которые можно выполнять над данными значениям.

Язык программирования Python предоставляет большой выбор встроенных типов данных, о которых речь пойдет дальше.

Глава 3
ПЕРЕМЕННЫЕ

Рассмотрим выражение **y = x + 3 * 6**, где **y** и **x** являются *переменными*, которые могут содержать некоторые значения. На языке Python вычислить значение **y** при **x** равном 1 можно следующим образом:

```
>>> x = 1
>>> y = x + 3 * 6
>>> y
19
```

В выражении нельзя использовать переменную, если ранее ей не было присвоено значение с помощью *инструкции присваивания*. Для Python такие переменные не определены, и их использование приведет к ошибке.

Содержимое переменной **y** можно вывести на экран, если в интерактивном режиме ввести ее имя.

Имена переменным задает программист, но есть несколько ограничений, связанных с их наименованием. Имена переменных нельзя начинать с цифры и в качестве имен переменных нельзя использовать ключевые слова, которые для Python имеют определенный смысл (эти слова подсвечиваются в IDLE оранжевым цветом, табл. 3.1).

Таблица 3.1

Ключевые слова Python

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

Далее мы часто будем обращаться к формуле перевода из шкалы в градусах по Цельсию (T_C) в шкалу в градусах по Фаренгейту (T_F):

```
TF = 9/5 * TC + 32
```

Определим значение T_F при T_C, равном 26. Создадим переменную с именем **cel**, содержащую значение целочисленного типа 26:

```
>>> cel = 26
>>> cel
26
>>> 9/5 * cel + 32
78.80000000000001
```

В момент выполнения инструкции присваивания **cel** = 26 в памяти компьютера создается *объект* (рис. 3.1), расположенный по некоторому *адресу* (условно обозначим его как *id*1), имеющий значение 26 целочисленного типа **int**. Затем создается переменная с именем **cel**, которой *присваивается адрес* объекта *id*1.

Таким образом, переменные в Python содержат адреса объектов. Иначе можно сказать, что *переменные ссылаются на объекты.* Тип переменной определяется типом объекта, на который она ссылается. В дальнейшем для упрощения будем говорить, что переменная хранит значение.

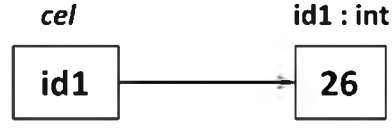


Рис. 3.1. **Модель памяти для выражения cel = 26**[1]

Вычисление следующего выражения приведет к присваиванию переменной **cel** значения 72, т.е. сначала вычисляется правая часть, затем результат присваивается левой части:

```
>>> cel = 26 + 46
>>> cel
72
```

Рассмотрим более сложный пример:

```
>>> diff = 20
>>> double = 2 * diff
>>> double
40
```

Во втором выражении в первую очередь произойдет вычисление правой части, где на место переменной **diff** подставится значение 20, и результат вычисления присвоится переменной **double**. По окончании вычислений память будет иметь вид, представленный на рис. 3.2.

---

[1] Рисунки к учебному пособию даны по изданию: *Федоров Д. Ю.* Основы программирования на примере языка Python : учеб. пособие. СПб., 2016.

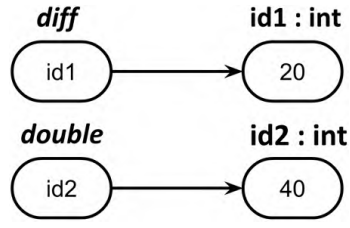


Рис. 3.2. **Схема памяти Python при работе с переменными**

Далее присвоим переменной **diff** значение 5 и выведем на экран содержимое переменных **double** и **diff**:

```
>>> diff = 5
>>> double
40
>>> diff
5
```

В момент присваивания переменной **diff** значения 5 в памяти (рис. 3.3) создастся объект по адресу *id*3, содержащий целочисленное значение 5. После этого изменится содержимое переменной **diff**, вместо адреса *id*1 туда запишется адрес *id*3. Также Python увидит, что на объект по адресу *id*1 больше никто не ссылается и поэтому удалит его из памяти (произведет автоматическую *сборку мусора*).

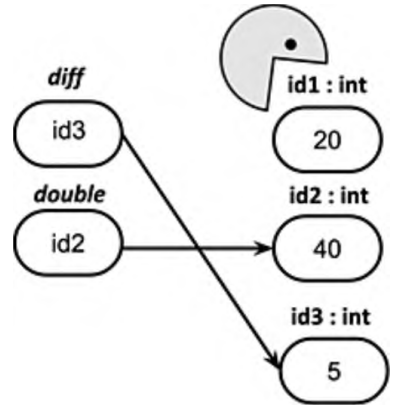


Рис. 3.3. **Схема памяти Python при работе с переменными**

Внимательный читатель заметил, что Python не изменяет существующие числовые объекты, а создает новые, т.е. объекты числового типа данных в Python являются *неизменяемыми*.

У начинающих программистов часто возникает недоумение при виде следующих выражений:

```
>>> num = 20
>>> num = num * 3
>>> num
60
```

Если вспомнить, что сначала вычисляется правая часть выражения, то все станет на свои места.

Глава 4
ФУНКЦИИ

Функцией в программировании называется последовательность инструкций, которая выполняет вычисления. С чем можно сравнить функцию? Напрашивается аналогия с «черным ящиком», когда мы знаем, что поступает на вход и что при этом получается на выходе, а внутренности «черного ящика» от нас скрыты. В качестве примера можно привести банкомат. На вход банкомата поступает пластиковая карточка (пин-код, денежная сумма), на выходе мы ожидаем получить запрашиваемую сумму. Нас не очень сильно интересует принцип работы банкомата до тех пор, пока он работает без сбоев.

Рассмотрим встроенную функцию с именем **abs**, принимающую на вход один аргумент — объект числового типа, и возвращающую абсолютное значение для этого объекта (рис. 4.1).

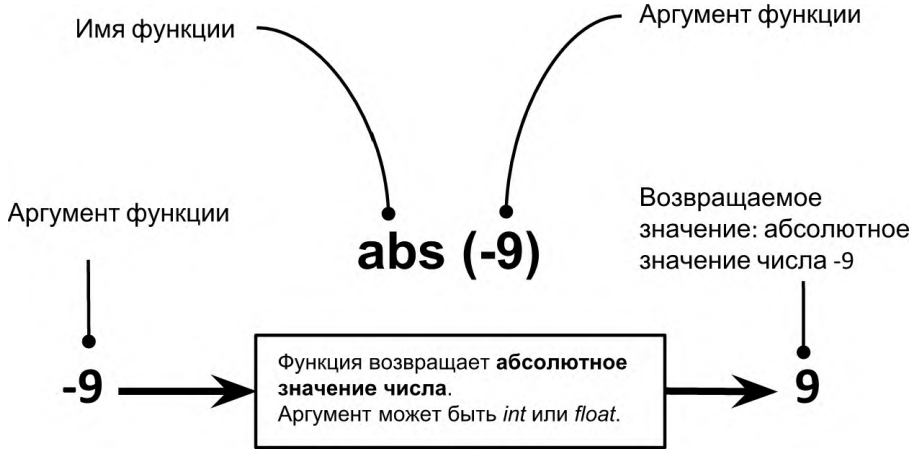


Рис. 4.1. **Функция как «черный ящик»**

Пример вызова функции **abs** с *аргументом* –9 имеет вид:

```
>>> abs(-9)
9
>>> d = 1
>>> n = 3
>>> abs(d - n)
2
```

```
>>> abs(-9) + abs(5.6)
14.6
```

Результат вызова функции можно присвоить переменной, использовать его в качестве операндов математических выражений, что позволяет формировать более сложные выражения.

Рассмотрим примеры нескольких встроенных математических функций.

Функция **pow(x, y)** возвращает значение **x** в степени **y**. Эквивалентно записи **x******y**, с которой мы уже встречались.

```
>>> pow(4, 5)
1024
```

Функция **pow** может принимать третий аргумент, тогда вызов функции с аргументами **pow(x, y, z)** эквивалентен вычислению выражения **(x ** y) % z**:

```
>>> pow(4, 5, 3)
1
```

Функция **round(number)** возвращает число с плавающей точкой, округленное до 0 цифр после запятой (по умолчанию). Функция может быть вызвана с двумя аргументами: **round(number [, ndigits])**, где **ndigits** — число знаков после запятой:

```
>>> round(4.56666)
5
>>> round(4.56666, 3)
4.567
```

Помимо составления сложных математических выражений Python позволяет передавать результат вызова функции в качестве аргументов других функций без использования дополнительных переменных.

На рис. 4.2 представлен пример вызова функций и порядок их вычисления. В этом примере на месте числовых объектов –2, 4.3 могут находиться более сложные выражения, поэтому они также нуждаются в вычислении.

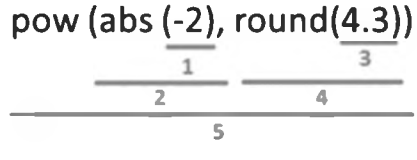


Рис. 4.2. **Порядок вычисления составного выражения**

На практике часто при написании программ требуется преобразовывать типы объектов.

Функция **int** принимает любое значение и преобразует его в целое число, если это возможно (возвращает 0, если аргументы не переданы):

```
>>> int()
0
```

Функция **int** может преобразовать число с плавающей точкой в целое, но это не округление, а отсечение дробной части:

```
>>> int(5.6)
5
```

Функция **float** возвращает число с плавающей точкой, построенное из числа или строки[1] (возвращает 0.0, если аргументы не переданы):

```
>>> float(5)
5.0
>>> float()
0.0
```

Описание функций содержится в документации, которая может быть вызвана с помощью функции **help** (на вход подается имя функции):

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Вернемся к формуле перевода градусов по шкале Фаренгейта (T_F) в градусы по шкале Цельсия (T_C):

```
TC = 5/9 * (TF - 32)
```

Произведем несколько вычислений, где переменная **deg_f** будет содержать значение в градусах по Фаренгейту:

```
>>> deg_f = 80
>>> deg_f
80
>>> 5/9 * (deg_f - 32)
26.666
>>> deg_f = 70
>>> 5/9 * (deg_f - 32)
```

Заметим, что каждый раз для перевода приходится набирать одно и то же выражение. Упростим вычисления, создав собственную функцию, переводящую градусы по шкале Фаренгейта в градусы по шкале Цельсия.

В первую очередь необходимо придумать имя функции (рис. 4.3), к примеру, назовем функцию **convert_co_cels**. Постарайтесь, чтобы имя было осмысленным (**lena123** — плохой пример) и отражало смысл функции, вспомните о правилах наименования переменных. Помимо этого, нежелательно, чтобы имя вашей функции совпада-

[1] О строках речь пойдет ниже.

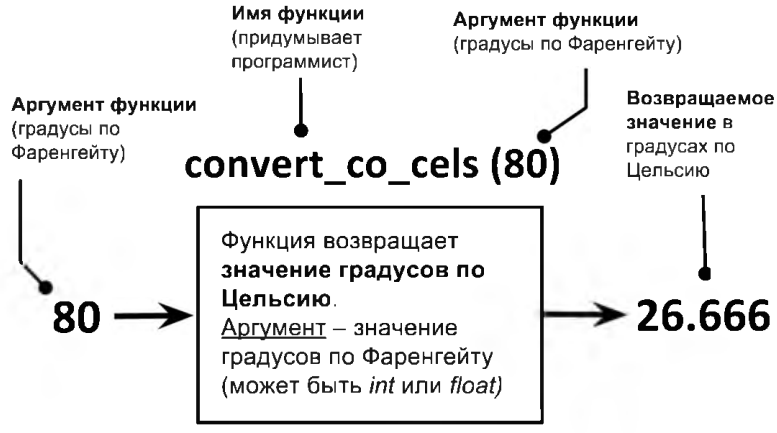


Рис. 4.3. **Создание собственной функции**

ло с именами встроенных функций Python (встроенные функции в IDLE подсвечиваются фиолетовым цветом).

Представим, что функция с именем **convert_co_cels** создана, тогда ее вызов для значения (аргумента) 80 будет иметь вид: **convert_co_cels(80)**.

Перейдем непосредственно к созданию функции (рис. 4.4). Ключевое слово **def** означает, что далее следует определение функции. После **def** указывается имя функции **convert_co_cels**, затем в скобках указывается *параметр* (или параметры), которому будет присваиваться значение при вызове функции. Параметры функции — обычные переменные, которыми функция пользуется для внутренних вычислений. Переменные, объявленные внутри функции, называются *локальными* и не видны вне функции. После символа «:» начинается *тело функции* — блок команд, относящийся к функции. Тело функции может содержать любое количество инструкций. В интерактивном режиме Python самостоятельно расставит отступы[1] от края редактора, тем самым обозначив, где начи-

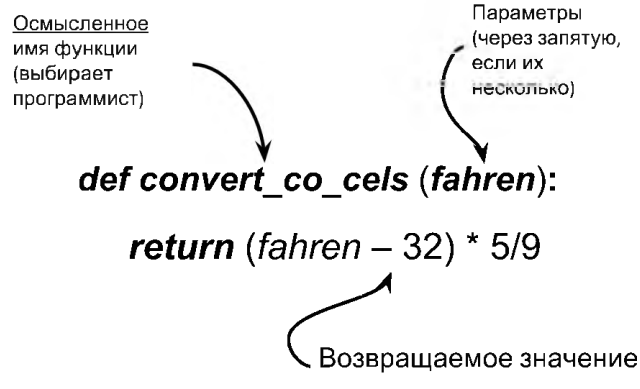


Рис. 4.4. **Схема создания функции в Python**

[1] Отступы играют важную роль в Python, отделяя блок команд тела функции, цикла и пр.

нается тело функции. Выражение, стоящее после инструкции **return**, будет возвращаться в качестве результата вызова функции.

В интерактивном режиме создание функции имеет следующий вид (для завершения ввода функции необходимо два раза нажать клавишу *<Enter>*, дождавшись приглашения для ввода команд >>>):

```
>>> def convert_co_cels(fahren):
    return (fahren-32) * 5/9

>>> convert_co_cels(451)
232.77777777777777
>>> convert_co_cels(300)
148.88888888888889
```

После создания функции ее можно вызвать, подставив в скобках аргументы, т.е. задав конкретные значения.

Глава 5
ПРОГРАММЫ В ОТДЕЛЬНОМ ФАЙЛЕ

Внимательный читатель заметил, что в интерактивном режиме нельзя вернуться и внести изменения в выражение, которое уже было выполнено, поэтому приходится набирать его повторно.

Многострочные выражения, где легко допустить ошибку при наборе, удобно помещать в отдельные текстовые файлы с расширением **.py**.

В меню IDLE выберите *File → New File*. Появится окно текстового редактора, в котором можно набирать команды на языке программирования Python. Наберем в редакторе следующий код:

```
# это комментарии, и они игнорируются Python
# firstprog.py
a=5
print(a)
print(a+5)
```

В меню редактора выберем *Save As* и сохраним файл в директорию **C:/Python36-32/**, указав произвольное имя, например **firstprog.py**. В ранних версиях IDLE приходилось вручную прописывать расширение файла.

Для выполнения программы в меню редактора IDLE выберем *Run → Run Module* (или нажмем клавишу *<F5>*). Результат работы программы отобразится в интерактивном режиме:

```
========RESTART: C:/Python36-32/firstprog.py ========
5
10
```

Функция **print** в примере отображает содержимое переменных, переданных ей в качестве аргументов. Вспомним, что интерактивный режим позволял нам обходиться без вызова этой функции.

Разберемся теперь, как создавать и вызывать функции, находящиеся в отдельном файле. Создадим файл **myfunc.py**, содержащий следующий код (тело функции в Python принято отделять либо че-

тырьмя пробелами, либо одной табуляцией — придерживайтесь в рамках файла одного из этих правил):

```
# myfunc.py
def f(x):
    x = 2 * x
    return x
```

Выполним программу с помощью нажатия на клавишу *<F5>*. Увидим, что в интерактивном режиме программа выполнилась, но ничего не вывела на экран:

```
>>>
```

Все правильно, ведь мы не вызвали функцию! После запуска программы в интерактивном режиме вызовем функцию **f** с различными аргументами:

```
>>> f(4)
8
>>> f(56)
112
```

Работает! Теперь вызовем функцию **f** в файле, но не забудем про **print**. Далее представлена обновленная версия программы:

```
# myfunc2.py
def f(x):
    x = 2 * x
    return x

print(f(4))
print(f(56))
```

Выполним программу и увидим, что в интерактивном режиме отобразился результат:

```
8
112
```

Глава 6
ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ

Ранее мы отметили, что переменная является *локальной* (видна только внутри функции), если значение ей присваивается внутри функции, в ином случае — переменная *глобальная*, т.е. видна (к ней можно обратиться) во всей программе, в том числе и внутри функции.

В отдельный файл с именем **var_prog.py** поместим следующий код:

```
# var_prog.py
a = 3 # глобальная переменная
print('глобальная переменная a = ', a)
y = 8 # глобальная переменная
print('глобальная переменная y = ', y)

def func():
    print('func: глобальная переменная a = ', a)
    y = 5 # локальная переменная
    print('func: локальная переменная y = ', y)

func() # вызываем функцию func
print('??? y = ', y) # отобразится глобальная переменная
```

Обратим внимание, что у функции **print** может быть несколько аргументов, заданных через запятую. В одиночные апострофы помещается строка[1]. После выполнения программы получим следующий результат:

```
глобальная переменная a = 3
глобальная переменная y = 8
func: глобальная переменная a = 3
func: локальная переменная y = 5
??? y = 8
```

Внутри функции мы смогли обратиться к глобальной переменной **a** и вывести ее значение на экран. Затем внутри функции создается локальная переменная **y**, причем ее имя совпадает с именем

[1] Можно было бы использовать двойные кавычки. О строках речь пойдет ниже.

глобальной переменной — в этом случае при обращении к **y** выводится содержимое локальной переменной, а глобальная переменная остается неизменной.

Как быть, если мы хотим изменить содержимое глобальной переменной внутри функции? В следующем примере демонстрируется подобное изменение с использованием ключевого слова **global**:

```
# var_prog2.py
x = 50        # глобальная переменная
def func():
    global x # указываем, что x глобальная переменная
    print('x равно', x)
    x = 2     # изменяем глобальную переменную
print('Заменяем глобальное значение x на', x)

func()
print('Значение x составляет', x)
```

Результат работы программы **var_prog2.py**:

```
x равно 50
Заменяем глобальное значение x на 2
Значение x составляет 2
```

Глава 7
ПРИМЕНЕНИЕ ФУНКЦИЙ

На практике часто функции используются для сокращения исходного кода программы, например, если объявить функцию вида:

```
def func(x):
    c = 7
    return x + 8 + c
```

то код на рис. 7.1 может быть заменен тремя вызовами функции с различными аргументами.

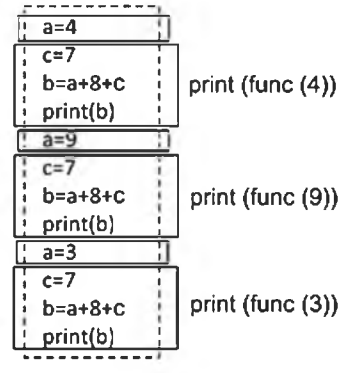


Рис. 7.1. **Пример многократного вызова функции**

Бывают случаи, когда функция ничего не принимает на вход и ничего не возвращает[1] (явно не используется инструкция **return**).

Пример подобной функции:

```
def print_hello():
    print('Привет')
    print('Hello')
    print('Hi')
```

Видим, что внутри функции **print_hello** происходит вызов **print**, поэтому в момент вызова **print_hello** еще раз вызывать **print** не требуется. Пример на рис. 7.2 демонстрирует, что множество повторяющихся вызовов **print** можно заменить несколькими вызовами функции **print_hello**.

---

[1] На самом деле, если не указать инструкцию return, то Python все равно вернет объект None.

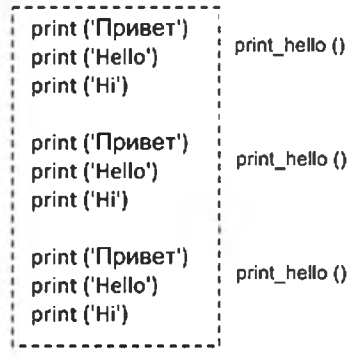


Рис. 7.2. **Пример использования вызова функций**

Имена функций в Python являются обычными переменными, содержащими адрес объекта[1] типа функция[2], поэтому этот адрес можно присвоить другой переменной и вызвать функцию уже с другим именем:

```
# call_func.py
def summa(x, y):
    return x + y
f = summa
v = f(10, 3) # вызываем функцию с другим именем
```

Параметры функции могут принимать значения по умолчанию:

```
# func_var.py
def summa(x, y = 2):
    return x + y
a = summa(3) # вместо y подставляется значение
             # по умолчанию
b = summa(10, 40) # теперь значение второго параметра
                  # равно 40
```

В связи с тем, что имя функции является обычной переменной, можно передать ее в качестве аргумента при вызове функции:

```
# call_summa.py
def summa(x, y):
    return x + y
def func(f, a, b):
    return f(a, b)
v = func(summa, 10, 3) # передаем summa в качестве
                       # аргумента
```

Этот пример показывает, как из функции **func** можно вызвать функцию **summa**.

Помимо этого, в момент вызова функции можно присваивать значения конкретным параметрам (использовать *ключевые аргументы*):

---

[1] В Python все является объектами.

[2] Да-да, это еще один тип данных в Python.

```
# key_arg.py
def func(a, b=5, c=10):
    print('a равно', a, ', b равно', b, ', a c равно', c)

func(3, 7) # a=3, b=7, c=10
func(25, c=24) # a=25, b=5, c=24
func(c=50, a=100) # a=100, b=5, c=50
```

Ошибкой будет являться вызов функции, при котором не задан аргумент **a**, так как для него не указано значение по умолчанию.

Глава 8
СТРОКИ И ОПЕРАЦИИ НАД СТРОКАМИ

Для работы с текстом в Python предусмотрен специальный строковый тип данных **str**. Строковые объекты создаются, если текст поместить в одиночные апострофы или двойные кавычки:

```
>>> 'hello'
'hello'
>>> "Hello"
'Hello'
```

Без кавычек Python расценит текст как имя переменной и попытается вывести на экран ее содержимое, если такая переменная существует:

```
>>> hello
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    hello
NameError: name 'hello' is not defined
```

Можно создать пустую строку:

```
>>> ''
''
```

Для работы со строками в Python предусмотрено большое число встроенных функций. Рассмотрим, например, функцию **len**, которая определяет длину строки, переданной ей в качестве аргумента:

```
>>> help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Пример вызова функции **len** для строкового аргумента:

```
>>> len('Привет!')
7
```

Глава 9
ОПЕРАЦИИ НАД СТРОКАМИ

С помощью операции *конкатенации* (оператор «+» для строк) Python позволяет объединить несколько строк в одну (также допускается расположить строки последовательно без каких-либо операторов):

```
>>> 'Привет, ' + 'земляне!'
'Привет, земляне!'
>>> 'Привет, ' 'земляне!'
'Привет, земляне!'
```

Здесь начинаются удивительные вещи! Помните, мы говорили, что операции зависят от типа данных? Над объектами определенного типа можно производить только определенные операции: числа — складывать, умножать и т.д., т.е. производить над ними арифметические операции. Так вот, для строковых объектов операция сложения объединяет строки, а для числовых — складывает. Что произойдет, если применить оператор сложения одновременно к числу и строке?

```
>>> 'Марс' + 5
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    'Марс' + 5
TypeError: must be str, not int
```

Python не разобрался, что мы от него хотим: сложить числа или объединить строки. К примеру, чтобы объединить строки, можно с помощью функции **str** преобразовать число 5 в строку '5' и выполнить объединение:

```
>>> 'Марс' + str(5)
'Марс5'
```

Можно произвести обратное преобразование типов (из строки в число):

```
>>> int("-5")
-5
```

Попросим Python повторить (размножить) строку 10 раз:

```
>>> "СПАМ" * 10
'СПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМ'
```

Отметим, что оператор умножения для строковых объектов приобрел новый смысл.

Строки, по аналогии с числами, можно присваивать[1] переменным:

```
>>> s = "Я изучаю программирование"
>>> s
'Я изучаю программирование'
>>> s*4
'Я изучаю программированиеЯ изучаю программированиеЯ
изучаю программированиеЯ изучаю программирование'
>>> s + " на языке Python"
'Я изучаю программирование на языке Python'
```

Поместить разные виды кавычек в строку можно несколькими способами:

```
>>> "Hello's"
"Hello's"
>>> 'Hello\'s'
"Hello's"
```

Первый способ — заключить строку в кавычки разных типов, чтобы указать Python, где заканчивается строка. Второй — использовать специальные символы (*управляющие escape-последовательности*), которые записываются как два символа, но Python видит их как один:

```
>>> len("\'")
1
```

Полезно помнить часто встречающиеся управляющие последовательности:

```
\n - переход на новую строку
\t - знак табуляции
\\ - наклонная черта влево
\' - символ одиночной кавычки
\" - символ двойной кавычки
```

При попытке разбить длинную строку с помощью <*Enter*> возникает ошибка:

```
>>> 'Это длинная
SyntaxError: EOL while scanning string literal
>>> строка
Traceback (most recent call last):
File "<pyshell#20>", line 1, in <module>
строка
NameError: name 'строка' is not defined
```

---

[1] Напоминаем, что в переменной хранится адрес объекта (в данном случае строкового объекта).

Для создания многострочной строки ее необходимо заключить с обеих сторон в три одиночных апострофа:

```
>>> '''Это длинная
строка'''
'Это длинная\nстрока'
```

Отметим, что при выводе на экран многострочной строки перенос строки отобразился в виде специального символа '**\n**'.

Глава 10
ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ ФУНКЦИИ PRINT

Вернемся к уже встречавшейся нам функции **print**. Передадим на вход этой функции строку со специальным символом:

```
>>> print('Это длинная\nстрока')
Это длинная
строка
```

Функция **print** смогла распознать специальный символ и отобразить на экране перевод строки. Рассмотрим еще несколько примеров:

```
>>> print(1, 3, 5)
1 3 5
>>> print(1, '2', 'снова строка', 56)
1 2 снова строка 56
```

Таким образом, функция **print** позволяет выводить объекты разных типов.

На самом деле у этой функции есть несколько «скрытых» аргументов, которые задаются по умолчанию в момент вызова (рис. 10.1).

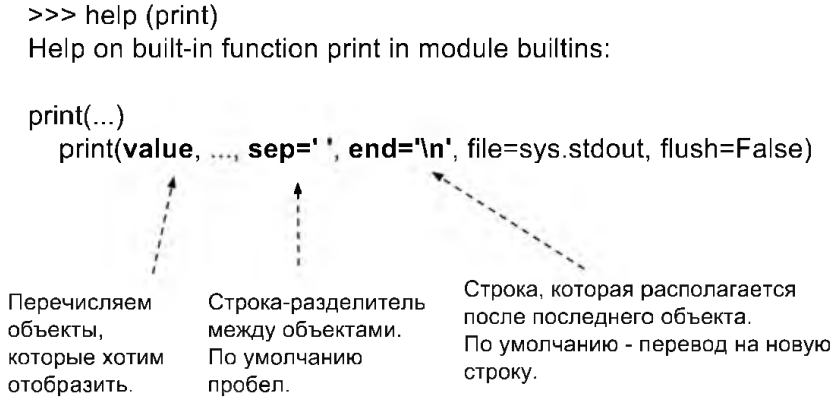


Рис. 10.1. **«Скрытые» параметры функции print**

Рассмотрим пример:

```
>>> print(1, 6, 7, 8, 9)
1 6 7 8 9
```

По умолчанию функция **print** для вывода набора объектов использует в качестве разделителя пробел. Для задания другого разделителя можно изменить значение параметра **sep=':'** в момент вызова функции:

```
>>> print(1, 6, 7, 8, 9, sep=':')
1:6:7:8:9
```

Глава 11
ВВОД ЗНАЧЕНИЙ С КЛАВИАТУРЫ

Усложним программу и попросим пользователя ввести что-нибудь с клавиатуры. В Python для этого предусмотрена специальная функция **input**:

```
>>> s = input()
Земляне, мы прилетели с миром!
>>> s
'Земляне, мы прилетели с миром!'
>>> type(s)
<class 'str'>
```

В примере мы вызвали функцию **input** и результат ее работы присвоили переменной **s**. После этого пользователь ввел значение с клавиатуры и нажал *<Enter>*, т.е. указал на завершение ввода. Содержимое переменной **s** вывели на экран — отобразилась фраза, которую ввел пользователь. Затем вызвали функцию **type**, которая позволяет определить тип объекта, ссылка на который содержится в переменной **s**.

Внимание: функция **input** возвращает строковый объект!

Работа с функцией **input** может привести к неожиданным ошибкам:

```
>>> s = input("Введите число: ")
Введите число: 555
>>> s + 5
Traceback (most recent call last):
File "<pyshell#33>", line 1, in <module>
s + 5
TypeError: must be str, not int
```

В этом примере используется входной аргумент функции **input**, который выводит на экран строку-приглашение перед пользовательским вводом: *"Введите число: "*. После ввода значения с клавиатуры пытаемся сложить его с числом 5 и вместо ожидаемого результата получаем сообщение об ошибке. Исправить ошибку позволяет преобразование типов (вызов функции **int**):

```
>>> s = int(input("Введите число: "))
Введите число: 555
>>> s + 5
560
```

Каждый символ строки имеет собственный порядковый номер (*индекс*). Нумерация символов начинается с нуля. Python позволяет обратиться к заданному символу строки с помощью оператора скобок следующим образом:

```
>>> s = 'Я люблю писать программы!'
>>> s[0]
'Я'
>>> s[-1]
'!'
```

В квадратных скобках указывается индекс символа. Если нулевой индекс — первая буква строки, что тогда означает индекс –1? Очевидно, что последний символ. Для перевода отрицательных индексов в положительные используется следующая формула: длина строки + отрицательный индекс. Например, для –1: **len(s)**–1, т.е. 24.

```
>>> len(s)-1
24
>>> s[24]
'!'
```

Попробуем изменить первый символ строки (с нулевым индексом):

```
>>> s = 'Я люблю писать программы!'
>>> s[0] = 'J'
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    s[0] = 'J'
TypeError: 'str' object does not support item assignment
```

Попытка изменить символ в строке **s** привела к ошибке. Дело в том, что в Python строки, как и числа, являются неизменяемыми объектами.

Работа со строковыми объектами не отличается от работы с числовыми объектами (рис. 11.1).

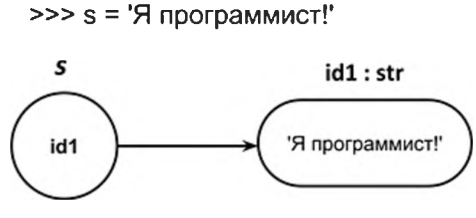


Рис. 11.1. **Схема присвоения переменной строкового объекта**

В момент изменения значения переменной **s** (рис. 11.2) будет создан новый строковый объект по адресу *id*2, и этот адрес будет записан в переменную **s**.

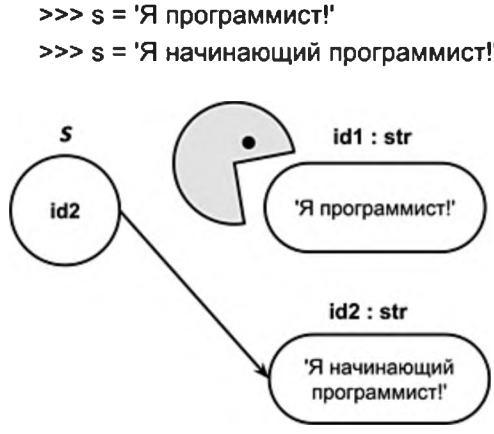


Рис. 11.2. **Схема присвоения переменной нового строкового объекта**

Прежде чем мы поймем, как можно изменить строку, познакомимся со *срезами*:

```
>>> s = 'Питоны водятся в Африке'
>>> s[1:3]
'ит'
```

s[1:3] — срез (часть) строки **s**, начиная с индекса 1, заканчивая индексом 3 (не включительно). Это легко понять, если индексы представить в виде смещений (рис. 11.3).

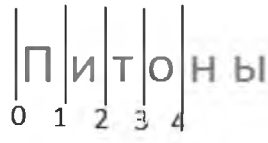


Рис. 11.3. **Индексы символов в строке как смещения**

Со срезами можно производить различные манипуляции:

```
>>> s[:3] # с 0 индекса по третий не включительно
'Пит'
>>> s[:] # вся строка
'Питоны водятся в Африке'
>>> s[::2] # третий аргумент задает шаг (по умолчанию один)
'Птн ояс фие'
>>> s[::-1] # «обратный» шаг
'екирфА в ястядов ынотиП'
>>> s[:-1] # вспомним, как мы находили отрицательный
           # индекс
'Питоны водятся в Африк'
>>> s[-1:] # снова отрицательный индекс
'е'
```

Теперь, используя срезы, изменим первый символ в строке, создав новую строку:

```
>>> s = 'Я люблю писать программы!'
>>> 'J' + s[1:] # формируется новая строка
'J люблю писать программы!'
```

Глава 12
ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ

В Python для сравнения чисел предусмотрено несколько операторов сравнения:

- > — больше;
- < — меньше;
- >= — больше или равно;
- <= — меньше или равно;
- == — равно;
- != — не равно.

В интерактивном режиме произведем сравнение двух чисел:

```
>>> 6 > 5
True
>>> 7 < 1
False
>>> 7 == 7 # не путайте == и = (присвоить)
True
>>> 7 != 7
False
```

Python возвращает **True**[1] (истину[2]), когда сравнение верное, и **False** (ложь) — в ином случае. **True** и **False** относятся к *логическому* (*булевому*) типу данных **bool**:

```
>>> type(True)
<class 'bool'>
```

Рассмотрим более сложные логические выражения.

Логическим высказыванием (*предикатом*)[3] будем называть любое повествовательное предложение, в отношении которого можно однозначно сказать, истинно оно или ложно.

---

[1] Обязательно с большой буквы.

[2] **True** интерпретируется Python как число **1**, а **False** как число **0**.

[3] Информатика. Теория (с задачами и решениями). Интернет-версия издания: *Шауцукова Л. З.* Информатика 10—11. М. : Просвещение, 2000 (режим доступа: http://book.kbsu.ru/theory/chapter5/1_5_1.html).

Например, высказывание «6 — четное число». Истинно или ложно? Очевидно, что истинно. А высказывание «6 больше 19»? Высказывание ложно.

Является ли высказыванием фраза «у него голубые глаза»? Однозначности в этой фразе нет, поэтому она не является высказыванием.

Высказывания можно комбинировать. Высказывания «Петров — врач», «Петров — шахматист» можно объединять с помощью связок И (and), ИЛИ (or).

«Петров — врач И шахматист». Это высказывание истинно, если *оба* высказывания «Петров — врач» И «Петров — шахматист» являются истинными.

«Петров — врач ИЛИ шахматист». Это высказывание истинно, если истинным является *одно* из высказываний «Петров — врач» или «Петров — шахматист».

Рассмотрим пример комбинаций из высказываний на языке Python:

```
>>> 2 > 4
False
>>> 45 > 3
True
>>> 2 > 4 and 45 > 3 # комбинация False and (и) True
                     # вернет False
False
>>> 2 > 4 or 45 > 3  # комбинация False or (или) True
                     # вернет True
True
```

Все, что сказано выше о комбинации логических высказываний, можно объединить и представить в виде таблицы истинности, где 0 — **False**, а 1 — **True** (табл. 12.1).

Таблица 12.1

Истинность комбинаций логических высказываний

X Y	and	or
0 0	0	0
0 1	0	1
1 0	0	1
1 1	1	1

Для Python истинным или ложным может быть не только логическое высказывание, но и объект.

Внимание: в Python любое число, не равное нулю, или непустой объект интерпретируются как истина. Числа, равные нулю, пустые объекты и специальный объект **None**[1] интерпретируются как ложь.

[1] Имеет специальный тип **NoneType**.

Рассмотрим пример:

```
>>> '' and 2 # False and True
''
>>> '' or 2 # False or True
2
```

В примере логический оператор **and** (и) применяется к двум операндам: пустому строковому объекту (ложный) и ненулевому числовому объекту (истинный). В итоге Python вернул нам пустой строковый объект. Затем аналогично применяется логический оператор **or** (или), в результате чего мы получили числовой объект. Разберемся, откуда взялся такой результат.

Рассмотрим более подробно три логических оператора: **and**, **or**, **not**.

Отрицание **not** (не) — наиболее простой из них, поэтому начнем с него:

```
>>> y = 6 > 8
>>> y
False
>>> not y
True
>>> not None
True
>>> not 2
False
```

Результатом применения логического оператора **not** будет отрицание операнда, т.е. если операнд истинный, то результатом применения оператора **not** станет ложь, и наоборот.

В результате применения логического оператора **and** (и) получим **True** (истину) или **False** (ложь)[1], если его операндами являются логические высказывания:

```
>>> 2 > 4 and 45 > 3 # комбинация False and (и) True
                     # вернет False
False
```

Если операндами оператора **and** являются объекты, то в результате Python вернет объект:

```
>>> '' and 2 # False and True
''
```

Для вычисления результата применения оператора **and** Python вычисляет операнды слева направо и возвращает первый объект, имеющий ложное значение.

Посмотрим на столбец **and** таблицы истинности (см. табл. 12.1) и проследим закономерность. Если среди операндов **X**, **Y** есть лож-

[1] Исходя из таблицы истинности.

ный, то результатом является ложное значение, но вместо ложного значения для операндов-объектов Python возвращает первый ложный операнд, встретившийся в выражении, и дальше вычисления не производит. Это называется *вычислением по короткой схеме*:

```
>>> 0 and 3 # вернет первый ложный объект-операнд
0
>>> 5 and 4 # вернет крайний правый объект-операнд
4
```

Если Python не удается найти ложный объект-операнд, то он возвращает крайний правый операнд.

Логический оператор **or** действует похожим образом, но для объектов-операндов Python возвращает первый объект, имеющий истинное значение. Python прекратит дальнейшие вычисления, как только будет найден первый объект, имеющий истинное значение:

```
>>> 2 or 3 # вернет первый истинный объект-операнд
2
>>> None or 5 # вернет второй объект-операнд,
              # так как первый ложный
5
>>> None or 0 # вернет оставшийся объект-операнд
0
```

Таким образом, конечный результат становится известен еще до вычисления остальной части выражения!

Логические выражения можно комбинировать следующим образом:

```
>>> 1 + 3 > 7 # приоритет + выше, чем >
False
>>> (1 + 3) > 7 # скобки способствуют наглядности
False
>>> 1 + (3 > 7)
1
```

В Python есть возможность проверить принадлежность числа интервалу:

```
>>> x = 0
>>> -5 < x < 10 # эквивалентно: x> -5 and x< 10
True
```

Теперь вы без труда сможете разобраться в работе следующего кода:

```
>>> x = 5 < 10 # True
>>> y = 2 > 3 # False
>>> x or y
True
>>> (x or y) + 6
7
```

Решим небольшую задачку. Как вычислить **1/x**, чтобы не возникало ошибки деления на нуль? Для этого достаточно воспользоваться логическим оператором. Каким? Прямой путь приводит к ошибке:

```
>>> x = 0
>>> 1 / x
Traceback (most recent call last):
  File "<pyshell#88>", line 1, in <module>
    1 / x
ZeroDivisionError: division by zero
```

Решением будет следующий код:

```
>>> x = 1
>>> x and 1/x
1.0
>>> x = 0
>>> x and 1/x
0
```

Строки в Python можно сравнивать аналогично числам. Символы представлены в компьютере в виде чисел. Есть специальная таблица, которая ставит в соответствие каждому символу некоторое число[1]. Определить, какое число соответствует символу, можно с помощью функции **ord**:

```
>>> ord('L')
76
>>> ord('Ф')
1060
>>> ord('A')
65
```

Таким образом, сравнение символов сводится к сравнению чисел, которые им соответствуют:

```
>>> 'A' > 'L'
False
```

Сравнение строк в Python происходит посимвольно:

```
>>> 'Aa' > 'Ll'
False
```

Следующий полезный логический оператор, с которым познакомимся, — **in**. Он принимает две строки и возвращает **True**, если первая строка является подстрокой во второй строке:

```
>>> 'a' in 'abc'
True
>>> 'A' in 'abc' # большой буквы A нет
False
```

[1] Подробнее о Unicode см.: https://docs.python.org/3/howto/unicode.html.

```
>>> "" in 'abc' # пустая строка есть в любой строке
True
>>> '' in ''
True
```

Освоив логические операторы, перейдем к их применению на практике.

Глава 13
УСЛОВНАЯ ИНСТРУКЦИЯ IF

Наиболее часто логические выражения используются внутри условной инструкции **if** (рис. 13.1).

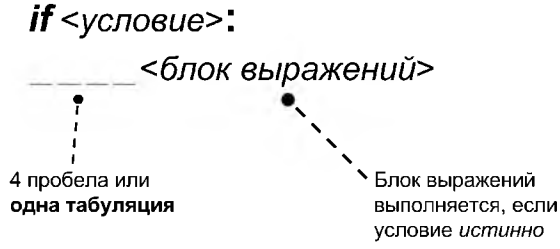


Рис. 13.1. **Схема использования условной инструкции if**

Блок выражений выполняется только в том случае, если выражение, которое находится в условии, является истинным. Для примера обратимся к таблице водородных показателей[1] для различных веществ (табл. 13.1).

Таблица 13.1

Значения pH для различных веществ

Вещество	pH
Яблочный сок	3,0
Кофе	5,0
Шампунь	5,5
Чай	5,5
Питьевая вода	6,5—8,5
Кровь	7,36—7,44
Морская вода	8,0
Мыло (жировое) для рук	9,0—10,0

[1] См.: Wikipedia, Водородный показатель, http://ru.wikipedia.org/wiki/Водородный_показатель.

Произведем проверку:

```
>>> pH = 5.0
>>> if pH == 5.0:
  print(pH, "Кофе")

5.0 Кофе
```

В примере переменной **pH** присваивается вещественное значение 5.0. Затем значение переменной сравнивается с водородным показателем для кофе (5.0), и, если они совпадают, вызывается функция **print**.

Можно производить несколько проверок подряд:

```
>>> pH = 5.0
>>> if pH == 5.0:
  print(pH, "Кофе")

5.0 Кофе
>>> if pH == 8.0:
  print(pH, "Вода")
```

Ход выполнения данной программы показан на рис. 13.2.

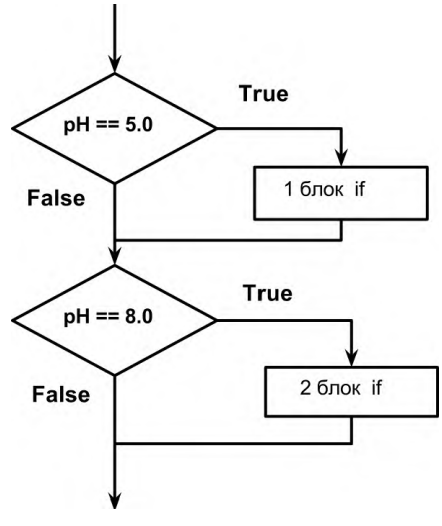


Рис. 13.2. **Ход выполнения программы, содержащей условные инструкции**

На практике встречаются задачи, где выполнять все проверки не имеет смысла. В следующем примере используется инструкция **elif** (сокращение от **else if**):

```
# elif.py
pH = 3.0
if pH == 8.0:
    print(pH, "Вода")
elif 7.36 < pH < 7.44:
    print(pH, "Кровь")
```

Ход выполнения программы показан на рис. 13.3.

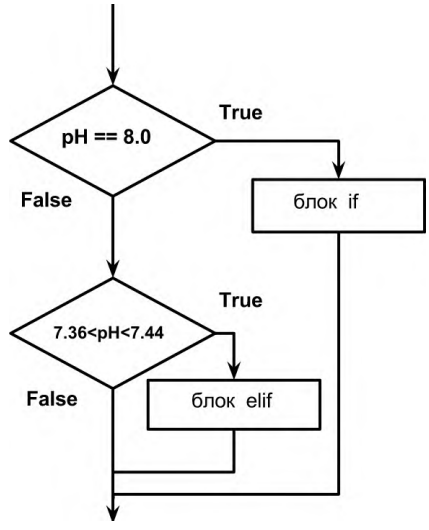


Рис. 13.3. **Ход выполнения программы, содержащей условную инструкцию elif**

Если **pH == 8.0** является ложным, то проверяется **7.36<pH<7.44**, в случае его истинности выполняется блок выражений **elif**. Не существует ограничений на количество инструкций **elif**. Общий синтаксис представлен ниже.

```
if <условие>:
    <блок выражений>
elif <условие>:
    <блок выражений>
else:
    <блок выражений>
```

Блок выражений, относящийся к **else**, выполняется, когда все вышестоящие условия вернули **False**.

Рассмотрим первую «большую» программу:

```
# big_prog.py
# строку преобразовали к вещественному типу
pH = float(input("Введите pH: "))
if pH == 7.0:
    print(pH, "Вода")
elif 7.36 < pH < 7.44:
    print(pH, "Кровь")
else:
    print("Что это?!")
```

В следующем листинге представлен более сложный пример программы, использующей условные инструкции[1]:

```
# big_prog2.py
value = input("Введите pH: ")
```

[1] При наборе исходного текста следите за отступами — в Python это чрезвычайно важно.

```
if len(value) > 0: # проверяем, что пользователь хоть
                   # что-нибудь ввел
    # переводим в вещественное число ввод пользователя:
    pH = float(value)
    if pH == 7.0: # вложенный if
        print(pH, "Вода")
    elif 7.36 < pH < 7.44:
        print(pH, "Кровь")
    else:
        print("Что это?!")
else:
    print("Введите значение pH!")
```

Глава 14
СТРОКИ ДОКУМЕНТАЦИИ

В тройные двойные кавычки (""") в теле функции с обеих сторон помещается информация, которую выводит на экран функция справки **help**.

Рассмотрим пример документирования собственных функций. Для этого напишем функцию, которая ничего не будет делать (в теле функции для этого указывается инструкция **pass**). Исходный текст функции представлен в следующем листинге:

```
# mydoc.py
def my_function():
    """ Функция, которая ничего не делает.
    """
    pass
help(my_function) # отображение описания функции
                  # my_function
```

Результат запуска программы:

```
Help on function my_function in module __main__:

my_function()
    Функция, которая ничего не делает.
```

Глава 15
МОДУЛИ

Предположим, что мы написали несколько полезных функций, которые часто используем в своих программах. Чтобы к ним быстро обращаться, удобно эти функции поместить в отдельный файл и загружать их оттуда. В Python такие файлы с набором функций называются *модулями*. Чтобы воспользоваться функциями, которые находятся в отдельном модуле, его необходимо *импортировать* с помощью инструкции **import**:

```
>>> import math
```

В приведенном ниже примере в память загружается стандартный модуль math, который содержит набор математических функций. Теперь мы можем обращаться к функциям, находящимся внутри этого модуля, следующим образом (например, для нахождения квадратного корня из 9):

```
>>> math.sqrt(9)
3.0
```

При обращении к функции из модуля указывается имя модуля и, через точку, имя функции с аргументами. Узнать о функциях, которые содержит модуль, позволяет функция **help**:

```
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available. It provides access
to the mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
...
```

Если необходимо посмотреть описание отдельной функции модуля, то вызываем **help** для нее:

```
>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

В момент импортирования модуля **math** создается переменная с именем **math**:

```
>>> type(math)
<class 'module'>
```

Функция **type** показала, что тип данных переменной **math** — модуль (**module**). Переменная **math** содержит ссылку (адрес) модульного объекта. В этом объекте содержатся ссылки на функции. Схема работы с объектом типа модуль представлена на рис. 15.1.

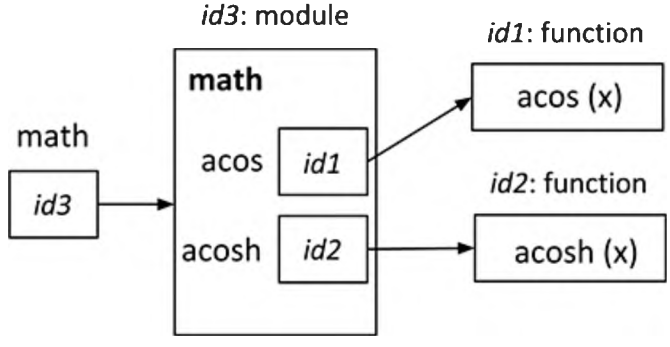


Рис. 15.1. **Схема, представляющая объект типа модуль**

В момент вызова функции **sqrt** Python находит переменную **math** (модуль должен быть предварительно импортирован), просматривает модульный объект, находит функцию **sqrt** внутри этого модуля и затем вызывает ее.

В Python можно импортировать отдельную функцию из модуля:

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

В этом случае переменная **math** создана не будет, а в память загрузится только функция **sqrt**. Теперь вызов функции можно производить, не обращаясь к имени модуля math, но надо быть крайне внимательным. Приведем пример:

```
>>> def sqrt(x):
    return x * x
```

```
>>> sqrt(5)
25
>>> from math import sqrt
>>> sqrt(9)
3.0
```

Объявили собственную функцию с именем **sqrt**, затем вызвали ее и убедились, что она работает. После этого импортировали функцию **sqrt** из модуля **math**. Затем снова вызвали **sqrt** и видим, что это не наша функция! Ее подменили!

Следующий пример:

```
>>> def sqrt(x):
  return x * x

>>> sqrt(6)
36
>>> import math
>>> math.sqrt(9)
3.0
>>> sqrt(7)
49
```

Снова объявляем собственную функцию с именем **sqrt** и вызываем ее. Затем импортируем модуль **math** и из него вызываем стандартную функцию **sqrt**. Видим, что теперь вызываются обе функции.

Когда мы только начинали знакомиться с функциями, то использовали функцию **abs** для нахождения модуля числа. На самом деле эта функция тоже находится в модуле, который Python загружает в память в момент начала своей работы. Этот модуль называется **__builtins__** (два нижних подчеркивания до и после имени модуля). Если вызывать справку для данного модуля, то обнаружим огромное количество функций и переменных:

```
>>> help(__builtins__)
Help on built-in module builtins:

NAME
    builtins - Built-in functions, exceptions, and
    other objects.

DESCRIPTION
    Noteworthy: None is the 'nil' object; Ellipsis
    represents '...' in slices.
...
```

Функция **dir** возвращает перечень имен всех функций и переменных, содержащихся в модуле:

```
>>> dir(__builtins__)
...
'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',
'bytes', 'callable', 'chr', 'classmethod', 'compile',
```

```
'complex', 'copyright', 'credits', 'delattr', 'dict',
'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input', 'int', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'map', 'max',
'memoryview', 'min', 'next', 'object', 'oct', 'open',
'ord', 'pow', 'print', 'property', 'quit', 'range',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'vars', 'zip']
```

Часть из этих функций вы знаете, с другими мы познакомимся в следующих главах.

Глава 16
СОЗДАНИЕ СОБСТВЕННЫХ МОДУЛЕЙ

Рассмотрим процесс создания собственных модулей. Для этого создадим файл с именем **mm.py** (для модулей *обязательно* указывается расширение **.py**), содержащий следующий исходный код (содержимое будущего модуля):

```
# mm.py
def f():
    return 4
```

Затем сообщим Python, где искать созданный модуль. Выясним через обращение к переменной **path** модуля **sys**, где Python по умолчанию хранит модули (у вас список каталогов может отличаться):

```
>>> import sys
>>> sys.path
['', 'C:\\Python36-32\\Lib\\idlelib', 'C:\\Python36-
32\\python36.zip', 'C:\\Python36-32\\DLLs',
'C:\\Python36-32\\lib', 'C:\\Python36-32',
'C:\\Python36-32\\lib\\site-packages']
```

Поместим созданный модуль в один из перечисленных каталогов, например в **'C:\\Python36-32'**. Если все сделано правильно, то импортируем модуль, указав только его имя без расширения:

```
>>> import mm
>>> mm.f() # через точку из модуля mm вызываем функцию f
4
```

Создадим следующий модуль (по аналогии с предыдущим), укажем для него имя **mtest.py**:

```
# mtest.py
print('test')
```

Импортируем созданный модуль несколько раз подряд:

```
>>> import mtest
test
>>> import mtest
```

Заметим, что, во-первых, импорт модуля приводит к выполнению содержащихся в нем инструкций, во-вторых, модуль повторно

не импортируется. Дело в том, что импорт модулей — ресурсоемкий процесс, поэтому лишний раз Python его не производит. В случае внесения изменений в модуль необходимо воспользоваться функцией **reload** из модуля **imp**:

```
>>> import imp
>>> imp.reload(mtest)
test
<module 'mtest' from 'C:\\Python36-32\\mtest.py'>
```

Этот код указывает Python, что модуль требует повторной загрузки. После вызова функции **reload** с указанием в качестве аргумента имени модуля обновленный модуль загрузится повторно.

Продолжим эксперименты. Создадим еще один модуль с именем **mypr.py**:

```
# mypr.py
def func(x):
    return x ** 2 + 7

x = int(input("Введите значение: "))
print(func(x))
```

Импорт модуля приводит к выполнению инструкций, содержащихся внутри модуля:

```
>>> import mypr
Введите значение: 111
12328
```

Каким образом поступить, если необходимо импортировать функцию **func** из модуля для использования ее в другой программе? Чтобы отделить исполнение модуля (Run → Run Module) от его импортирования (**import mypr**), существует специальная переменная **__name__** (имена специальных функций и переменных в Python начинаются и заканчиваются двумя нижними подчеркиваниями).

При выполнении модуля (нажатии *<F5>*) переменная **__name__** будет содержать строку "**__main__**", а в случае импорта — имя модуля (рис. 16.1).

Рис. 16.1. **Разделение импорта и выполнения модуля**

Рассмотрим использование этого приема на практике, для этого создадим модуль с именем **mmtest.py**:

```
# mmtest.py
def func(x):
    return x ** 2 + 7

if __name__ == "__main__":
    x = int(input("Введите значение: "))
    print(func(x))
```

При запуске (Run → Run Module) данного модуля выполнятся все инструкции, содержащиеся в нем, так как условие **__name__ == "__main__"** вернет значение **True**. Импорт модуля (import mmtest) приведет к загрузке в память функции **func**.

Глава 17
АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ ФУНКЦИЙ

В разработке программного обеспечения существует подход (разработка через тестирование, *TDD, Test-Driven Development*), при котором сначала разрабатываются тесты, т.е. сперва описывают, как программа (функция) должна работать, а после этого пишут саму программу (функцию). Это позволяет впоследствии проверить правильность ее написания.

Представим, что мы уже создали функцию **func_m**, которая вычисляет среднее арифметическое, округляя его до трех знаков после запятой. Представим, как бы мы вызвали такую функцию, т.е. напишем тесты:

```
>>> func_m(20, 30, 70)
40.0
>>> func_m(1, 5, 8)
4.667
```

Теперь реализуем функцию **func_m** и в ее описание добавим тесты, проверяющие правильность работы функции. Затем импортируем модуль **doctest** и вызовем функцию **testmod**, которая запустит текущий модуль и проверит, совпадают ли результаты вызовов функций в описании с тем, что получается в реальности. Если результат совпадает, то на экране ничего не появится, в ином случае отобразятся ошибки.

В отдельном файле с именем **test_func.py** создайте и выполните (Run Run Module) следующий код:

```
# test_func.py
def func_m(v1, v2, v3):
    """Вычисляет среднее арифметическое трех чисел.

    >>> func_m(20, 30, 70)
    40.0

    >>> func_m(1, 5, 8)
    4.667
```

```
    """
    return round((v1 + v2 + v3) / 3, 3)

import doctest
# автоматически проверяет тесты в документации
doctest.testmod()
```

В результате выполнения программа ничего не выведет на экран, так как автоматизированное тестирование не выявило ошибок. Теперь исправим тестовые вызовы в программе (**test_func2.py**):

```
# test_func2.py
def func_m(v1, v2, v3):
    """Вычисляет среднее арифметическое трех чисел.

    >>> func_m(20, 30, 70)
    60.0

    >>> func_m(1, 5, 8)
    6.667

    """
    return round((v1 + v2 + v3) / 3, 3)
import doctest
# автоматически проверяет тесты в документации
doctest.testmod()
```

В результате выполнения программы увидим сообщения о возникших ошибках в процессе проверки результатов:

```
**********************************************************
File "C:/Python36-32/test_func2.py", line 4,
in __main__.func_m
Failed example:
    func_m(20, 30, 70)
Expected:
    60.0
Got:
    40.0
**********************************************************
File "C:/Python36-32/test_func2", line 7,
in __main__.func_m
Failed example:
    func_m(1, 5, 8)
Expected:
    6.667
Got:
    4.667
**********************************************************
1 items had failures:
    2 of 2 in __main__.func_m
***TestFailed*** 2 failures.
>>>
```

Глава 18
СТРОКОВЫЕ МЕТОДЫ

Вызовем функцию **type** и передадим ей целочисленный аргумент:

```
>>> type(0)
<class 'int'>
```

Функция сообщила, что целочисленный объект 0 относится к классу **'int'**, т.е. тип данных в Python является *классом*. Для упрощения представим класс как аналог модуля, т.е. набор функций и переменных, содержащихся внутри класса. Функции, которые находятся внутри класса, называются *методами*. Их главное отличие от вызова функций из модуля заключается в том, что в качестве первого аргумента метод принимает, например, строковый объект, если это метод строкового класса.

Рассмотрим пример вызова строкового метода:

```
>>> str.capitalize('hello')
'Hello'
```

По аналогии с вызовом функции из модуля указывается имя класса — **str**, затем через точку — имя строкового метода **capitalize**, который принимает один строковый аргумент (рис. 18.1).

Рис. 18.1. **Полная форма для строковых методов**

Метод — это обычная функция, расположенная внутри класса. Вызовем строковый метод **center**:

```
>>> str.center('hello', 20)
'  hello  '
```

Этот метод принимает два аргумента — строку и число (рис. 18.2).

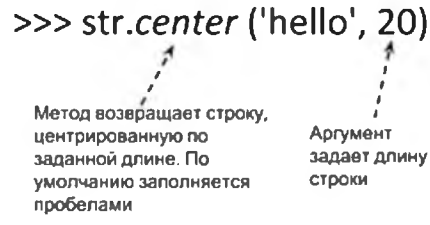


Рис. 18.2. **Полная форма для строковых методов**

Форма вызова метода через обращение к его классу с помощью точки называется *полной формой*, но чаще используют *сокращенную форму вызова метода*:

```
>>> 'hello'.capitalize()
'Hello'
```

Первый аргумент метода поместили на место имени класса (рис. 18.3).

Рис. 18.3. **Сокращенная форма для вызова методов**

Важно отметить, что Python, перед тем как вызвать метод, всегда преобразует сокращенную форму в аналогичную ей полную форму.

Получение справки для метода производится путем вызова функции **help** (вместо имени модуля указывается имя класса):

```
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the
    first character have upper case and the rest lower
    case.
```

Вынесенный из метода первый строковый аргумент может быть выражением, возвращающим строку:

```
>>> ('TTA' + 'G' * 3).count('T')
2
```

Несложно догадаться, что делает метод **count**.

Следующий полезный метод — **format**[1]:

```
>>> '{0} и {1}'.format('труд', 'май')
'труд и май'
```

Вместо {0} и {1} подставятся аргументы метода **format**.

Поменяем их местами:

```
>>> '{1} и {0}'.format('труд', 'май')
'май и труд'
```

Python содержит строковые методы, которые возвращают истину (True) или ложь (False).

Строковый метод **startswith** проверяет, начинается ли строка с символа, переданного в качестве аргумента методу:

```
>>> 'spec'.startswith('a')
False
```

При работе с текстом на практике часто используют строковый метод **strip**, который возвращает строку, очищенную от символа переноса строки (\n) и пробелов:

```
>>> s = '                    \n ssssss \n'
>>> s.strip()
'ssssss'
```

Строковый метод **swapcase** возвращает строку с противоположными регистрами символов:

```
>>> 'Hello'.swapcase()
'hELLO'
```

Вызовы методов в Python можно производить в одну строку:

```
>>> 'ПРИВЕТ'.swapcase().endswith('т')
True
```

В первую очередь вызывается строковый метод **swapcase** для строки **'ПРИВЕТ'**, затем для результирующей строки вызывается метод **endswith** с аргументом **'т'** (рис. 18.4).

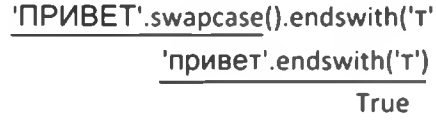


Рис. 18.4. **Порядок вызова методов**

Запустите каждый из перечисленных ниже строковых методов в интерактивном режиме на примере различных строк и разберитесь в принципе их работы.

Предположим, что переменная **s** содержит некоторую строку, тогда к ней применимы следующие методы[2]:

- **s.upper** — возвращает строку в верхнем регистре;

---

[1] Подробнее см. https://docs.python.org/3.1/library/string.html#format-examples.

[2] https://docs.python.org/3/library/stdtypes.html#string-methods.

• **s.lower** — возвращает строку в нижнем регистре;

• **s.title** — возвращает строку, первый символ которой в верхнем регистре;

• **s.find('вет', 2, 3)** — возвращает позицию подстроки в интервале либо –1;

• **s.count('e', 1, 5)** — возвращает количество подстрок в интервале либо –1;

• **s.isalpha** — проверяет, состоит ли строка только из букв;

• **s.isdigit** — проверяет, состоит ли строка только из чисел;

• **s.isupper** — проверяет, написаны ли все символы в верхнем регистре;

• **s.islower** — проверяет, написаны ли все символы в нижнем регистре;

• **s.istitle** — проверяет, начинается ли строка с большой буквы;

• **s.isspace** — проверяет, состоит ли строка только из пробелов.

Выполним объединение двух строк с помощью оператора **+**:

```
>>> 'TT' + 'rr'
'TTrr'
```

При выполнении этого кода Python вызывает специальный строковый метод **__add__** и передает ему в качестве первого аргумента строку **'rr'**:

```
>>> 'TT'.__add__('rr')
'TTrr'
```

Напомним, что этот вызов затем преобразуется в полную форму:

```
>>> str.__add__("TT", 'rr')
'TTrr'
```

Забегая вперед, скажем, что за каждой из операций над типами данных скрывается свой специальный метод.

Глава 19
СПИСКИ

19.1. Создание списка

Предположим, что необходимо обработать информацию о курсах валют[1] (рис. 19.1).

Дата	Доллар США USD	ЕВРО EUR
30.05.2015	52.9716	58.0145
29.05.2015	52.2907	57.1433
28.05.2015	51.0178	55.6757
27.05.2015	50.3223	54.8412
26.05.2015	49.8613	54.7477
23.05.2015	49.7901	55.5508
22.05.2015	49.9204	55.5714

Рис. 19.1. **Информация о курсе валют**

Курс валюты на каждый день можно поместить в отдельную переменную:

```
>>> day1 = 56.8060
>>> day2 = 57.1578
```

Схематично подобная работа с переменными представлена на рис. 19.2.

Как поступить, если необходимо обработать курсы валют за последние два года? В таком случае на помощь приходят *списки*. Их можно рассматривать как аналог массива в других языках программирования, за исключением важной особенности — списки в качестве своих элементов могут содержать объекты различных типов.

Список (*list*) в Python является объектом, поэтому может быть присвоен переменной (напоминаем, что в переменной хранится адрес объекта).

---

[1] URL: http://www.sberometer.ru/cbr/.

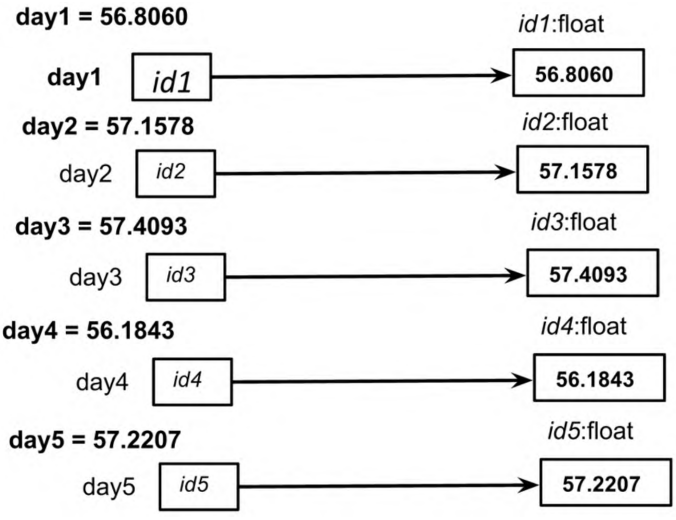


Рис. 19.2. **Схема памяти Python при работе с переменными**

Представим список для задачи с курсом валют:

```
>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
```

Список позволяет хранить разнородные данные, обращаться к которым можно через имя списка (в данном случае переменную **e**). Рассмотрим схематично, как Python работает со списками (рис. 19.3).

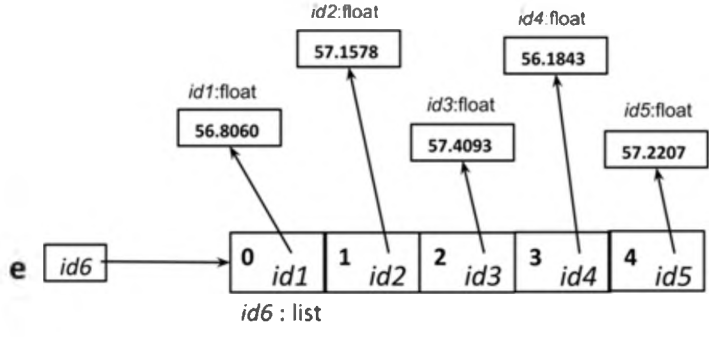


Рис. 19.3. **Схема памяти Python при работе со списком**

Переменная **e** содержит адрес списка (*id*6), каждый элемент списка содержит указатель (адрес) объекта. В общем виде создание списка показано на рис. 19.4.

На месте элементов списка могут находиться сложные выражения.

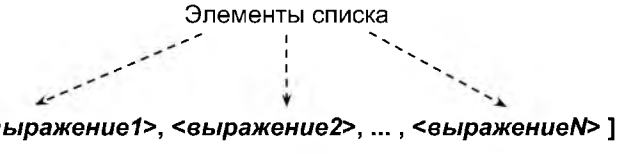


Рис. 19.4. **Общая форма создания списка**

19.2. Операции над списками

Обращаться к отдельным элементам списка можно по их *индексу* (позиции), начиная с нуля (по аналогии со строками):

```
>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e[0]
56.806
>>> e[1]
57.1578
>>> e[-1] # последний элемент
57.2207
```

Обращение по несуществующему индексу приведет к ошибке **IndexError**:

```
>>> e[100]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    e[100]
IndexError: list index out of range
```

До настоящего момента мы рассматривали неизменяемые типы данных (классы). Вспомните, как Python ругался при попытке изменить строку. Проведем эксперимент по изменению списка:

```
>>> h = ['Hi', 27 , -8.1, [1,2]] # создание списка
>>> h[1] = 'hello' # изменение элемента списка в позиции 1
>>> h # результирующий список
['Hi', 'hello', -8.1, [1, 2]]
>>> h[1] # измененный элемент списка
'hello'
```

На рис. *19.5* представлена схема памяти в момент создания списка **h**.

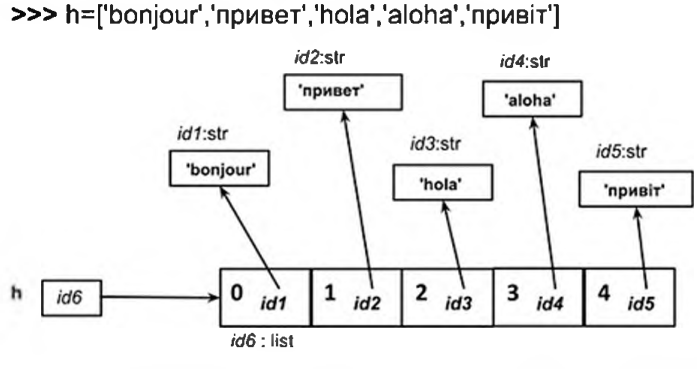


Рис. 19.5. **Схема памяти Python при работе со списком**

Произведем изменение списка:

```
>>> h[1] = 'hello'
>>> h
['bonjour', 'hello', 'hola', 'aloha', 'привіт']
>>> h[1]
'hello'
```

Схема памяти в момент изменения списка показана на рис. *19.6*.

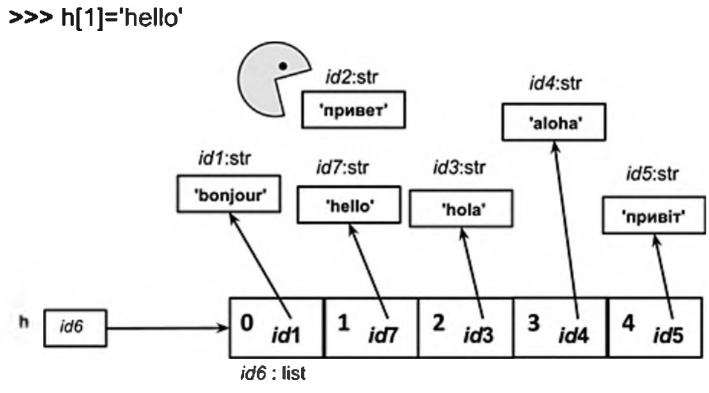


Рис. 19.6. **Схема памяти Python в момент изменения списка**

Видим, что в памяти создается новый строковый объект **'hello'**. Затем адрес этого объекта (*id7*) помещается в первую ячейку списка (вместо адреса *id2*). Python обнаружит, что на объект по адресу *id2* больше нет ссылок, поэтому удалит его из памяти (произведет автоматическую сборку мусора).

Список, наверное, наиболее часто встречающийся тип данных, с которым приходится сталкиваться при написании программ. Это связано со встроенными в Python функциями, которые позволяют легко и быстро обрабатывать списки:

- **len(**L**)** — возвращает число элементов в списке L;
- **max(**L**)** — возвращает максимальное значение в списке L;
- **min(**L**)** — возвращает минимальное значение в списке L;
- **sum(**L**)** — возвращает сумму значений в списке L;
- **sorted(**L**)** — возвращает копию списка L, в котором элементы упорядочены по возрастанию. Не изменяет список L.

Примеры вызовов функций для работы со списками:

```
>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
>>> len(e)
5
>>> max(e)
57.4093
>>> min(e)
56.1843
>>> sum(e)
284.7781
>>> sorted(e)
[56.1843, 56.806, 57.1578, 57.2207, 57.4093]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
```

Оператор «+», примененный к спискам, служит для их объединения (вспомните строки):

```
>>> original = ['H','B']
>>> final = original + ['T']
>>> final
['H', 'B', 'T']
```

Операция повторения (вновь возникает аналогия со строками):

```
>>> final = final * 5
>>> final
['H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T', 'H', 'B',
'T', 'H', 'B', 'T']
```

Инструкция **del** позволяет удалять из списка элементы по указанному индексу (это инструкция, поэтому скобки не требуются):

```
>>> del final[0]
>>> final
['B', 'T', 'H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T',
'H', 'B', 'T']
```

Рассмотрим интересный пример. Создадим функцию **f**, объединяющую два списка:

```
>>> def f(x, y):
  return x + y

>>> f([1, 2, 3], [4, 5, 6])
[1, 2, 3, 4, 5, 6]
```

Теперь передадим в качестве аргументов функции **f** две строки:

```
>>> f("123", "456")
'123456'
```

Далее передадим в качестве аргументов функции **f** два числа:

```
>>> f(1, 2)
3
```

В итоге небольшая функция умеет объединять или складывать переданные ей объекты в зависимости от их класса (типа данных).

Оператор **in** по аналогии со строками применим к списку:

```
>>> h = ['bonjour', 7, 'hola', -1.0, 'привіт']
>>> if 7 in h:
  print('Значение есть в списке')

Значение есть в списке
```

Аналогично строкам, для списка существует оператор извлечения среза:

```
>>> h = ['bonjour', 7, 'hola', -1.0, 'привіт']
>>> h
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> g = h[1:2]
>>> g
[7]
```

Схема памяти применительно к срезам представлена на рис. 19.7.

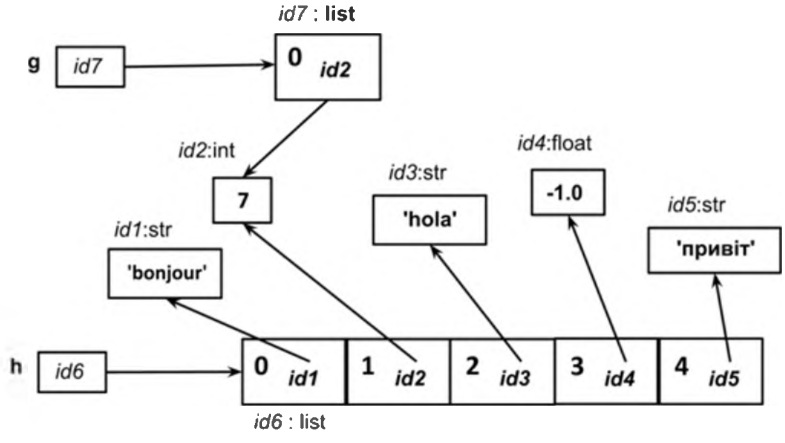


Рис. 19.7. **Схема памяти Python при работе со срезами**

Переменной **g** присваивается адрес нового списка (*id7*), содержащего указатель на числовой объект, выбранный с помощью среза.

Вернемся к инструкции **del** и удалим с помощью среза подсписок:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]    # удаление подсписка
>>>a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

19.3. Псевдонимы и копирование списков

Рассмотрим особенность, связанную с изменяемостью спискового типа данных, для этого выполним следующие инструкции:

```
>>> h
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> p = h    # содержат указатель на один и тот же
             # список
>>> p
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> p[0] = 1      # модифицируем одну из переменных
>>> h          # изменилась другая переменная!
[1, 7, 'hola', -1.0, 'привіт']
>>> p
[1, 7, 'hola', -1.0, 'привіт']
```

На схеме, представленной на рис. 19.8, видно, что переменные **p** и **h** указывают на один и тот же список, т.е. являются псевдонимами.

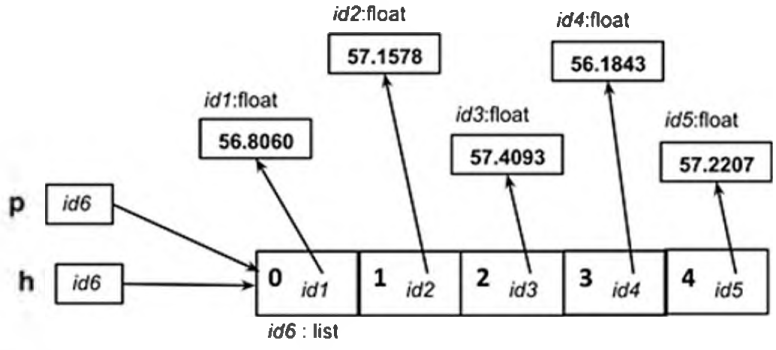


Рис. 19.8. **Схема памяти Python при работе с псевдонимами**

Возникает вопрос, как проверить, ссылаются ли переменные на один и тот же список:

```
>>> x = y = [1, 2] # создали псевдонимы
>>> x is y # ссылаются ли переменные на один и тот же
           # объект
True
>>> x = [1, 2]
>>> y = [1, 2]
>>> x is y
False
```

К спискам применимы два вида копирования. Первый вид – *поверхностное копирование*, при котором создается новый объект, но он будет заполнен ссылками на элементы, которые содержались в оригинале:

```
>>> a = [4, 3, [2, 1]]
>>> b = a[:]
>>> b is a
False
>>> b[2][0] = -100
>>> a
[4, 3, [-100, 1]] # список a тоже изменился
>>>
```

Второй вид копирования — *глубокое копирование*. При глубоком копировании создается новый объект и рекурсивно создаются копии всех объектов, содержащихся в оригинале:

```
>>> import copy
>>> a = [4, 3, [2, 1]]
>>> b = copy.deepcopy(a)
>>> b[2][0] = -100
>>> a
[4, 3, [2, 1]] # список a не изменился
>>>
```

С одной стороны, список предоставляет возможность модификации, с другой — появляется опасность повлиять на псевдоним списка.

19.4. Методы списка

Работа с методами списка похожа на работу со строковыми методами[1]. Далее приведены наиболее популярные методы списка[2]:

```
>>> colors = ['red', 'orange', 'green']
>>> colors.extend(['black','blue']) # расширяет
                                    # список списком
>>> colors
['red', 'orange', 'green', 'black', 'blue']

>>> colors.append('purple') # добавляет элемент в список
>>> colors
['red', 'orange', 'green', 'black', 'blue', 'purple']

>>> colors.insert(2,'yellow')  # добавляет элемент
                               # в указанную позицию
>>> colors
['red', 'orange', 'yellow', 'green', 'black', 'blue',
'purple']

>>> colors.remove('black')  # удаляет элемент из списка
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']

>>> colors.count('red') # количество повторений
                        # аргумента метода
1

>>> colors.index('green') # возвращает позицию в списке
3

>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']

>>> colors.pop() # удаляет и возвращает последний
                 # элемент списка
'purple'

>>> colors
['red', 'orange', 'yellow', 'green', 'blue']

>>> colors.reverse() # список в обратном порядке
>>> colors
['blue', 'green', 'yellow', 'orange', 'red']

>>> colors.sort() # сортирует список (вспомните
                  # о сравнении строк)
```

---

[1] Только вместо класса **str** используется **list**, а в качестве первого аргумента в полной форме метод принимает список.

[2] Подробнее см. https://docs.python.org/3/tutorial/datastructures.html#more-on-lists.

```
>>> colors
['blue', 'green', 'orange', 'red', 'yellow']

>>> colors.clear() # или del color[:], очищает список,
                   # с версии Python 3.3
>>> colors
[]
```

19.5. Преобразование типов

На практике довольно часто возникает необходимость в изменении строк, но напрямую мы этого сделать не можем (строки относятся к неизменяемому типу данных). Тогда на помощь приходят списки. Решением может стать преобразование строки в список, изменение списка и обратное преобразование списка в строку:

```
>>> s = 'Строка для изменения'
>>> list(s) # функция list пытается преобразовать
            # аргумент в список
['С', 'т', 'р', 'о', 'к', 'а', ' ', 'д', 'л', 'я', ' ',
'и', 'з', 'м', 'е', 'н', 'е', 'н', 'и', 'я']
>>> lst = list(s)
>>> lst[0] = 'М'  # изменяем список, полученный из стро-
ки
>>> lst
['М', 'т', 'р', 'о', 'к', 'а', ' ', 'д', 'л', 'я', ' ',
'и', 'з', 'м', 'е', 'н', 'е', 'н', 'и', 'я']
>>> s = ''.join(lst)  # преобразуем список в строку
>>> s
'Мтрока для изменения'
```

Отдельно остановимся на использовании *строкового* метода **join**:

```
>> A = ['red', 'green', 'blue']
>>> ' '.join(A)
'red green blue'
>>> ''.join(A)
'redgreenblue'
>>> '***'.join(A)
'red***green***blue'
```

Строковый метод **join** принимает в качестве аргумента список, который необходимо преобразовать в строку, а в качестве строкового объекта указывается соединитель элементов списка.

Похожим образом можно преобразовать число в список (на промежуточном этапе используется строка) и затем изменить полученный список:

```
>>> n = 73485384753846538465
>>> list(str(n)) # число преобразуем в строку, затем
                 # строку в список
['7', '3', '4', '8', '5', '3', '8', '4', '7', '5', '3',
```

```
'8', '4', '6', '5', '3', '8', '4', '6', '5']
```

Если строка содержит разделитель, то ее можно преобразовать в список с помощью строкового метода **split**, который по умолчанию в качестве разделителя использует пробел:

```
>>> s = 'd a dd dd gg rr tt yy rr ee'.split()
>>> s
['d', 'a', 'dd', 'dd', 'gg', 'rr', 'tt', 'yy', 'rr', 'ee']
```

Вызов метода **split** с разделителем ":":

```
>>> s = 'd:a:dd:dd:gg:rr:tt:yy:rr:ee'.split(":")
>>> s
['d', 'a', 'dd', 'dd', 'gg', 'rr', 'tt', 'yy', 'rr', 'ee']
```

19.6. Вложенные списки

Ранее уже упоминалось, что в качестве элементов списка могут находиться объекты любого типа (класса), например списки:

```
>>> lst = [['A', 1], ['B',2], ['C',3]]
>>> lst
[['A', 1], ['B', 2], ['C', 3]]
>>> lst[0]
['A', 1]
```

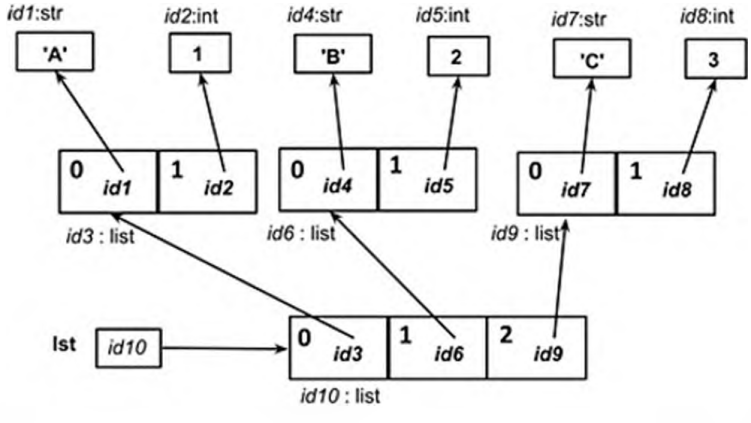


Рис. 19.9. **Схема памяти при работе Python с вложенными списками**

Подобные структуры данных используются для хранения матриц. Обращение к вложенному списку происходит через указание двух индексов (индекс внешнего списка и индекс вложенного списка):

```
>>> lst[0][1]
1
```

Схематично вложенные списки показаны на рис. 19.9.

Глава 20
ИТЕРАЦИИ

Язык программирования Python позволяет в кратчайшие сроки создавать прототипы[1] реальных программ благодаря тому, что в него заложены конструкции для решения типовых задач, с которыми часто приходится сталкиваться программисту.

Вспомните, как мы решали задачу подсчета суммы элементов списка через вызов единственной функции **sum([1, 4, 5, 6, 7.0, 3, 2.0])**.

Рассмотрим еще несколько подобных приемов, которые значительно упрощают жизнь разработчика на языке Python.

20.1. Инструкция for

К примеру, мы хотим вывести на экран каждый из элементов списка **num**:

```
>>> num = [0.8, 7.0, 6.8, -6]
>>> num
[0.8, 7.0, 6.8, -6]
>>> print(num[0],'- number')
0.8 - number
>>> print(num[1],'- number')
7.0 - number
```

Теперь представим, что в списке пятьсот элементов... Для подобных случаев в Python существуют *циклы*. Перепишем этот пример с использованием инструкции **for**:

```
>>>num = [0.8, 7.0, 6.8, -6]
>>>for i in num:
print(i, '- number')

0.8 - number
7.0 - number
6.8 - number
-6 - number
```

[1] Быстрая, черновая реализация будущей программы.

Цикл **for** позволяет обратиться к каждому из элементов указанного списка. Имя переменной, в которую на каждом шаге будет помещаться элемент списка, выбирает программист. В нашем примере это переменная с именем **i**. На первом шаге переменной **i** присваивается первый элемент списка **num**, равный 0.8. Затем программа переходит в тело цикла **for**, отделенное отступами (четыре пробела или одна табуляция). В теле цикла содержится вызов функции **print**, которой передается на вход переменная **i**. На следующем шаге (итерации цикла) переменной **i** будет присвоен второй элемент списка, равный 7.0. Произойдет вызов функции **print** для отображения текущего содержимого переменной **i** на экране и т.д. Тело цикла выполняется до тех пор, пока не будет достигнут конец списка.

В общем виде инструкция **for** для обращения к каждому из элементов указанного списка выглядит так:

```
for <переменная> in <список> :
    <тело цикла>
```

Например:

```
>>> for i in [1, 2, 'hi']:
    print(i)

1
2
hi
```

Инструкция **for** схожим образом работает для строк:

```
>>> for i in 'hello':
    print(i)

h
e
l
l
o
```

По аналогии со списком для строки происходит обращение к каждому из символов, пока не будет достигнут конец строки. В общем виде запись инструкции цикла **for** для заданной строки выглядит так:

```
for <переменная> in <строка> :
    <тело цикла>
```

Инструкция **for** позволяет производить операции над отдельными элементами списка или строки (в общем случае *последовательности*):

```
>>> num = [0.8, 7.0, 6.8, -6]
>>> for i in num:
  if i == 7.0:
    print (i, '- число 7.0')

7.0 - число 7.0
```

В примере условная инструкция **if** позволила вывести на экран только указанное значение из списка, выполнив в теле цикла сравнение с каждым элементом списка.

Похожим образом с помощью инструкции цикла производится поиск символа в строке:

```
>>> country = "Russia"
>>> for ch in country:
  if ch.isupper():
    print (ch)

R
```

20.2. Функция range

Достаточно часто на практике при разработке программ необходимо получить *диапазон* целых чисел (рис. 20.1).

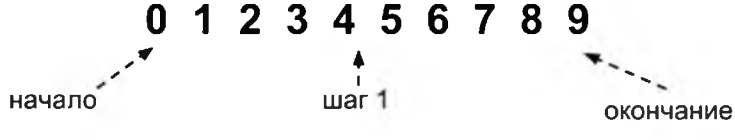


Рис. 20.1. **Диапазон целых чисел**

Для решения этой задачи в Python предусмотрена функция **range**[1]. В качестве аргументов функция принимает начальное значение диапазона (по умолчанию 0), конечное значение (*не включительно*) и шаг (по умолчанию 1). Если вызвать функцию **range** в интерактивном режиме, то диапазона чисел мы не увидим[2]:

```
>>> range(0,10,1)
range(0, 10)
>>> range(10)
range(0, 10)
```

Для создания диапазона (неизменяемой последовательности) необходимо использовать, например, инструкцию **for**:

```
>>> for i in range(0, 10, 1):
    print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

[1] Подробнее см. https://docs.python.org/3.6/library/stdtypes.html#range.

[2] Функция **range** вернула объект типа **range**. Интересно, что **range** для экономии места хранит в памяти только начало, окончание и шаг диапазона.

```
>>> for i in range(10):
    print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
>>> for i in range(2, 20, 2):
    print(i, end=' ')

2 4 6 8 10 12 14 16 18
```

Таким образом, в переменную **i** на каждом шаге цикла (итерации) будет записываться значение из диапазона, который генерируется функцией **range**.

При желании можно получить диапазон в обратном порядке следования (обратите внимание на аргументы функции **range**):

```
>>> for i in range(20, 2, -2):
    print(i, end=' ')

20 18 16 14 12 10 8 6 4
```

Теперь с помощью диапазона найдем сумму чисел на интервале от 1 до 100:

```
>>> total = 0
>>> for i in range(1, 101):
    total = total + i

>>> total
5050
```

Переменной **i** на каждой итерации цикла последовательно присваиваются значения из диапазона от 1 до 100 (напоминаем, что крайнее правое значение не включается). На первой итерации переменная **total** содержит значение 0 (инициализировали **total** перед входом в цикл). В теле цикла сначала вычисляется правая часть инструкции присваивания, т.е. **total + i**. Переменная **i** на первом шаге содержит значение 1 (первое значение из диапазона), таким образом, правая часть инструкции присваивания будет равна значению 1. Это значение помещается в левую часть, т.е. присваивается переменной **total**. На втором шаге **total** уже будет содержать значение 1, **i** — 2, т.е. правая часть инструкции присваивания станет равна 3, далее это значение помещается в переменную **total** и т.д., до конца диапазона, т.е. до значения 100. По окончании цикла в **total** будет содержаться искомая сумма чисел.

В Python возможно более элегантное решение данной задачи в функциональном стиле:

```
>>> sum(list(range(1, 101)))
5050
```

Это решение требует небольших пояснений. Диапазоны можно использовать при создании списков:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2, 10, 2))
[2, 4, 6, 8]
```

Вызов функции **sum** для списка в качестве аргумента приводит к вычислению суммы элементов списка.

Диапазон, создаваемый функцией **range**, на практике часто используется для задания *индексов*. Следующий пример демонстрирует изменение списка путем умножения каждого из его элементов на 2:

```
# range_two.py
lst = [4, 10, 5, -1.9]
print(lst)
for i in range(len(lst)):
    lst[i] = lst[i] * 2
print(lst)
```

В качестве аргумента функции **range** в примере задается длина списка. В этом случае создаваемый диапазон будет от 0 до **len(lst) - 1**. Python не включает крайний правый элемент диапазона, так как длина списка всегда на 1 больше, чем индекс последнего его элемента (напоминаем, индексация списка начинается с нуля).

В результате выполнения программы:

```
[4, 10, 5, -1.9]
[8, 20, 10, -3.8]
```

20.3. Создание списка

Первым рассмотрим пример создания списка с помощью метода **append**:

```
>>> a = []
>>> for i in range(1,15):
        a.append(i)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Инструкция **for** позволяет последовательно из диапазона от 1 до 14 выбрать числа и с помощью метода **append**, находящегося в теле цикла, добавить их к списку **a**.

Следующий пример — создание списка из диапазона с помощью функции **list**:

```
>>> a = list(range(1, 15))
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Python поддерживает отдельные возможности функционального стиля программирования, который строится исключительно на вызове функций и позволяет компактно записывать команды. Одна из таких возможностей — «списковое включение» (*list comprehension*, иначе «списочное встраивание»). Рассмотрим его применительно к созданию списка:

```
>>> a = [i for i in range(1,15)]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Правила работы со списковым включением представлены на рис. 20.2.

Рис. 20.2. **Схема использования спискового включения**

В следующем примере выбираем из диапазона все числа от 1 до 14, возводим их в квадрат и «на лету» формируем из них новый список:

```
>>> a = [i**2 for i in range(1, 15)]
>>> a
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

Списковое включение позволяет задавать условие для выбора значения из диапазона (в следующем примере исключили значение 4):

```
>>> a = [i**2 for i in range(1, 15) if i != 4]
>>> a
[1, 4, 9, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

Вместо диапазонов, генерируемых функцией **range**, можно указывать существующий список:

```
>>> a = [2, -2, 4, -4, 7, 5]
>>> b = [i**2 for i in a]
>>> b
[4, 4, 16, 16, 49, 25]
```

В примере последовательно выбираются значения из списка **a**, каждый из его элементов возводится в квадрат и «на лету» добавляется в новый список.

По аналогии со списками можно последовательно перебирать символы из строки и формировать из них список (в следующем примере исключили символ **i**):

```
>>> c = [c*3 for c in 'list' if c != 'i']
>>> c
['lll', 'sss', 'ttt']
```

Функция **map**[1] позволяет, используя функциональный стиль программирования, создавать новый список на основе существующего:

```
>>> def f(x):
    return x + 5

>>> list(map(f, [1, 3, 4]))
[6, 8, 9]
```

В примере функция **map** принимает в качестве аргументов имя функции **f** и список (или строку). В процессе вызова функции **map** каждый элемент указанного списка (или строки) подается на вход функции **f**, и результат вызова функции **f** для каждого из элементов указанного списка «на лету» добавляется как элемент нового списка. Функция **map** возвращает объект типа **map**, поэтому получить итоговый список можно с помощью инструкции **for** либо функции **list**.

Пример вызова функции **map** для строки:

```
>>> def f(s):
    return s * 2

>>> list(map(f, "hello"))
['hh', 'ee', 'll', 'll', 'oo']
```

Функциональные возможности Python позволяют определять небольшие однострочные функции на месте вызова функции (исторически такие функции получили название λ-функций):

```
>>> list(map(lambda s: s*2, "hello"))
['hh', 'ee', 'll', 'll', 'oo']
```

Далее рассмотрим генерацию списка, состоящего из случайных целых чисел:

```
>>> from random import randint
>>> A = [randint(1, 9) for i in range(5)]
>>> A
[2, 1, 1, 7, 8]
```

В данном примере функция **range** выступает как счетчик количества элементов создаваемого списка. Обратите внимание, что при формировании нового списка переменная **i** не используется. В результате пять раз будет произведен вызов функции **randint**[2], которая сгенерирует целое псевдослучайное число из интервала от 1 до 9, и уже это число добавится в новый список.

Сформируем список, значения для которого будем задавать с клавиатуры:

---

[1] Подробнее см.: https://docs.python.org/3.6/library/functions.html#map.

[2] Подробнее см.: https://docs.python.org/3.6/library/random.html.

```
# input_list.py
a = [] # объявляем пустой список
n = int(input()) # указываем количество элементов списка
for i in range(n): # функция range контролирует длину
                   # списка
    new_element = int(input()) # указываем очередной
                               # элемент списка
    a.append(new_element) # добавляем элемент в список
    # последние две строки можно было заменить одной:
    # a.append(int(input()))
print(a)
```

Результат выполнения программы:

```
3
4
2
1
[4, 2, 1]
```

Перепишем программу в одну строку, используя возможности спискового включения:

```
>>> A = [int(input()) for i in range(int(input()))]
3
4
2
1
>>> A
[4, 2, 1]
```

20.4. Инструкция while

Инструкция **for** используется, если заранее известно, сколько итераций необходимо выполнить (указывается через аргумент функции **range** или пока не закончится список/строка). Если заранее количество итераций цикла неизвестно, то применяется инструкция цикла **while**. Тело цикла **while** выполняется до тех пор, пока выражение является истинным или не произошел выход из цикла с помощью инструкции **break**.

```
while <выражение>:<тело цикла>
```

Далее приведен пример программы подсчета числа кроликов с использованием инструкции цикла **while**:

```
# while_rabbits.py
rabbits = 3
while rabbits > 0:
    print(rabbits)
    rabbits = rabbits - 1
```

Инструкция **while** выполняется до тех пор, пока число кроликов в условии положительное (**rabbits > 0**). На каждой итерации цикла

переменная **rabbits** уменьшается на 1, чтобы не получился бесконечный цикл, при котором условие всегда будет оставаться истинным. В начале работы программы инициализируется переменная **rabbits** (переменной присваивается начальное значение 3), затем происходит переход в тело цикла **while**, так как условие **rabbits** > 0 является истинным (вернет значение **True**). Далее в теле цикла вызывается функция **print**, которая отобразит на экране текущее значение переменной **rabbits**. После этого переменная **rabbits** уменьшится на 1 и снова произойдет проверка условия 2 > 0 (вернет **True**), перейдем в тело цикла и т.д., пока не дойдем до условия 0 > 0. В этом случае вернется логическое значение **False** и произойдет выход из инструкции **while**. В результате выполнения программы:

```
3
2
1
```

Бесконечный цикл, организованный с помощью инструкции **while**, позволяет производить обработку введенных с клавиатуры строк:

```
# while_input.py
text = ""
while True:
    text = input("Введите число или стоп для выхода: ")
    if text == "стоп":
        print("Выход из программы! До встречи!")
        break    # инструкция выхода из цикла
    elif text == '1':
        print("Число 1")
    else:
        print("Что это?!")
```

Результат выполнения программы имеет следующий вид:

```
Введите число или стоп для выхода: 4
Что это?!
Введите число или стоп для выхода: 1
Число 1
Введите число или стоп для выхода: стоп
Выход из программы! До встречи!
```

Тело цикла **while** выполняется бесконечное число раз, так как **True** всегда является истиной. Выход из цикла осуществляется по инструкции **break**, если пользователь введет с клавиатуры строку «стоп».

Следующий исходный текст демонстрирует пример подсчета суммы чисел, встречающихся в строке:

```
# total_num.py
s = 'aa3aBbb6ccc'
total = 0
```

```
for i in range(len(s)):
    if s[i].isalpha(): # посимвольно проверяем наличие
                       # буквы
        continue # инструкция перехода к следующему
                 # шагу цикла
    total = total + int(s[i]) # накапливаем сумму,
                              # если встретилась цифра

print("сумма чисел:", total)
```

В примере используется инструкция **continue**. Выполнение данной инструкции приводит к переходу на следующий шаг цикла, т.е. все команды, которые находятся после **continue**, будут проигнорированы. Далее представлен результат выполнения программы:

```
сумма чисел: 9
```

20.5. Вложенные циклы

Python позволяет вкладывать инструкции циклов друг в друга так, как показано в следующем примере:

```
# for_outer_inner.py
outer = [1, 2, 3, 4]
inner = [5, 6, 7, 8]
for i in outer: # внешний цикл
    for j in inner: # вложенный (внутренний) цикл
        print('i=', i, 'j=', j)
```

В примере внешний цикл **for** перебирает все элементы списка **outer** (на первой итерации внешнего цикла фиксируется **i** = 1), затем управление передается вложенному циклу **for**, который проходит по всем элементам списка **inner** (изменяется переменная **j**). После того, как закончатся элементы в списке **inner**, управление вновь возвращается внешнему циклу (фиксируется следующее значение **i** = 2), после этого вложенный цикл проходит по всем элементам списка **inner**. Так повторяется до тех пор, пока не закончатся элементы в списке **outer** (рис. 20.3).

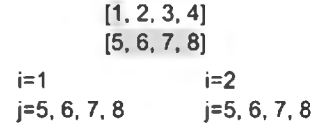


Рис. 20.3. **Пример выполнения вложенных циклов**

Результат выполнения программы:

```
i= 1 j= 5
i= 1 j= 6
i= 1 j= 7
i= 1 j= 8
```

```
i= 2 j= 5
i= 2 j= 6
i= 2 j= 7
i= 2 j= 8
i= 3 j= 5
i= 3 j= 6
i= 3 j= 7
i= 3 j= 8
i= 4 j= 5
i= 4 j= 6
i= 4 j= 7
i= 4 j= 8
```

Рассмотрим пример работы с вложенными списками (см. п. 19.6):

```
## for_lst.py
lst = [[1, 2, 3],
       [4, 5, 6]]
for i in lst:
    print(i)
```

Результат выполнения программы:

```
[1, 2, 3]
[4, 5, 6]
```

В примере с помощью инструкции **for** перебираются все элементы списка, которые также являются списками. Чтобы добраться до элементов вложенных списков, необходимо воспользоваться вложенной инструкцией **for**:

```
# for_lst2.py
lst = [[1, 2, 3],
       [4, 5, 6]]

for i in lst:     # внешний цикл
    print()
    for j in i:  # вложенный цикл
        print(j, end = "")
```

Результат выполнения программы:

```
123
456
```

Глава 21
МНОЖЕСТВА

Множество **(set)** в языке Python — неупорядоченная коллекция неизменяемых уникальных элементов:

```
>>> v = {'A', 'C', 4, '5', 'B'}
>>> v
{'C', 'B', '5', 4, 'A'}
```

Отметим, что порядок элементов созданного множества отличается от порядка элементов множества, отображенного на экране, так как множество — это *неупорядоченная коллекция*. На рис. 21.1 множество представлено схематично.

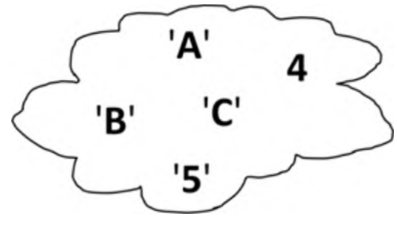


Рис. 21.1. **Множество в языке Python**

Множества в Python обладают интересными свойствами:

```
>>> v = {'A', 'C', 4, '5', 'B', 4}
>>> v
{'C', 'B', '5', 4, 'A'}
```

Видим, что повторяющиеся элементы, которые мы добавили при создании множества, были удалены (выполняется свойство *уникальности* элементов множества). Перейдем к рассмотрению способов создания множества:

```
>>> set([3, 6, 3, 5])
{3, 5, 6}
```

В примере множество было создано на основе списка. Обратите внимание, что в момент создания множества из списка будут удалены повторяющиеся элементы. Это легкий способ очистить список от повторов:

```
>>> list(set([3, 6, 3, 5]))
[3, 5, 6]
```

Функция **range** позволяет создавать множества из диапазона:

```
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Рассмотрим некоторые операции над множествами[1]:

```
>>> s1 = set(range(5))
>>> s2 = set(range(2))
>>> s1
{0, 1, 2, 3, 4}
>>> s2
{0, 1}
>>> s1.add('5') # добавление элемента
>>> s1
{0, 1, 2, 3, 4, '5'}
```

У множеств в языке Python много общего с математическими множествами (рис. 21.2):

```
>>> s1.intersection(s2) # пересечение множеств (s1 & s2)
{0, 1}
>>> s1.union(s2)        # объединение множеств (s1 | s2)
{0, 1, 2, 3, 4, '5'}
```

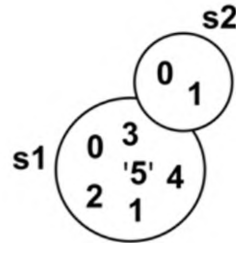


Рис. 21.2. **Операции над множествами**

[1] Подробнее см.: https://docs.python.org/3/tutorial/datastructures.html#sets.

Глава 22
КОРТЕЖИ

Кортеж (**tuple**) в Python схож по своим свойствам со списком за исключением изменяемости. Кортежи используются, когда необходимо быть уверенным, что элементы структуры данных не будут изменены в процессе работы программы. Вспомните проблему с псевдонимами (см. п. 19.3).

Рассмотрим некоторые популярные операции над кортежами[1]:

```
>>> ()      #  создание пустого кортежа
()
>>> (4)     # это не кортеж, а целочисленный объект!
4
>>> (4,)    # а вот это - кортеж, состоящий из одного
            # элемента!
(4,)
>>> b = ('1', 2, '4')    # создаем кортеж
>>> b
('1', 2, '4')
>>> len(b)  # определяем длину кортежа
3
>>> t = tuple(range(10)) # создание кортежа с помощью
                         # функции range
>>> t + b    # слияние кортежей
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, '1', 2, '4')
>>> r = tuple([1, 5, 6, 7, 8, '1']) # кортеж из списка
>>> r
(1, 5, 6, 7, 8, '1')
```

С помощью кортежей можно присваивать значения одновременно двум переменным:

```
>>> (x, y) = (10, 5)
>>> x
10
>>> y
5
>>> x, y = 1, 3 # убрали круглые скобки
```

[1] Подробнее см.: https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences.

```
>>> x
1
>>> y
3
```

Поменять местами содержимое двух переменных можно следующим образом:

```
>>> x, y = y, x
>>> x
3
>>> y
1
```

Кортеж изменить нельзя, но можно изменить, например, список, входящий в кортеж:

```
>>> t = (1, [1,3], '3')
>>> t[1]
[1, 3]
>>> t[1][0] = '1'
>>> t
(1, ['1', 3], '3')
```

Глава 23
СЛОВАРИ

Словарь (**dict**) в Python — неупорядоченная изменяемая коллекция или, проще говоря, «список» с произвольными ключами, неизменяемого типа.

Рассмотрим пример создания словаря, который каждому слову на английском языке (*множество ключей словаря*) ставит в соответствие слово на испанском языке (*множество значений словаря*). Схематично такой словарь представлен на рис. 23.1.

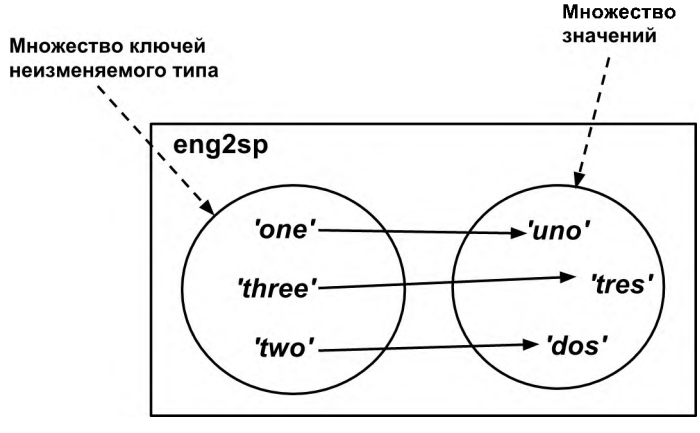


Рис. 23.1. **Схема соответствия множеству ключей множества значений словаря**

Создадим словарь-переводчик с английского языка на испанский язык:

```
>>> eng2sp = dict()      # создаем пустой словарь
>>> eng2sp
{}
>>> eng2sp['one']='uno' # добавляем 'uno' для элемента
                        # с индексом 'one'
>>> eng2sp
{'one': 'uno'}
>>> eng2sp['one']
'uno'
>>> eng2sp['two'] = 'dos'
>>> eng2sp['three'] = 'tres'
```

```
>>> eng2sp
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
```

В качестве индексов словаря в примере используются неизменяемые строки, но можно было воспользоваться кортежами, так как они тоже неизменяемые:

```
>>> e = {}
>>> e
{}
>>> e[(4,'6')] = '1'
>>> e
{(4, '6'): '1'}
```

Результирующий словарь **eng2sp** отобразился в «перемешанном» виде, так как, по аналогии с множествами, словари являются *неупорядоченной коллекцией*. Фактически, словарь — это отображение двух множеств: множества ключей и множества значений.

К словарям применим оператор **in**:

```
>>> eng2sp
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
>>> 'one' ineng2sp    # поиск по множеству КЛЮЧЕЙ
True
```

На практике часто словари используются, если требуется найти частоту встречаемости элементов последовательности (списке, строке, кортеже[1]).

Напишем функцию, которая возвращает словарь, содержащий частоту встречаемости элементов переданной *последовательности*:

```
def histogram(s):
        d = dict()
        for c in s:
            if c not in d:
                d[c] = 1
            else:
                d[c] = d[c] + 1 # или d[c] += 1
        return d
```

Результат вызова функции **histogram** для списка, строки, кортежа соответственно:

```
>>> histogram([2, 5, 6, 5, 4, 4, 4, 4, 3, 2, 2, 2, 2])
{2: 5, 3: 1, 4: 4, 5: 2, 6: 1}
>>> histogram("ywte3475eryt3478e477477474")
{'4': 6, '8': 1, 'e': 3, '3': 2, '7': 7, '5': 1, 'r': 1,
'y': 2, 'w': 1, 't': 2}
>>> histogram((5, 5, 5, 6, 5, 'r', 5))
{5: 5, 6: 1, 'r': 1}
```

---

[1] Все эти типы данных имеют общие свойства, так как относятся к последовательностям. Подробнее см.: https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range.

Глава 24
ОБРАБОТКА ИСКЛЮЧЕНИЙ В PYTHON

Рассмотрим пример программы, приводящей к ошибке:

```
# zero_error.py
x = int(input())
print(5/x)
```

Выполним программу и убедимся, что перевод буквы в число и деление на нуль приводят к ошибкам:

```
r
Traceback (most recent call last):
  File "C:\Python36-32\zero_error.py", line 1, in <module>
    x = int(input())
ValueError: invalid literal for int() with base 10: 'r'
>>>
0
Traceback (most recent call last):
  File "C:\Python36-32\zero_error.py", line 2, in <module>
    print(5/x)
ZeroDivisionError: division by zero
```

Каким образом произвести проверку, чтобы избежать аварийного завершения программы? Можно, например, путем проверок контролировать ввод с клавиатуры, как это обычно делается в процедурных языках программирования:

```
# if_error.py
x = int(input())
if x == 0:
    print("Error!")
else:
    print(5/x)
```

Теперь программа при делении на нуль не производит аварийного завершения:

```
>>>
0
Error!
```

В Python реализован[1] перехват ошибок, основанный на обработке исключительных ситуаций. Рассмотрим измененную программу:

```
# try_error.py
try:
    x = int(input("Введите число: "))
    print(5/x)
except:
    print("Возникла ошибка деления на нуль")
```

Выполним программу и попытаемся разделить на нуль:

```
Введите число: 0
Возникла ошибка деления на нуль
```

В блок **try** помещается код, в котором может произойти ошибка. В случае возникновения ошибки (исключительной ситуации) управление передается в блок **except**. Повторно выполним программу и обнаружим, что при возникновении ошибки перевода буквы в число, снова попадаем в блок **except**:

```
Введите число: к
Возникла ошибка деления на нуль
```

Дело в том, что **except** без указания типа исключительной ситуации перехватывает все виды возникающих ошибок. Как нам отделить ошибку деления на нуль от ошибки преобразования типов? Перейдем в интерактивный режим и выполним несколько операций, приводящих к исключениям[2]:

```
>>> 4/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4/0
ZeroDivisionError: division by zero
>>> int("r")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int("r")
ValueError: invalid literal for int() with base 10: 'r'
```

При делении на нуль возникает ошибка **ZeroDivisionError**, при преобразовании типов — **ValueError**. Воспользуемся этим знанием и изменим нашу программу:

```
# try_except.py
try:
    x = int(input("Введите число: "))
    print(5/x)
```

---

[1] Другие объектно-ориентированные языки тоже поддерживают данный механизм.

[2] Подробнее см.: https://docs.python.org/3/library/exceptions.html#bltin-exceptions.

```
except ZeroDivisionError:  # указываем тип исключения
    print("Возникла ошибка деления на нуль")
except ValueError:
    print("Возникла ошибка преобразования типов")
```

Запустим программу и убедимся, что теперь срабатывают различные блоки **except** в зависимости от типа возникающих исключений:

```
Введите число: 0
Возникла ошибка деления на нуль
>>>
Введите число: y
Возникла ошибка преобразования типов
```

Рассмотрим следующий пример обработки исключений:

```
# try_finally.py
try:
    x = int(input("Введите число: "))
    print(5/x)
except ZeroDivisionError as z:
    print("Обрабатываем исключение - деление на нуль!")
    print(z) # выводим на экран информацию об ошибке
except ValueErrorasv:
    print("Обрабатываем исключение - преобразование"
        + " типов!")
    print(v)
else:
    print("Выполняется, если не произошло"
        + " исключительных ситуаций!")
finally:
    print("Выполняется всегда и в последнюю очередь!")
```

Выполним программу для различных входных значений:

```
Введите число: 0
Обрабатываем исключение - деление на нуль!
division by zero
Выполняется всегда и в последнюю очередь!
>>>
Введите число: r
Обрабатываем исключение - преобразование типов!
invalid literal for int() with base 10: 'r'
Выполняется всегда и в последнюю очередь!
```

В примере показано, что информацию об исключении можно помещать в переменную (с помощью инструкции **as**) и выводить на экран с помощью функции **print**.

Перехват исключений используется при написании функций. Рассмотрим пример обработки ошибки, когда искомый элемент не обнаружен в списке:

```
# list_find.py
def list_find(lst, target):
    try:
        index = lst.index(target)
    # метод index генерирует исключение ValueError:
    except ValueError:
    ## ValueError: value is not in list
        index = -1
    return index

print(list_find([3, 5, 6, 7], -6))
```

Результат выполнения программы:

```
-1
```

Глава 25
РАБОТА С ФАЙЛАМИ

На практике данные для обработки часто поступают из внешних источников — файлов. Существуют различные форматы файлов, наиболее простой и универсальный — текстовый. Он открывается в любом текстовом редакторе (например, стандартном Блокноте). Расширения у текстовых файлов: .txt, .html, .csv и т.д.

Помимо текстовых есть другие типы файлов (музыкальные, видео, .doc, .ppt и пр.), которые открываются в специальных программах (музыкальных или видеопроигрывателях, MS Word и т.п.).

В этой главе остановимся на текстовых файлах, хотя возможности Python ими не ограничиваются.

Выполните следующие шаги:

1) создайте каталог *file_examples*;

2) с помощью (например, Блокнота) создайте в каталоге *file_examples* текстовый файл *example_text.txt*, содержащий следующий текст:

```
First line of text
Second line of text
Third line of text
```

3) создайте в каталоге *file_examples* файл *file_reader.py*, содержащий исходный текст программы на языке Python:

```
# file_reader.py
file = open('example_text.txt', 'r')
contents = file.read()
print(contents)
file.close()
```

Выполним программу *file_reader.py*:

```
First line of text
Second line of text
Third line of text
```

Описание программы представлено на рис. 25.1.

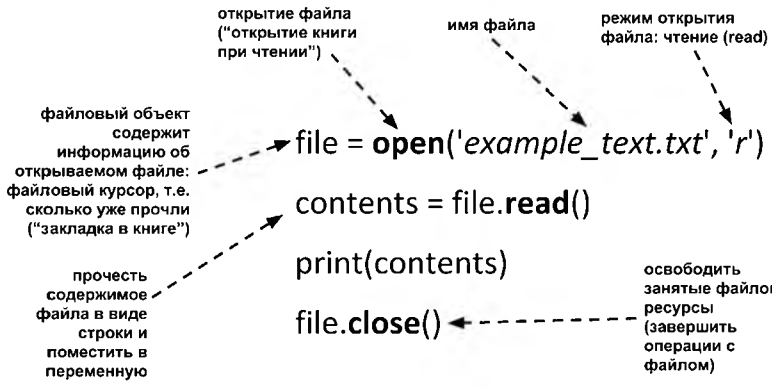


Рис. 25.1. **Описание программы для работы с файлами в Python**

Рассмотренный подход к работе с файлами[1] Python унаследовал от языка C. По умолчанию, если не указывать режим открытия, используется открытие на «чтение» (можно открыть на «запись» или «добавление»).

Файлы особенно подвержены ошибкам во время работы с ними: жесткий диск может заполниться, пользователь может удалить используемый файл во время записи, файл могут переместить и т.д. Эти и другие типы ошибок можно перехватить с помощью обработки исключений.

Далее показан пример обработки ошибки открытия несуществующего файла с именем **1example_text.txt**:

```
# file_reader2.py
# Ошибка при открытии файла
try:
    f = open('1example_text.txt') # открытие файла на чте-
ние
except:
    print("Ошибка открытия файла")
else:   # выполняется, если не произошло ошибки
    f.close()
    print('(Очистка: Закрытие файла)')
```

Выполним программу:

```
Ошибка открытия файла
```

В дальнейшем для работы с файлами будем использовать *менеджер контекста* (инструкцию **with**[2]), который не требует ручного освобождения ресурсов, т.е. вызова метода **close**. В следующем примере представлен исходный код программы с использованием менеджера контекста:

```
# file_reader3.py
```

---

[1] Подробнее см.: https://docs.python.org/3/library/os.html#os-file-dir.

[2] Менеджер контекста используется не только при работе с файлами.

```
try:
    # освобождение ресурсов происходит автоматически
    # внутри менеджера контекста:
    with open('1example_text.txt', 'r') as file:
        contents = file.read()
    print(contents)
except:
    print("Ошибка открытия файла")
```

Выполним программу:

```
Ошибка открытия файла
```

Разберемся, каким образом Python определяет, где искать файл для открытия. В момент вызова функции **open** Python ищет указанный файл в текущем *рабочем каталоге* (там, где расположена запускаемая программа). Определить текущий рабочий каталог можно следующим способом:

```
>>> import os
>>> os.getcwd()
'C:\\Python36-32\\file_examples'
```

Если открываемый файл находится в другом каталоге, то необходимо указать путь к нему:

1) абсолютный путь, т.е. начиная с корневого каталога:

```
'C:\\Users\\Dmitriy\\data1.txt'
```

2) относительный путь *'data\\data1.txt'*, т.е. путь относительно текущего рабочего каталога *'C:\\Users\\Dmitriy\\home'* (см. рис. 25.2).

Рис. 25.2. **Схема определения относительного пути**

Далее рассмотрим некоторые способы чтения содержимого файла.

Пример чтения содержимого всего файла, начиная с текущей *позиции курсора* (перемещает курсор в конец файла):

```
# file_reader4.py
with open('example_text.txt', 'r') as file:
    contents = file.read()
print(contents)
```

Результат выполнения программы:

```
First line of text
Second line of text
```

```
Third line of text
```

Следующий пример демонстрирует работу с *курсором* (аналог закладки в книге, с которой продолжается чтение):

```
# file_reader5.py
with open('example_text.txt', 'r') as file:
    contents = file.read(10)   # указываем кол-во
                               # символов для чтения
    # курсор перемещается на 11 символ
    rest = file.read()     # читаем с 11 символа
print("10:", contents)
print("остальное:", rest)
```

Результат работы программы:

```
10: First line
остальное: of text
Second line of text
Third line of text
```

Если необходимо получить список, состоящий из строк, то можно воспользоваться методом **readlines** так, как показано в примере:

```
# file_reader6.py
with open('example_text.txt', 'r') as file:
    lines = file.readlines()
print(lines)
```

Результат работы программы:

```
['First line of text\n', 'Second line of text\n', 'Third
line of text']
```

Для выполнения следующего примера создайте файл **plan.txt**, содержащий текст:

```
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

Напишем программу, обрабатывающую содержимое файла **plan.txt**:

```
# file_reader7.py
with open('plan.txt', 'r') as file:
    # Читаем содержимое файла в виде списка строк
    planets = file.readlines()
print(planets)
# Отображаем элементы списка в обратном порядке
for planet in reversed(planets):
```

```
    # Возвращаем копию строки, из которой удален символ /n
    print(planet.strip())
```

В результате выполнения программы:

```
['Mercury\n', 'Venus\n', 'Earth\n', 'Mars\n',
'Jupiter\n', 'Saturn\n', 'Uranus\n', 'Neptune']
Neptune
Uranus
Saturn
Jupiter
Mars
Earth
Venus
Mercury
```

Если необходимо выполнить некоторые операции с каждой из строк файла, начиная с текущей позиции файлового курсора до конца файла:

```
# file_reader8.py
with open('plan.txt', 'r') as file:
    for line in file:
        print(line)
        print(len(line.strip()))
```

Результат выполнения программы:

```
Mercury

7
Venus

5
Earth

5
Mars

4
Jupiter

7
Saturn

6
Uranus

6
Neptune
7
```

В следующем примере производится запись строки в файл. Если файла с указанным именем в рабочем каталоге нет, то он будет

создан, если файл с таким именем существует, то он будет перезаписан:

```
# file_write.py
with open("top.txt", 'w') as output_file:
    output_file.write("Hello!\n")
    # метод write возвращает число записанных символов
```

Для добавления строки в файл необходимо открыть файл в режиме '**a**' (сокр. от append):

```
# file_append.py
with open("top.txt", 'a') as output_file:
    output_file.write("Hello!\n")
```

Глава 26
РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Python поддерживает мощный язык регулярных выражений[1], т.е. шаблонов, по которым можно искать/заменять некоторый текст. Например, регулярное выражение **'[ea]'** означает любой символ из набора в скобках, т.е. регулярное выражение **'r[ea]d'** совпадает с **'red'** и **'radar'**, но не со словом **'read'**. Для работы с регулярными выражениями необходимо импортировать модуль **re**:

```
>>> import re
>>> re.search("r[ea]d", "rad") # указываем шаблон и текст
<_sre.SRE_Match object; span=(0, 3), match='rad'>
>>> re.search("r[ea]d", "read")
>>> re.search("[1-8]", "3")
<_sre.SRE_Match object; span=(0, 1), match='3'>
>>> re.search("[1-8]", "9")
>>>
```

В случае совпадения текста с шаблоном возвращается объект **match**[2], иначе возвращается **None**.

---

[1] Подробнее см.: https://docs.python.org/3/howto/regex.html.

[2] Подробнее см.: https://docs.python.org/3/library/re.html#match-objects.

Глава 27
ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА PYTHON

27.1. Основы объектно-ориентированного подхода

Предположим, что существует набор строковых переменных для описания адреса проживания некоторого человека:

```
addr_name = 'Иван Иванов'
addr_line1 = 'Московский пр. 122' # адрес прописки
addr_line2 = ''           # фактический адрес проживания
addr_city = 'Москва'
addr_state = 'Восточный'      # административный округ
addr_zip = '123678'           # индекс адреса прописки
```

Напишем функцию (**addr.py**), которая выводит на экран всю информацию о человеке:

```
def printAddress(name, line1, line2, city, state,
zip_code):
    print(name)
    if(len(line1) > 0):
        print(line1)
    if(len(line2) > 0):
        print(line2)
    print(city + ", " + state + " " + zip_code)
# Вызов функции, передача аргументов:
printAddress(addr_name, addr_line1, addr_line2,
   addr_city, addr_state, addr_zip)
```

Результат работы получившейся программы (**addr.py**):

```
Иван Иванов
Московский пр. 122
Москва, Восточный 123678
```

Предположим, изменились сведения о человеке и появился индекс адреса проживания. Создадим новую переменную:

```
# создадим переменную, содержащую индекс адреса проживания
addr_zip2 = "678900"
```

Функция **printAddress** с учетом новых сведений будет иметь следующий вид (**addr1.py**):

```
def printAddress(
    name, line1, line2, city, state, zip, zip2):
    # добавили параметр zip2
    print(name)
    if(len(line1) > 0):
        print(line1)
    if(len(line2) > 0):
        print(line2)
    # добавили вывод на экран переменной zip2
    print(city + ", " + state + " " + zip + " " + zip2)

# Добавили новый аргумент addr_zip2:
printAddress(addr_name, addr_line1, addr_line2,
    addr_city, addr_state, addr_zip, addr_zip2)
```

Пришлось в нескольких местах изменить исходный текст программы **(addr1.py)**, чтобы учесть новое условие. В чем состоит недостаток рассмотренного подхода? Огромное количество переменных! Чем больше сведений о человеке нужно обработать, тем больше переменных мы должны создать. Конечно, можно поместить все в список (элементами списка тогда будут строки), но в Python есть более универсальный подход для работы с наборами разнородных данных.

Объединим все сведения о человеке в единую структуру (*класс*) с именем **Address**:

```
class Address():     # имя класса выбирает программист
    name = ""        # строковое поле класса
    line1 = ""
    line2 = ""
    city = ""
    state = ""
    zip_code = ""
```

Класс в нашей программе задает шаблон для хранения места проживания человека. Превратить шаблон в конкретный адрес можно через создание *объекта* (*экземпляра*)[1] класса **Address**[2]:

```
homeAddress = Address()
```

Теперь заполним поля объекта конкретными значениями (через точку):

```
## заполняем поле name объекта homeAddress:
homeAddress.name = "Иван Иванов"
homeAddress.line1 = "Московский пр. 122"
homeAddress.line2 = "Тихая ул. 12"
homeAddress.city = "Москва"
homeAddress.state = "Восточный"
homeAddress.zip_code = "123678"
```

---

[1] В Python классы являются объектами, но для упрощения будем считать, что это только шаблон для создания объектов.

[2] Вспомните о создании объекта класса int: a = int ().

Создадим еще один объект класса **Address**, который содержит информацию о загородном доме того же человека:

```
# переменная содержит адрес объекта класса Address:
vacationHomeAddress = Address()
```

Зададим поля объекта, адрес которого находится в переменной **vacationHomeAddress**:

```
vacationHomeAddress.name = "Иван Иванов"
vacationHomeAddress.line1 = "пр.Мира 12"
vacationHomeAddress.line2 = ""
vacationHomeAddress.city = "Коломна"
vacationHomeAddress.state = "Центральный район"
vacationHomeAddress.zip_code = "12489"
```

Выведем на экран информацию о городе для основного и загородного адресов проживания (через указание имен объектов):

```
print("Основной адрес проживания " + homeAddress.city)
print("Адрес загородного дома "
    + vacationHomeAddress.city)
```

Изменим исходный код функции **printAddress** с учетом полученных знаний об объектах:

```
def printAddress(address): # передаем в функцию объект
    print(address.name)    # выводим на экран поле объекта
    if(len(address.line1) > 0):
        print(address.line1)
    if(len(address.line2) > 0):
        print(address.line2)
    print(address.city + ", " + address.state + "  "
    + address.zip_code)
```

Если объекты **homeAddress** и **vacationHomeAddress** ранее были созданы, то можем вывести информацию о них, передав в качестве аргумента функции **printAddress**:

```
printAddress(homeAddress)
printAddress(vacationHomeAddress)
```

В результате выполнения программы (**addr2.py**) получим:

```
Основной адрес проживания Москва
Адрес загородного дома Коломна
Иван Иванов
Московский пр. 122
Тихая ул. 12
Москва, Восточный 123678
Иван Иванов
пр.Мира 12
Коломна, Центральный район 12489
```

Возможности классов и объектов не ограничиваются лишь объединением переменных под одним именем, т.е. *хранением состоя-*

ния объекта. Классы также позволяют задавать функции внутри себя (*методы*) для работы с полями класса, т.е. влиять на *поведение объекта.* Для демонстрации создадим класс **Dog**:

```
# ndog.py
class Dog():
    age = 0       # возраст собаки
    name = ""     # имя собаки
    weight = 0    # вес собаки
    # первым аргументом любого метода всегда является
    # self, т.е. сам объект:
    def bark(self): # функция внутри класса называется
                    # методом
        # self.name - обращение к имени текущего
        # объекта-собаки
        print(self.name, " говорит гав")

# Создаем объект myDog класса Dog:
myDog = Dog()

# Присваиваем значения полям объекта myDog:
myDog.name = "Лайка" # имя собаки
myDog.weight = 20    # вес собаки
myDog.age = 1        # возраст собаки

# Вызываем метод bark объекта myDog,
# т.е. попросим собаку подать голос:
myDog.bark()
# Полная форма для вызова метода myDog.bark()
# будет: Dog.bark(myDog), где myDog - сам объект (self)
```

Результат работы программы:

```
Лайка говорит гав
```

Данный пример демонстрирует *объектно-ориентированный подход* в программировании, когда создаются объекты, приближенные к реальной жизни. Между объектами происходит взаимодействие по средствам вызова методов. Поля объекта (переменные) фиксируют его состояние, а вызов метода приводит к реакции объекта и (или) изменению его состояния (изменению переменных внутри объекта).

В предыдущем примере между созданием объекта **myDog** класса **Dog** и присвоением ему имени (**myDog.name = "Лайка"**) прошло некоторое время. Может произойти так, что программист забудет указать имя, и тогда собака останется безымянной — такого допустить нельзя. Избежать подобной ошибки позволяет специальный метод (*конструктор*), который вызывается автоматически в момент создания объекта заданного класса. Рассмотрим пример работы конструктора:

```
# dog1.py
class Dog():
    name = ""
    # Конструктор вызывается в момент создания объекта
    # этого класса;
    # специальный метод Python, поэтому два нижних
    # подчеркивания
    def __init__(self):
        print("Родилась новая собака!")

# Создаем собаку (объект myDog класса Dog)
myDog = Dog()
```

Запустим программу:

```
Родилась новая собака!
```

Следующий пример демонстрирует присвоение имени собаке через вызов конструктора класса:

```
# dog2.py
class Dog():
    name = ""
    # Конструктор
    # Вызывается на момент создания объекта этого класса
    def __init__(self, newName):
        self.name = newName

# Создаем собаку и устанавливаем ее имя:
myDog = Dog("Лайка")

# Выводим имя собаки:
print(myDog.name)

# Следующая команда выдаст ошибку, потому что
# конструктору не было передано имя:
# herDog = Dog()
```

Теперь имя собаки присваивается в момент ее создания (рождения). В конструкторе указали **self.name**, так как в момент вызова конструктора вместо **self** подставится конкретный объект, т.е. **myDog**.

Результат выполнения программы:

```
Лайка
```

В предыдущем примере для обращения к имени собаки мы выводили на экран поле **myDog.name**, т.е. залезали во внутренности объекта и доставали оттуда информацию о нем. Звучит жутковато, поэтому для более «гуманной» работы добавим в класс **Dog** два метода **setName** и **getName**:

```
# dog3.py
class Dog():
    name = ""
```

```
    # Конструктор вызывается в момент создания объекта
    # этого класса
    def __init__(self, newName):
        self.name = newName
    # Можем в любой момент вызвать метод и изменить
    # имя собаки
    def setName(self, newName):
        self.name = newName
    # Можем в любой момент вызвать метод и узнать имя
    # собаки
    def getName(self):
        return self.name
    # возвращаем текущее имя объекта

# Создаем собаку с начальным именем:
myDog = Dog("Лайка")

# Выводим имя собаки:
print(myDog.getName())

# Устанавливаем новое имя собаки:
myDog.setName("Шарик")

# Проверяем, что имя изменилось:
print(myDog.getName())
```

Выполним программу:

```
Лайка
Шарик
```

27.2. Наследование классов

Объектно-ориентированный подход (ООП) в программировании тесно связан с мышлением человека, с устройством его памяти. Чтобы лучше понять особенности ООП, рассмотрим модель хранения и извлечения информации из памяти человека (модель предложена Коллинзом и Квиллианом)[1]. В своем эксперименте ученые использовали семантическую сеть, в которой были представлены знания о канарейке (рис. 27.1).

Например, «канарейка — это желтая птица, которая умеет петь», «птицы имеют перья и крылья, умеют летать» и т.п. Знания в этой сети представлены на различных уровнях: на нижнем уровне располагаются более частные знания, а на верхних — более общие. При таком подходе для понимания высказывания «Канарейка может летать» необходимо воспроизвести информацию о том, что канарейка относится к множеству птиц, и у птиц есть общее свойст-

[1] *Гаврилова Т. А., Кудрявцев Д. В., Муромцев Д. И.* Инженерия знаний. Модели и методы : учебник. СПб. : Лань, 2016.

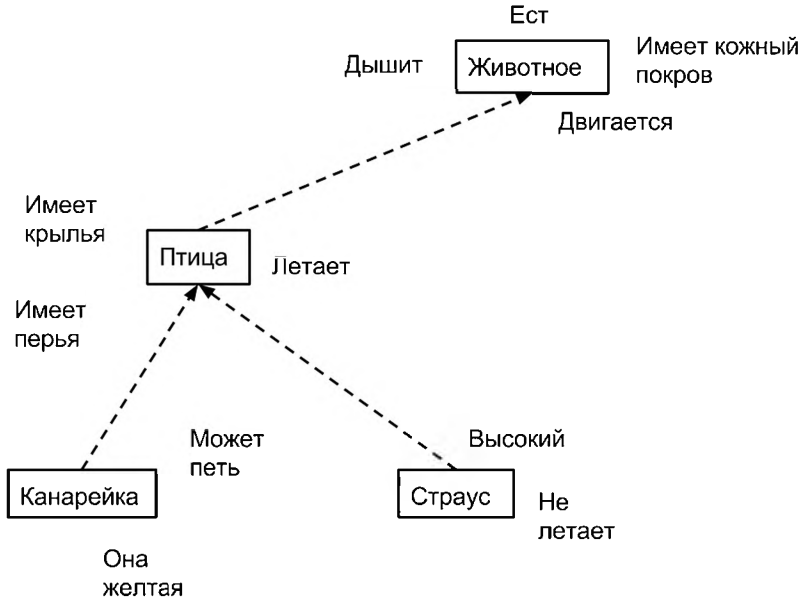


Рис. 27.1. **Модель хранения и извлечения информации из памяти человека (предложена Коллинзом и Квиллианом)**

во «летать», которое распространяется (*наследуется*) и на канареек. Лабораторные эксперименты показали, что реакции людей на простые вопросы типа «Канарейка — это птица?», «Канарейка может летать?» или «Канарейка может петь?» различаются по времени. Ответ на вопрос «Может ли канарейка летать?» требует большего времени, чем на вопрос «Может ли канарейка петь». По мнению Коллинза и Квиллиана, это связано с тем, что информация запоминается человеком на наиболее абстрактном уровне. Вместо того чтобы запоминать все свойства каждой птицы, люди запоминают только отличительные особенности, например, желтый цвет и умение петь у канареек, а все остальные свойства переносятся на более абстрактные уровни: канарейка как птица умеет летать и покрыта перьями; птицы, будучи животными, дышат и питаются и т.д. Действительно, ответ на вопрос «Может ли канарейка дышать?» требует большего времени, так как человеку необходимо проследовать по иерархии понятий в своей памяти. С другой стороны, конкретные свойства могут перекрывать более общие, что также требует меньшего времени на обработку информации. Например, вопрос «Может ли страус летать» требует меньшего времени для ответа, чем вопросы «Имеет ли страус крылья?» или «Может ли страус дышать?».

Упомянутое выше свойство наследования нашло свое отражение в объектно-ориентированном программировании.

К примеру, необходимо создать программу, содержащую описание классов Работник (**Employee**) и Клиент (**Customer**). Эти клас-

сы имеют общие свойства, присущие всем людям, поэтому создадим *базовый* класс Человек (**Person**) и наследуем от него *дочерние* классы **Employee** и **Customer** (рис. 27.2).

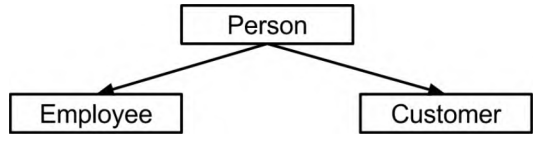


Рис. 27.2. **Иерархия классов**

Код, описывающий иерархию классов, представлен ниже (**person.py**):

```
class Person():
    name = ""       # имя человека
class Employee(Person): # указываем базовый класс Person
    job_title = ""    # наименование должности работника
class Customer(Person): # указываем базовый класс Person
    email = ""        # почта клиента
```

Создадим объекты на основе классов и заполним их поля:

```
person1 = Person()       # создаем объект класса Person
person1.name = "Иван Петров"    # заполняем поле объекта

personEmployee = Employee() # создаем объект класса
                            # Employee
# поле name унаследовано от класса Person:
personEmployee.name = "Петр Иванов"
personEmployee.job_title = "Программист"

personCustomer = Customer()
personCustomer.name = "Петр Петров"
personCustomer.email = "me@me.ru"
```

В объектах классов **Employee** и **Customer** появилось поле **name**, унаследованное от класса **Person**.

Результат работы программы **person.py**:

```
Иван Петров Петр Иванов Программист Петр Петров me@me.ru
```

Далее представлен пример наследования методов:

```
# person1.py
class Person():
    name = ""
    def __init__(self): # конструктор базового класса
        print("Создан человек")

class Employee(Person):
    job_title = ""
class Customer(Person):
    email = ""
```

```
person1 = Person()
personEmployee = Employee()
personCustomer = Customer()
```

Результат выполнения программы:

```
Создан человек
Создан человек
Создан человек
```

Таким образом, при создании объектов вызывается конструктор, унаследованный от базового класса. Если дочерние классы содержат собственные конструкторы, то выполняться будут они. Пример создания собственных конструкторов для дочерних классов:

```
# person2.py
class Person():
    name = ""
    def __init__(self):  # конструктор базового класса
        print("Создан человек")

class Employee(Person):
    job_title = ""
    def __init__(self): # конструктор дочернего класса
        print("Создан работник")

class Customer(Person):
    email = ""
    def __init__(self):  # конструктор дочернего класса
        print("Создан покупатель")

person1 = Person()
personEmployee = Employee()
personCustomer = Customer()
```

Результат работы программы:

```
Создан человек
Создан работник
Создан покупатель
```

Видим, что в момент создания объекта вызывается конструктор, содержащийся в дочернем классе, т.е. конструктор базового класса был *переопределен* в дочерних классах. Порой на практике требуется вызвать конструктор базового класса из конструктора дочернего класса:

```
# person3.py
class Person():
    name = ""
    def __init__(self):
        print("Создан человек")
class Employee(Person):
    job_title = ""
    def __init__(self):
```

```
        Person.__init__(self) # вызываем конструктор
                              # базового класса
        print("Создан работник")

class Customer(Person):
    email = ""
    def __init__(self):
         Person.__init__(self) # вызываем конструктор
                               # базового класса
         print("Создан покупатель")

person1 = Person()
personEmployee = Employee()
personCustomer = Customer()
```

Результат работы программы:

```
Создан человек
Создан человек
Создан работник
Создан человек
Создан покупатель
```

Глава 28
РАЗРАБОТКА ПРИЛОЖЕНИЙ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

28.1. Основы работы с модулем tkinter

Язык Python позволяет создавать приложения с графическим интерфейсом, для этого применяются различные графические библиотеки[1]. Остановимся на рассмотрении стандартной графической библиотеки **tkinter**[2] (входит в стандартную поставку Python, доступную с официального сайта **python.org**).

Первым делом при работе с **tkinter** необходимо создать главное (*корневое*) окно (рис. 28.1), в котором размещаются остальные графические элементы — *виджеты*. Существует большой набор виджетов[3] на все случаи жизни: для ввода текста, вывода текста, выпадающие меню и т.д. Среди виджетов есть кнопка, при нажатии на которую происходит заданное событие. Некоторые виджеты (*фреймы*) используются для группировки других виджетов внутри себя.

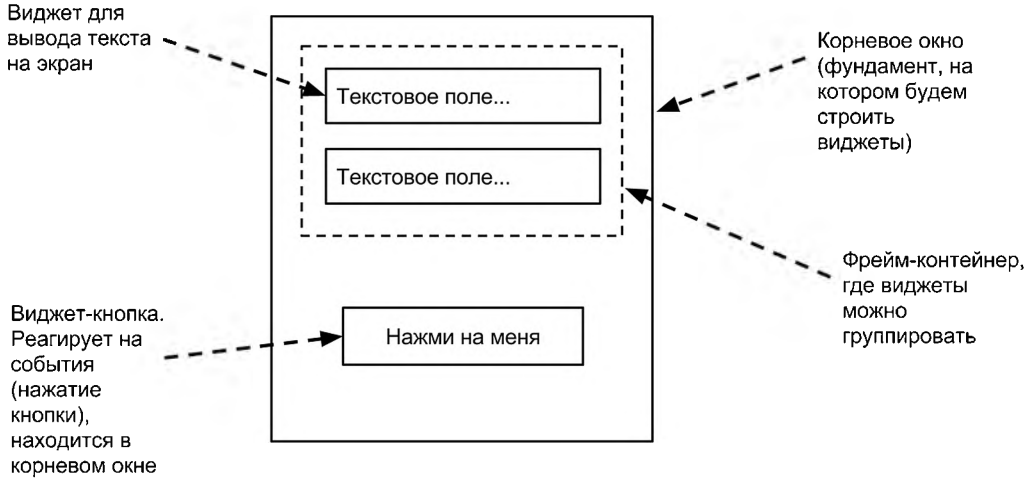


Рис. 28.1. **Схема главного окна в tkinter**

[1] Подробнее см.: https://wiki.python.org/moin/GuiProgramming.
[2] Подробнее см.: https://docs.python.org/3/library/tkinter.html.
[3] Подробнее см.: http://effbot.org/tkinterbook/tkinter-index.htm.

Приведем пример простейшей программы для отображения главного окна:

```
# mytk1.py
# Подключаем модуль, содержащий методы для работы
# с графикой
import tkinter
# Создаем главное (корневое) окно,
# в переменную window записываем ссылку на объект
# класса Tk
window = tkinter.Tk()
# Задаем обработчик событий для корневого окна
window.mainloop()
```

Результат выполнения программы показан на рис. 28.2.

Рис. 28.2. **Главное окно**

С помощью трех строчек кода на языке Python удалось вывести на экран полноценное окно, которое можно свернуть, растянуть или закрыть.

Графические (оконные) приложения отличаются от консольных (без оконных) наличием обработки событий. Для консольных (скорее, интерактивных) приложений, с которыми мы работали ранее, не требовалось определять, какую кнопку мыши и в какой момент времени нажал пользователь программы. В оконных приложениях важны все манипуляции с мышью и клавиатурой, так как от этого зависит, например, какой пункт меню выберет пользователь.

Слева на рис. 28.3 показан алгоритм работы консольной программы, справа — программы с графическим интерфейсом.

Следующий пример демонстрирует создание виджета **Label**:

```
# mytk2.py
import tkinter
window = tkinter.Tk()
# Создаем объект-виджет класса Label в корневом окне window
# text - параметр для задания отображаемого текста
label = tkinter.Label(window, text = "Это текст в окне!")
# Отображаем виджет с помощью менеджера pack
label.pack()
window.mainloop()
```

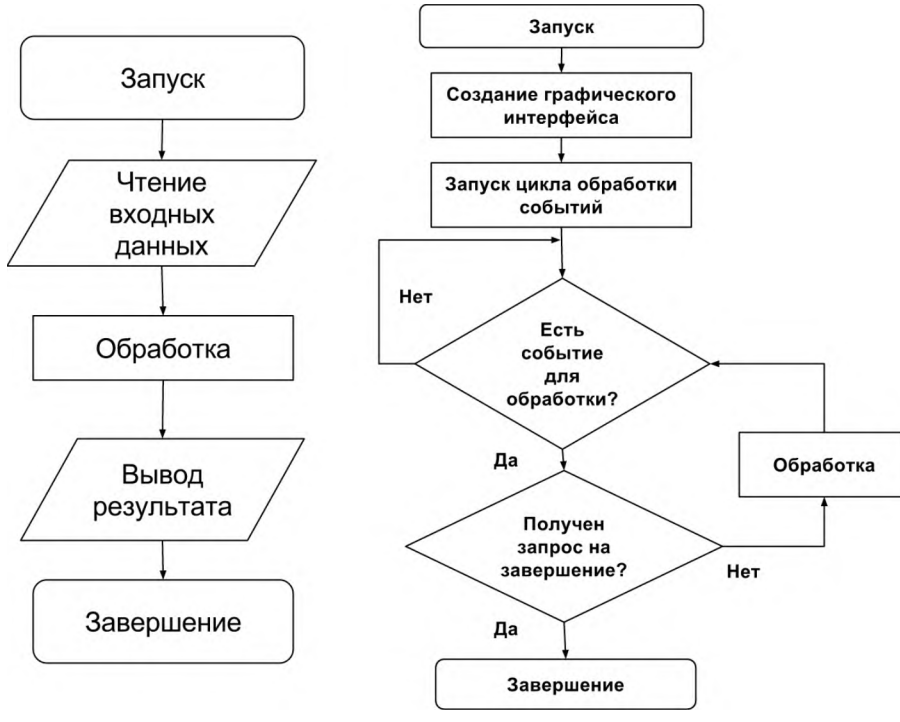


Рис. 28.3. **Схема работы консольной программы и программы с графическим интерфейсом**

В результате работы программы отображается графическое окно с текстом внутри (рис. 28.4).

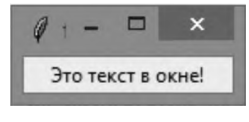


Рис. 28.4. **Текст в окне**

Далее представлен пример размещения виджетов во фрейме:

```
# mytk3.py
import tkinter
window = tkinter.Tk()
# Создаем фрейм в главном окне
frame = tkinter.Frame(window)
frame.pack()
# Создаем виджеты и помещаем их во фрейме frame
first = tkinter.Label(frame, text='First label')
# Отображаем виджет с помощью менеджера pack
first.pack()
second = tkinter.Label(frame, text='Second label')
second.pack()
third = tkinter.Label(frame, text='Third label')
third.pack()
window.mainloop()
```

Пример выполнения программы показан на рис. 28.5.

Рис. 28.5. **Размещение виджетов во фрейме**

Можно изменять параметры фрейма в момент создания объекта[1]:

```
# mytk4.py
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
# Можем изменять параметры фрейма:
frame2 = tkinter.Frame(window, borderwidth=4,
    relief=tkinter.GROOVE)
frame2.pack()
# Размещаем виджет в первом фрейме (frame)
first = tkinter.Label(frame, text='First label')
first.pack()
# Размещаем виджеты во втором фрейме (frame2)
second = tkinter.Label(frame2, text='Second label')
second.pack()
third = tkinter.Label(frame2, text='Third label')
third.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 28.6.

Рис. 28.6. **Изменение параметров фрейма при создании объекта**

В следующем примере для отображения в виджете **Label** содержимого переменной используется переменная **data** класса **StringVar** (из модуля **tkinter**). В дальнейшем из примеров станет понятнее, почему в **tkinter** используются переменные собственного класса[2].

```
# mytk5.py
import tkinter
window = tkinter.Tk()
# Создаем объект класса StringVar
```

[1] Подробнее см.: http://effbot.org/tkinterbook/frame.htm.

[2] Tkinter поддерживает работу с переменными классов: BooleanVar, DoubleVar, IntVar, StringVar.

```
# (создаем строковую переменную, с которой умеет
# работать tkinter):
data = tkinter.StringVar()
# Метод set позволяет изменить содержимое переменной:
data.set('Данные в окне')
# связываем текст для виджета Label с переменной data
label = tkinter.Label(window, textvariable = data)
label.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 28.7.

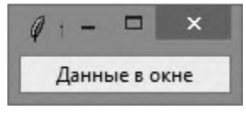


Рис. 28.7. **Отображение переменной в окне**

28.2. Шаблон «Модель — Вид — Контроллер» на примере модуля tkinter

Следующий пример показывает, каким образом использовать виджет **(Entry)** для ввода данных:

```
# mytk6.py
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
var = tkinter.StringVar()
# Обновление содержимого переменной в момент ввода текста
label = tkinter.Label(frame, textvariable=var)
label.pack()
# Пробуем набрать текст в появившемся поле для ввода
entry = tkinter.Entry(frame, textvariable=var)
entry.pack()
window.mainloop()
```

Запустим программу и внутри текстового окна попробуем набрать произвольный текст (рис. 28.8).

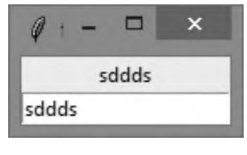


Рис. 28.8. **Виджет для ввода данных**

Видим, что текст, который мы набираем, сразу отображается в окне. Дело в том, что виджеты **Label** и **Entry** используют для вывода и ввода текста одну и ту же переменную **data** класса **StringVar**. Подобная схема работы оконного приложения укладывается в универсальный шаблон (паттерн) MVC. В общем виде под *моделью*

(Model) понимают способ хранения данных (например, в переменной какого класса). *Вид* (View) служит для отображения данных. *Контроллер* (Controller) отвечает за обработку данных (рис. 28.9).

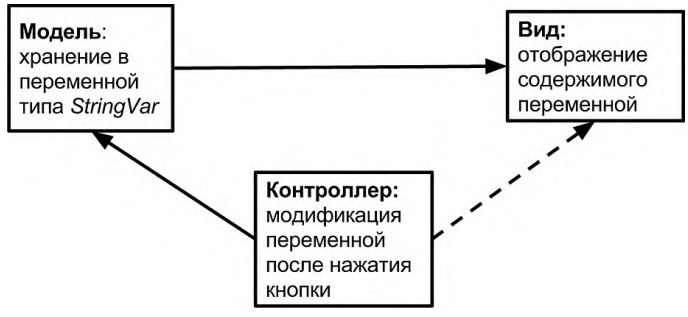


Рис. 28.9. **Схема шаблона MVC**

Особенностью шаблона MVC является то, что в случае изменения контроллером данных (как это было в предыдущем примере с изменением переменной **var**) «посылается сигнал» *виду* с просьбой обновить отображаемое содержимое (перерисовать окно), отсюда возникает обновление текста в режиме реального времени. Следующий пример демонстрирует возможности обработки событий при нажатии на кнопку (виджет **Button**):

```
# mytk7.py
import tkinter

# Контроллер: функция вызывается в момент нажатия
# на кнопку
def click():
    # метод get возвращает текущее значение counter
    # метод set устанавливает новое значение counter
    counter.set(counter.get() + 1)

window = tkinter.Tk()
# Модель: создаем объект класса IntVar
counter = tkinter.IntVar()
# Обнуляем созданный объект с помощью метода set
counter.set(0)
frame = tkinter.Frame(window)
frame.pack()
# Создаем кнопку и указываем обработчик (функция click)
# при нажатии на нее
button = tkinter.Button(frame, text='Click',
    command=click)
button.pack()
# Вид: в реальном времени обновляется содержимое
# виджета Label
label = tkinter.Label(frame, textvariable=counter)
label.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 28.10.

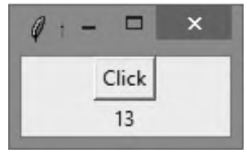


Рис. 28.10. **Обработка события при нажатии на кнопку**

Далее представлен более сложный пример с двумя кнопками и двумя обработчиками событий (**click_up**, **click_down**):

```
# mytk8.py
import tkinter
window = tkinter.Tk()
# Модель:
counter = tkinter.IntVar()
counter.set(0)

# Два контроллера:
def click_up():
    counter.set(counter.get() + 1)
def click_down():
    counter.set(counter.get() - 1)

# Вид:
frame = tkinter.Frame(window)
frame.pack()
button = tkinter.Button(frame, text='Up',
    command=click_up)
button.pack()
button = tkinter.Button(frame, text='Down',
    command=click_down)
button.pack()
label = tkinter.Label(frame, textvariable=counter)
label.pack()
window.mainloop()
```

Результат работы программы показан на рис. 28.11.

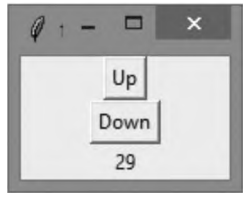


Рис. 28.11. **Пример с двумя кнопками и двумя обработчиками событий**

28.3. Изменение параметров по умолчанию при работе с tkinter

Tkinter позволяет изменять параметры виджетов в момент их создания:

```
# mytk9.py
import tkinter
window = tkinter.Tk()
# Создаем кнопку, изменяем шрифт с помощью кортежа
button = tkinter.Button(window, text='Hello',
                        font=('Courier', 14, 'bold italic'))
button.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 28.12.

Рис. 28.12. **Изменение шрифта при создании виджета**

В следующем примере изменяются параметры виджета **Label**:

```
# mytk10.py
import tkinter
window = tkinter.Tk()
# Изменяем фон, цвет текста:
button = tkinter.Label(window, text='Hello', bg='green',
    fg='white')
button.pack()
window.mainloop()
```

Результат выполнения программы показан на рис. 28.13.

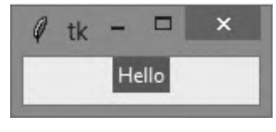


Рис. 28.13. **Изменение параметров виджета Label**

Менеджер расположения (*геометрии*) **pack** тоже имеет свои параметры. Далее представлен исходный код, демонстрирующий изменение параметров менеджера геометрии (рис. 28.14):

```
# mytk11.py
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
label = tkinter.Label(frame, text='Name')
# Выравнивание по левому краю
label.pack(side = 'left')
entry = tkinter.Entry(frame)
entry.pack(side='left')
window.mainloop()
```

Особенность следующего примера в том, что введенный текст (через виджет **Entry**) отображается на экране (через виджет **Label**)

Рис. 28.14. **Изменение параметров менеджера геометрии**

только в момент нажатия кнопки (виджет **Button**), а не в реальном времени, как это было ранее (рис. 28.15).

```
# mytk12.py
import tkinter
# Вызывается в момент нажатия на кнопку:
def click():
    # Получаем строковое содержимое поля ввода и
    # с помощью config изменяем отображаемый текст
    label.config(text=entry.get())

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
entry = tkinter.Entry(frame)
entry.pack()
label = tkinter.Label(frame)
label.pack()
# Привязываем обработчик нажатия на кнопку
# к функции click
button = tkinter.Button(frame, text='Печать!',
command=click)
button.pack()
window.mainloop()
```

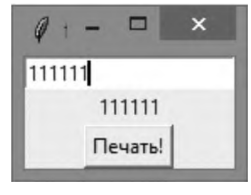


Рис. 28.15. **Вывод текста при нажатии на кнопку**

Глава 29
РЕАЛИЗАЦИЯ АЛГОРИТМОВ

Предположим, необходимо найти позицию наименьшего элемента в следующем наборе данных: 809, 834, 477, 478, 307, 122, 96, 102, 324, 476.

Первым делом выбираем подходящий для хранения тип данных. Очевидно, что это будет список:

```
>>> counts = [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
>>> counts.index(min(counts))
# решение задачи в одну строку!
6
>>>
```

Усложним задачу и попытаемся найти позицию двух наименьших элементов в неотсортированном списке. Возможные алгоритмы решения:

1) *поиск, удаление, поиск*. Поиск индекса минимального элемента в списке, удаление его, снова поиск минимального, возвращаем удаленный элемент в список;

2) *сортировка, поиск минимальных, определение индексов*;

3) *перебор всего списка*. Сравниваем каждый элемент по порядку, получаем два наименьших значения, обновляем значения, если найдены наименьшие.

Рассмотрим каждый из перечисленных алгоритмов.

1. **Поиск, удаление, поиск**: поиск индекса минимального элемента в списке, удаление его, снова поиск минимального, возвращение удаленного элемента обратно в список.

Начнем:

```
        [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
индекс:  0    1    2    3    4    5   6   7    8    9
```

Первый минимальный: 96.

Индекс элемента 96: 6.

Удаляем из списка найденный минимальный элемент **(96)**, при этом индексы в обновленном списке смещаются:

```
        [809, 834, 477, 478, 307, 122,     102, 324, 476]
индекс:  0    1    2    3    4    5       6    7    8
```

Второй минимальный: 102.

Индекс элемента 102: 6.

Возвращаем удаленный (первый минимальный) элемент обратно в список:

```
          [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
индекс:    0    1    2    3    4    5    6    7    8    9
```

Не забываем о смещении индексов после удаления первого минимального элемента: индекс второго минимального элемента равен индексу первого минимального элемента, поэтому увеличиваем индекс второго минимального на **1**. Функция, реализующая данный алгоритм:

```
def find_two_smallest(L):
  smallest = min(L)
  min1 = L.index(smallest)
  L.remove(smallest)
  # удаляем первый минимальный элемент
  next_smallest = min(L)
  min2 = L.index(next_smallest)
  L.insert(min1, smallest)
  # возвращаем первый минимальный
  # проверяем индекс второго минимального из-за смещения:
  if min1 <= min2:
    min2 += 1     # min2 = min2 + 1
  return (min1, min2)  # возвращаем кортеж
```

2. **Сортировка, поиск минимальных, определение индексов:** реализация второго алгоритма интуитивно понятна, поэтому приведем только исходный текст функции:

```
def find_two_smallest(L):
    temp_list = sorted(L)
    smallest = temp_list[0]
    next_smallest = temp_list[1]
    min1 = L.index(smallest)
    min2 = L.index(next_smallest)
    return (min1, min2)
```

3. **Перебор всего списка:** сравниваем каждый элемент по порядку, получаем два наименьших значения, обновляем значения, если найдены наименьшие.

Третий алгоритм наиболее сложный из перечисленных выше, поэтому остановимся на нем подробнее.

В отличие от человека, который может охватить взглядом сразу весь список и интуитивно сказать, какой из элементов является минимальным, компьютер не обладает подобным интеллектом. Он просматривает элементы по одному, последовательно перебирая их и сравнивая.

На первом шаге рассматриваются первые два элемента списка.

```
          [809, 834, ...]
индекс:     0    1
```

Сравниваем **809** и **834** и определяем наименьший из них, чтобы задать начальные значения **min1** и **min2**, где будут храниться *индексы* первого минимального и второго минимального элементов соответственно. Затем перебираем элементы, начиная с индекса **2** до окончания списка.

Определили, что **809** — первый минимальный, а **834** — второй минимальный элемент из двух первых встретившихся значений списка. Рассматриваем следующий элемент списка (**477**).

Элемент **477** оказался наименьшим (назовем это «первым случаем»). Поэтому обновляем содержимое переменных **min1** и **min2**, так как нашли новый наименьший элемент.

Рассматриваем следующий элемент списка (**478**). Он оказался между двумя минимальными элементами (назовем это «вторым случаем»).

Снова обновляем минимальные элементы (теперь обновился только **min2**), и т.д. пока не дойдем до конца списка.

Далее представлен исходный текст функции, реализующий предложенный алгоритм:

```
def find_two_smallest(L):
    if L[0] < L[1]:
        min1, min2 = 0, 1
        # устанавливаем начальные значения
    else:
        min1, min2 = 1, 0

    for i in range(2, len (L)):
        if L[i] < L[min1]:  # «первый случай»
            min2 = min1
            min1 = i
        elif L[i] < L[min2]:  # «второй случай»
            min2 = i
    return (min1, min2)
```

Контрольные вопросы и задания

1. Каковы сильные и слабые стороны языка программирования Python?

2. Какие правила наименования переменных в Python существуют? Опишите модель памяти Python при работе с переменными.

3. Опишите процесс создания функций в Python.

4. Какие отличия между выполнением команд в файле от выполнения в интерактивном режиме?

5. Какие существуют операции над строками в языке Python?

6. Какие существуют операторы отношений в Python? Перечислите правила логических операций над объектами.

7. В каких случаях применяется условная инструкция if?

8. Что такое модуль в Python?

9. Опишите процесс создания собственных модулей в Python.

10. Какие существуют строковые методы в Python? В чем отличие функций от методов?

11. Что такое список в Python? Опишите процесс создания списка.

12. Перечислите основные операции над списками в Python.

13. Что такое пседонимы? В чем заключается клонирование списков в Python?

14. Перечислите основные методы списка в Python.

15. Приведите примеры преобразования типов в Python (списки, строки).

16. Опишите возможности применения вложенных списков в Python.

17. Какие циклы существуют в Python?

18. В каких случаях применяется цикл for (на примере списков и строк)?

19. В каких случаях используется функция range в Python?

20. Перечислите способы генерации списка в Python.

21. В каких случаях применяется цикл while в Python?

22. Опишите область применения вложенных циклов в Python (на примере вложенных списков).

23. Что такое множество? Какие операции существуют над множествами в Python?

24. Что такое кортеж? Какие операции над кортежами существуют в Python?

25. Что такое словарь? Какие операции над словарями существуют в Python?

26. Как происходит обработка исключений в Python?

27. Какие особенности объектно-ориентированного программирования существуют в Python? Что такое классы, объекты?

28. Опишите структуру оконного приложения на примере модуля tkinter.

29. Что такое шаблон «Модель-вид-контроллер» (на примере модуля tkinter)?

Задания для самостоятельного выполнения

1. Найдите значение выражения $2 + 56 \cdot 5.0 - 45.5 + 5^5$.

2. Найдите значения для следующих выражений:

```
pow(abs(-5) + abs(-3), round(5.8))
int(round(pow(round(5.777, 2), abs(-2)), 1))
```

3. Создайте собственные функции для вычисления следующих выражений:

а) $x^4 + 4^x$;

б) $y^4 + 4^x$.

4. Создайте в отдельном файле функцию, переводящую градусы по шкале Цельсия T_C в градусы по шкале Фаренгейта T_F по формуле:

$$T_F = 9/5 \cdot T_C + 32.$$

5. Напишите в отдельном файле функцию, вычисляющую среднее арифметическое трех чисел. Задайте значения по умолчанию, в момент вызова используйте ключевые аргументы.

6. Попросите пользователя ввести свое имя и после этого отобразите на экране строку вида: «Привет, <имя>!» Вместо <имя> должно указываться то, что пользователь ввел с клавиатуры.

Пример работы программы:

```
Как тебя зовут?
Вася
Привет, Вася!
```

7. Напишите программу, определяющую сумму и произведение трех чисел (типа int, float), введенных с клавиатуры.

Пример работы программы:

```
Введите первое число: 1
Введите второе число: 4
Введите третье число: 7
Сумма введенных чисел: 12
Произведение введенных чисел: 28
```

8. Напишите собственную программу, определяющую максимальное из двух введенных чисел. Реализовать в виде вызова собственной функции, возвращающей большее из двух переданных ей чисел.

9. Напишите программу, проверяющую целое число на четность. Реализовать в виде вызова собственной функции.

10. Напишите программу, которая по коду города и длительности переговоров вычисляет их стоимость и выводит результат на экран: Екатеринбург — код 343, 15 руб/мин; Омск — код 381, 18 руб/мин; Воронеж — код 473, 13 руб/мин; Ярославль — код 485, 11 руб/мин.

11. Найдите площадь треугольника с помощью формулы Герона. Стороны задаются с клавиатуры. Реализовать вычисление площади в виде функции, на вход которой подаются три числа, на выходе возвращается площадь. Функция находится в отдельном модуле, где происходит разделение между запуском и импортированием.

12. Напишите программу-игру в виде отдельного модуля. Компьютер загадывает случайное число, пользователь пытается его угадать. Программа запрашивает число ОДИН раз. Если число угадано, то выводим на экран «Победа», иначе — «Повторите еще раз»! Для написания программы понадобится функция **randint()** из модуля **random**[1].

13. Напишите функцию, вычисляющую значение: $x^4 + 4^x$. Автоматизируйте процесс тестирования функции с помощью модуля **doctest**.

---

[1] https://docs.python.org/3/library/random.html.

14. Задана строка s = "У лукоморья 123 дуб зеленый 456":

1) определить, встречается ли в строке буква 'я'. Вывести на экран ее позицию (индекс) в строке;

2) определить, сколько раз в строке встречается буква 'у';

3) определить, состоит ли строка только из букв, ЕСЛИ нет, ТО вывести строку в верхнем регистре;

4) определить длину строки. ЕСЛИ длина строки превышает четыре символа, ТО вывести строку в нижнем регистре;

5) заменить в строке первый символ на 'О'. Результат вывести на экран.

15. Написать в отдельном модуле функцию, которая на вход принимает два аргумента: строку (s) и целочисленное значение (n). ЕСЛИ длина строки s превышает n символов, ТО функция возвращает строку s в верхнем регистре, ИНАЧЕ возвращается исходная строка s.

16. Дан список L = [3, 6, 7, 4, –5, 4, 3, –1]:

1) определите сумму элементов списка L. ЕСЛИ сумма превышает значение 2, ТО вывести на экран число элементов списка;

2) определите разность между минимальным и максимальным элементами списка. ЕСЛИ абсолютное значение разности больше 10, ТО вывести на экран отсортированный по возрастанию список, ИНАЧЕ вывести на экран фразу 'Разность меньше 10'.

17. Дан список L = [3, 'hello', 7, 4, 'привет', 4, 3, –1]. Определите наличие строки 'привет' в списке. ЕСЛИ такая строка в списке присутствует, ТО вывести ее на экран, повторив 10 раз.

18. Дан список L = [3, 'hello', 7, 4, 'привет', 4, 3, –1]. Определите наличие строки 'привет' в списке. ЕСЛИ такая строка в списке присутствует, ТО удалить ее из списка, ИНАЧЕ добавить строку в список. Подсчитать, сколько раз в списке встречается число 4, ЕСЛИ больше одного раза, ТО очистить список.

19. Напишите программу, которая запрашивает у пользователя две строки и формирует из этих строк список. Если строки состоят только из чисел, то программа добавляет в середину списка сумму введенных чисел, иначе добавляется строка, образованная из слияния двух введенных ранее строк. Итоговая строка выводится на экран.

20. Задан список слов. Необходимо выбрать из него случайное слово. Из выбранного случайного слова случайно выбрать букву и попросить пользователя ее угадать. Пример работы программы:

- задан список слов: ['самовар', 'весна', 'лето'];
- выбираем случайное слово: 'весна';
- выбираем случайную букву: 'с';
- выводим на экран: ве?на.

Пользователь пытается угадать букву.

Подсказка: используйте метод *choice()* модуля *random*.

21. Найдите все значения функции $y(x) = x^2 + 3$ на интервале от 10 до 30 с шагом 2.

22. Дан список L = [–8, 8, 6.0, 5, 'строка', –3.1]. Определить сумму чисел, входящих в список L.

Подсказка: для определения типа объекта можно воспользоваться сравнением вида: type(–8) == int.

23. Напишите программу-игру. Компьютер загадывает случайное число, пользователь пытается его угадать. Пользователь вводит число до тех пор, пока не угадает или не введет слово 'Выход'. Компьютер сравнивает число с введенным и сообщает пользователю, больше оно или меньше загаданного.

24. Дано число, введенное с клавиатуры. Определите сумму квадратов нечетных цифр в числе.

25. Найдите сумму чисел, вводимых с клавиатуры. Количество вводимых чисел заранее неизвестно. Окончание ввода, например, слово 'Стоп'.

26. Дан произвольный текст. Найдите номер первого самого длинного слова в нем.

27. Дан произвольный текст. Напечатайте все имеющиеся в нем цифры, определите их количество, сумму и найдите максимальное.

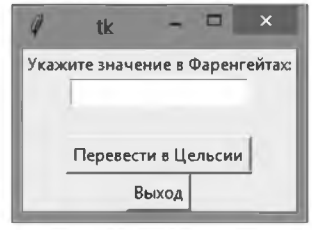
28. Напишите функцию, которая возвращает разность между наибольшим и наименьшим значениями из списка целых случайных чисел.

29. Напишите программу, проверяющую четность числа, вводимого с клавиатуры. Выполните обработку возможных исключений.

30. Создайте класс **StringVar** для работы со строковым типом данных, содержащий методы **set()** и **get()**. Метод **set()** служит для изменения содержимого строки класса **StringVar**, **get()** — для получения содержимого строки класса **StringVar**. Создайте объект типа **StringVar** и протестируйте его методы.

31. Создайте класс точки **Point**, позволяющий работать с координатами (x, y). Добавьте необходимые методы класса.

32. Напишите оконное приложение, позволяющее переводить градусы по шкале Фаренгейта в градусы по шкале Цельсия:



Литература

1. *Столяров, А. В.* Программирование: введение в профессию. Т. 1: Азы программирования / А. В. Столяров. — М. : МАКС Пресс, 2016.

2. *Столяров, А. В.* Программирование: введение в профессию. Т. 2: Низкоуровневое программирование / А. В. Столяров. — М. : МАКС Пресс, 2016.

3. *Керниган, Б. В.* Язык программирования Си : пер. с англ. / Б. В. Керниган, Д. М. Ритчи — 3-е изд. — СПб. : Невский Диалект, 2001.

4. *Подбельский, В. В.* Программирование на языке Си / В. В. Подбельский — М. : Финансы и статистика, 2004.

5. *Березин, Б. И.* Начальный курс С и С++ : учеб. пособие / Б. И. Березин, С. Б. Березин. — М. : ДИАЛОГ-МИФИ, 2004.

6. *Культин, Н. Б.* С/С++ в задачах и примерах : сб. задач / Н. Б. Культин. — СПб. : БХВ-Петербург, 2004.

7. *Лучано, Р.* Python. К вершинам мастерства / Р. Лучано. — М. : ДМК Пресс, 2016.

8. *Лутц, М.* Изучаем Python : пер. с англ. / М. Лутц. — 4-е изд.— СПб. : Символ-Плюс, 2011.

9. *Лутц, М.* Программирование на Python, Т. I : пер. с англ. / М. Лутц. — 4-е изд. — СПб. : Символ-Плюс, 2011.

10. *Лутц, М.* Программирование на Python, Т. II : пер. с англ. / М. Лутц. — 4-е изд. — СПб. : Символ-Плюс, 2011.

11. *Лаврищева, Е. М.* Технология программирования и программная инженерия : учебник для вузов / Е. М. Лаврищева. — М. : Издательство Юрайт, 2017.

12. *Лаврищева, Е. М.* Технология разработки и моделирования вариантов программных систем : монография для вузов / Е. М. Лаврищева. — М. : Издательство Юрайт, 2017.

Новые издания по дисциплине «Программирование» и смежным дисциплинам

1. *Бессмертный, И. А.* Системы искусственного интеллекта : учеб. пособие для академического бакалавриата / И. А. Бессмертный. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2016.

2. *Гниденко, И. Г.* Технологии и методы программирования : учеб. пособие для прикладного бакалавриата / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — М. : Издательство Юрайт, 2017.

3. *Демин, А. Ю.* Информатика. Лабораторный практикум : учеб. пособие для прикладного бакалавриата / А. Ю. Демин, В. А. Дорофеев. — М. : Издательство Юрайт, 2016.

4. *Дубров, Д. В.* Программирование: система построения проектов Cmake : учебник для магистратуры / Д. В. Дубров. — М. : Издательство Юрайт, 2016.

5. *Жмудь, В. А.* Моделирование замкнутых систем автоматического управления : учеб. пособие для академического бакалавриата / В. А. Жмудь. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2017.

6. *Зимин, В. П.* Информатика. Лабораторный практикум. В 2 ч. : учеб. пособие для вузов / В. П. Зимин. — М. : Издательство Юрайт, 2017.

7. *Зыков, С. В.* Программирование : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2016.

8. *Зыков, С. В.* Программирование. Объектно-ориентированный подход : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2017.

9. *Зыков, С. В.* Программирование. Функциональный подход : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2017.

10. *Казанский, А. А.* Объектно-ориентированный анализ и программирование на visual basic 2013 : учебник для прикладного бакалавриата / А. А. Казанский. — М. : Издательство Юрайт, 2017.

11. *Казанский, А. А.* Прикладное программирование на Excel 2013 : учеб. пособие для прикладного бакалавриата / А. А. Казанский. — М. : Издательство Юрайт, 2017.

12. *Казанский, А. А.* Программирование на Visual c# 2013 : учеб. пособие для прикладного бакалавриата / А. А. Казанский. — М. : Издательство Юрайт, 2017.

13. *Лебедев, В. М.* Программирование на VBA в MS Excel : учеб. пособие для академического бакалавриата / В. М. Лебедев. — М. : Издательство Юрайт, 2017.

14. *Малявко, А. А.* Формальные языки и компиляторы : учеб. пособие для вузов / А. А. Малявко. — М. : Издательство Юрайт, 2017.

15. *Мамонова, Т. Е.* Информационные технологии. Лабораторный практикум : учеб. пособие для прикладного бакалавриата / Т. Е. Мамонова. — М. : Издательство Юрайт, 2016.

16. *Маркин, А. В.* Программирование на SQL 6. В 2 ч. : учебник и практикум для бакалавриата и магистратуры / А. В. Маркин. — М. : Издательство Юрайт, 2017.

17. Методы оптимизации: теория и алгоритмы : учеб. пособие для академического бакалавриата / А. А. Черняк, Ж. А. Черняк, Ю. М. Метельский, С. А. Богданович. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2017.

18. *Тузовский, А. Ф.* Объектно-ориентированное программирование : учеб. пособие для прикладного бакалавриата / А. Ф. Тузовский. — М. : Издательство Юрайт, 2017.