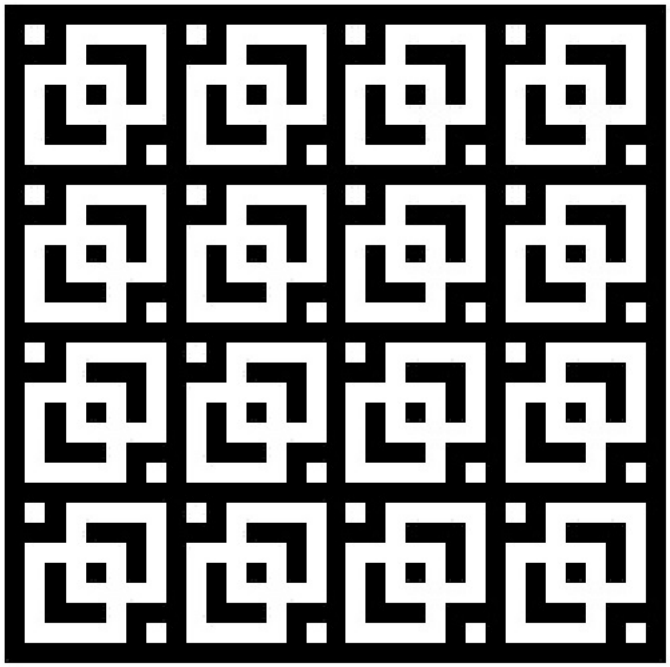
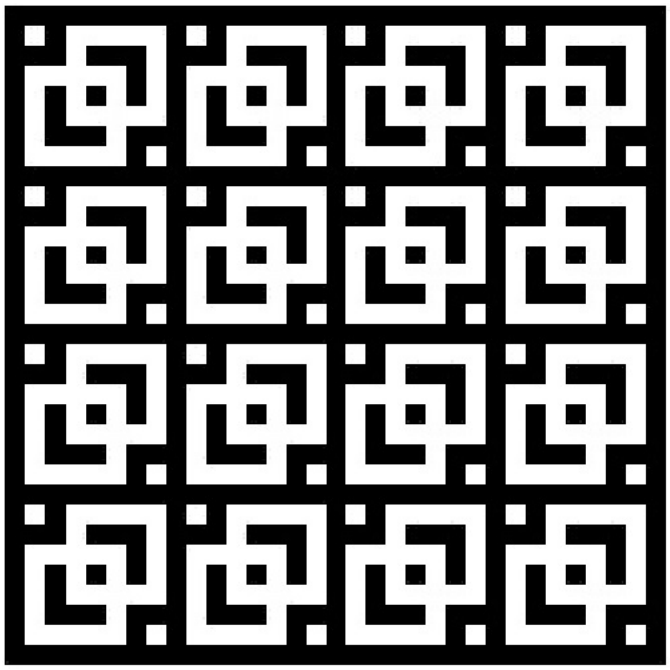
A Practical Introduction to Python Programming



author Brian Heinold

Department of Mathematics and Computer Science
Mount St.Mary's University

Практическое Введение в Программирование на Python



автор Brian Heinold
перевод Игорь Антоненко

Санкт-Петербург
2023 г.

Оглавление

Вступление

Моей целью здесь является что-то, что есть частично учебник, а частично справочник.Мне нравиться как учебники помогают быстро ввести Вас в курс дела,но они часто могут быть немного многословны и неупорядочены. Справочники содержат много хорошей информации, но они также кратки, и зачастую не дают Вам чувство что есть важно.Моя цель здесь, создать что-то типа учебника,но ещё полезного как справочник.Я обобщаю информацию в таблицах и даю много коротких примеров программ.Мне также нравиться перейти прямо к объектам, по ходу рассматривая его,чем сначала давать справочный материал.

Эта книга начиналась, как примерно 30 страниц заметок, для студентов моего начального курса программирования в Mount St. Mary's University. Большинство из этих студентов не имело первоначального опыта программирования и это определило мой подход.Я исключил множество технических деталей и иногда я упрощаю вещи. Некоторые из этих деталей пополняются дальше в книге, хотя другие детали никогда. Но эта книга не создана чтобы охватить все.Я рекомендую читать другие книги и документацию Python для заполнения пробелов.

Стиль программирования в этой книге направлен к видам программируемых объектов,которые я люблю делать—коротких программ,часто математической природы, маленьких утилит, которые делают мою жизнь легче, и маленьких компьютерных игр.В действительности, вещи, которые я рассматриваю в книге есть вещи, которые я нашел самыми полезными или интересными в моей практике программирования,и эта книга частично служит для документирования этих вещей для меня самого.Эта книга не создавалась для полной подготовке к карьере разработчика програмного обеспечения.Заинтересованные читатели должны продвигаться от этой книги к книге,которая имеет больше знаний о информатике и разработке и организации больших программ.

В плане построения курса на основе этой книги или самостоятельного изучения,основой является большая часть Главы I. Первые четыре части критически важны.Глава 5 полезна, но не все в ней имеет важное значение.Глава 6 (строки) должна быть проработана прежде главы 7(списки).Глава 8 содержит некоторые более расширенные темы списков.Многое из неё может быть пропущено,хотя она вся интересная и полезная.В особенности,эта глава раскрывает генераторы списков,которые я интенсивно использую далее в книге.В то время как Вы можете обойтись без использования генераторов списков,они предоставляют элегантный и эффективный способ реализации вещей.Глава 9(while циклы) важна.Глава 10 содержит группу различных тем, все из которых полезны,но многое может быть пропущено, если нужно.Последние четыре главы части I касаются словарей, текстовых файлов, функций и объектно-ориентированного программирования.

Часть II посвящена графике, в основном GUI программированию с Tkinter.Вы можете очень быстро написать хорошие программы применяя Tkinter.Например,раздел 15.7 представляет 20 строчную работающую, хотя и не идеально, игру "Крестики-нолики".Последняя глава части II немного охватывает Python Imaging Library.

Часть III содержит много веселых и интересных вещей,которые Вы можете делать с Python.Если Вы выстраиваете односеместровый курс по этой книге, Вы возможно захотите выбрать несколько тем в части III для просмотра.Эта часть книги может послужить как справочник или как место для заинтересованных и мотивированных студентов узнать больше.Все из тем в этой части книги есть вещи, которые я нашел полезными в тот или иной момент.

Хотя эта книга создавалась чтобы быть использованной в вводном курсе программирования, она также полезна тем, с начальным опытом программирования,желающим изучать Python.Если Вы один из этих людей, Вы должны быть способны пронестись через несколько первых глав.Вы возможно найдете часть II краткой, но не поверхностным,обращением в GUI программирование. Часть III содержит информацию об особенностях Python,которые позволят Вам совершать большие вещи с удивительно коротким кодом.

В подготовке этой книги Python документация с `www.python.org` была незаменимой.Эта книга была полностью набрана в LaTeX.Существует ряд LaTeXпакетов, особенно *listings* и *hyperref*, которые были крайне полезны.LaTeXкод из `http://blog.miliauskas.lt/` помог мне применить *listings* пакет для того,чтобы красиво выделить Python код.

Листинги длинных программ доступны на `https://www.brianheinold.net/python/`.Текстовые

файлы используемые в тексте и заданиях доступны на https://www.brianheinold.net/python/textfiles.html.У меня нет доступных решений к заданиям здесь, но имеется отдельный набор нескольких сотен заданий и решений на https://www.brianheinold.net/python/worked_exercises.html.

Пожалуйста присылайте комментарии, исправления, и предложения в *heinold@msmary.edu*.

Последнее обновление 18 марта 2022 года.

Часть I

Основы

Глава 1

Первые шаги

Эта глава введет Вас в Python, от его загрузки до написания простых программ.[1]

1.1 Установка Python

Идите на www.python.org и загрузите последнюю версию Python(версия 3.5 на момент написания). Python должен установиться легко. Если у Вас Mac или Linux,Python возможно уже установлен на компьютере, хотя он может быть ранней версии. Если он версии 2.7 или ранней, тогда Вам следует установить последнюю версию, так как многие программы в этой книге не будут правильно работать на старых версиях.

1.2 IDLE-Интегрированная среда разработки

IDLE это простая интегрированная среда разработки(IDE), которая идет с Python.Это программа, которая позволяет Вам вводить программы и запускать их. Имеются другие IDEs для Python,но сейчас я бы посоветовал придержаться IDLE, так как ею просто пользоваться. Вы можете найти IDLE в Python 3.4 папке на Вашем компьютере.

Когда Вы запускаете IDLE первый раз, она запускается в оболочке, которая является интерактивным окном, где Вы можете вводить Python код и видеть вывод в том же самом окне. Я часто использую оболочку вместо своего калькулятора или проверяю маленькие фрагменты кода. Но большую часть времени у Вас будет желание открыть новое окно и ввести там программу.

Заметка По крайней мере в Windows, если Вы нажимаете на Python файл на Вашем рабочем столе, система запустит программу, но не покажет код, это вероятно не то что Вы хотите. Наоборот, если Вы кликаете правой кнопкой на файл, то там должна быть опция называемая *Edit with IDLE*. Чтобы отредактировать существующий файл, либо сделайте это, либо запустите IDLE и откройте файл через *File* меню.

Комбинации клавиш Следующие комбинации работают в IDLE и могут реально ускорить Вашу работу.

Сочетание	Результат
CTRL+C	Копировать выбранный текст
CTRL+X	Вырезать выбранный текст
CTRL+V	Вставить
CTRL+Z	Отменить последнее сочетание или группу сочетаний
CTRL+SHIFT+Z	Повторить последнее сочетание или группу сочетаний
F5	Запустить модуль

1.3 Первая программа

Запустите IDLE и откройте новое окно (выберите *New Window* в меню *File*). Напечатайте следующую программу.

```
temp = eval (input('Введите температуру в Цельсиях: '))
print('Температура в Фаренгейтах равна', 9/5*temp+32)
```

Затем, в меню *Run*, выберите *Run Module* (или нажмите F5). IDLE попросит Вас сохранить файл, и Вам следует сделать так.Обязательно добавьте расширение *.py* к имени файла,т.к IDLE не будет автоматически его добавлять.Это укажет IDLE применять цвета, чтобы сделать Вашу программу легкочитаемой.
Как только Вы сохраните программу, она запустится в оболочке. Программа попросит Вас ввести температуру. Напечатайте 20 и нажмите ввод. Вывод программы выглядит как-то примерно так:

```
Введите температуру в Цельсиях: 20
Температура в Фаренгейтах равна 68.0
```

Давайте рассмотрим как программа работает. Первая строка просит пользователя ввести температуру. Задача функции **input** состоит в том, чтобы просить пользователя что-то ввести и зафиксировать это. Часть в скобках, строка-приглашение, которую пользователь видит. Она называется *string* и она будет отображаться пользователю программы точно как она появиться в самом коде. Функция **eval** это что-то, что мы применяем здесь, но зачем именно станет понятно позднее. Так что пока, просто запомните, что мы используем её когда получаем числовые данные. Нам надо дать имя значению, которое пользователь вводит, так чтобы программа могла запомнить его и использовать во второй строке. Используемое нами имя называется *temp* и мы применяем знак равно, чтобы присвоить *temp* значение пользователя.
Вторая строка использует функцию **print** для вывода результата преобразования.Часть в кавычках, другая строка(*string*), которая будет появляться в Вашей программе точно как она появиться в скобках здесь. Вторым аргументом функции **print** является результат.
Эта программа может показаться короткой и простой,чтобы быть полезной, но есть много вебсайтов, имеющих маленькие утилиты, которые делают подобные преобразования, и их код не намного сложнее чем код здесь.

Вторая программа Здесь программа вычисляющая среднее значение двух чисел, которые пользователь вводит:

```
num1 = eval (input('Введите первое число: '))
num2 = eval (input('Введите второе число: '))
print('Среднее чисел, которые Вы ввели равно', (num1 + num2)/2)
```

Для этой программы нам нужно взять два числа от пользователя. Есть способы сделать это в одну строку,но сейчас мы не будем усложнять вещи. Мы сразу берем числа и присваиваем им собственные имена. Единственная вещь обращающая внимание, это круглые скобки в вычислении среднего значения. Это из-за порядка выполнения операций. Все операции умножения и деления выполняются прежде операций сложения и вычитания.

1.4 Ввод данных

Регистр букв Регистр имеет значение. В Python *print,Print* и *PRINT* это разные вещи. Пока придерживайтесь строчных букв,так как большинство операторов Python в нижнем регистре.

Отступы Отступы имеют значения в начале строк,но не в других местах. Например, код ниже не будет работать.

```
temp = eval (input('Введите температуру в Цельсиях: '))
    print('Температура в Фаренгейтах равна', 9/5*temp+32)
```

Python использует отступы строк для вещей, о которых мы скоро узнаем. С другой стороны, отступы в большинстве других мест не имеют значения. Например, следующие строки имеют тот же самый эффект:

```
print('Привет мир')
print    ('Привет мир')
print(    'Привет мир')
```

По существу, компьютеры будут делать только то, что Вы им скажете, и они часто принимают вещи буквально. Python сам полностью полагается на такие вещи,как расстановка запятых и круглых скобок, таким образом, он знает что к чему. Это не очень хорошо к пониманию, что Вы имеете в виду, поэтому Вам нужно быть точным. Сначала это будет очень неприятно расставлять все скобки и запятые в нужных местах, но после некоторого времени это станет более привычно. Все равно, даже программируя длительное время, Вы все равно будете пропускать что-то. К счастью, интерпретатор Python достаточно хорош, чтобы помочь Вам найти свои ошибки.

1.5 Получение входных данных

Функция **input**, простой способ для Вашей программы получить информацию от людей, которые пользуются Вашей программой. Например:

```
name = input('Введите Ваше имя:' )
print('Привет, ', name )
```

Базовая структура

имя переменной = **input***(сообщение пользователю)*

Вышеприведенное работает для получения текста от пользователя. Получить числа от пользователя,применяемые в вычислениях, нам нужно сделать что-то дополнительное. Здесь пример:

```
num = eval( input('Введите число: '))
print('Квадрат Вашего числа:', num*num)
```

Функция **eval** преобразует текст введенный пользователем в число. Одна прекрасная особенность этого то, что Вы можете ввести выражения, в виде 3*12+5, и **eval** вычислит их для Вас.

Заметка Если Вы запустили свою программу, и кажется что ничего не происходит, попробуйте нажать клавишу enter. В IDLE есть небольшой глюк, который иногда происходить с оператором **input**.

1.6 Печать(Вывод)

Простой пример:

```
print('Привет всем')
```

Функция **print** требует заключения своих аргументов в круглые скобки. В программе выше,единственный аргумент, это строка 'Привет всем'. Что-либо внутри кавычек будет (за некоторыми исключениями) печататься точно так, как оно отображается. В следующих, первая инструкция выведет 3+4, в то время как вторая выведет 7.

```
print('3+4')
print(3+4)
```

Чтобы напечатать несколько элементов сразу, отделите их запятыми. Python автоматически вставит пробелы между ними.Ниже пример и вывод,который он выдает.

```
print('Значение 3+4 равно', 3+4)
print('A', 1, 'XYZ ', 2)
```

```
Значение 3+4 равно 7
A 1 XYZ 2
```

Необязательные аргументы

У функции **print** есть два необязательных аргумента. Они не слишком важны на этом этапе изучения,так что Вы спокойно можете пропустить эту часть, но они полезны,чтобы вывод выглядел привлекательно.

sep Python вставит пробел между каждым аргументом функции **print**. Имеется необязательный аргумент называемый *sep*, сокращение от separator(разделитель), который Вы можете использовать, чтобы изменить этот пробел на что-то другое. Например, применение *sep* = ':' будет разделять аргументы двоеточием, а *sep* = '## ' двойным знаком хэш.
Один особенно полезный вариант, ничего не вставлять в кавычки, как в *sep* = ''. Это говорит о том, чтобы между аргументами не было разделения. Здесь пример, где *sep* применяется, что бы вывод выглядел хорошо:

```
print('Значение 3+4 равно ', 3+4, '. ')
print('Значение 3+4 равно ', '. ', sep='')
```

```
Значение 3+4 равно 7 .
Значение 3+4 равно 7.
```

end Функция print автоматически переносит строку. Например, следующее будет выведено в две строки:

```
print('На первой строке ')
print('На второй строке')
```

```
На первой строке
На второй строке
```

Имеется необязательный аргумент *end*, который можете использовать с функцией **print**, чтобы при выводе строка не переносилась.

```
print('На первой строке', end='')
print('На второй строке')
```

```
На первой строкеНа второй строке
```

Конечно, это могло быть лучше выполнено с одним оператором **print**, но дальше мы увидим, что есть интересное применение аргумента *end*.

1.7 Переменные

Оглядываясь назад на нашу первую программу, мы видим использование переменной под названием *temp*:

```
temp = eval (input('Введите температуру в Цельсиях: '))
print('В Фаренгейтах равна', 9/5*temp+32)
```

Одно из главных предназначений переменной - это запоминать значение из одной части программы, так чтобы оно могло быть использовано в другой части программы. В случае выше, переменная *temp* хранит значение, которое пользователь ввел, для того чтобы мы могли делать вычисления с ним в следующей строке.

В примере ниже мы выполним вычисление и нам нужно применить его результат в нескольких местах программы. Если мы сохраним этот результат в переменной, то вычисление нам надо сделать только один раз. Это помогает сделать программу более читабельной.

```
temp = eval (input('Введите температуру в Цельсиях: '))
f_temp = 9/5*temp+32
print('В Фаренгейтах равна', 9/5*temp+32)
if f_temp > 212:
    print('Температура выше точки кипения.')
if f_temp < 32:
    print('Температура ниже точки замерзания.')
```

Мы пока не обсуждали операторы **if**, но они делают тоже самое,что Вы думаете.

Второй пример Здесь другой пример с переменными. Прежде чем читать дальше, попытайтесь узнать какие значения x и y будут после выполнения кода.

```
x= 3
y=4
z=x+y
z=z+1
x=y
y=5
```

После того как четыре строки кода выполнятся,x равняется 4, y=5, и z= 8. Один способ понять что-то подобное, это рассматривать каждую строку пошагово. Это особенно полезная техника для понимания более сложных частей кода. Здесь описание того, что происходит в коде выше:

1. x стартует со значением 3, а y со значением 4.
2.В строке 3 создается переменная z равная x+y, которая равна 7.
3.Затем значение z изменяется на единицу больше чем, текущее значение, изменяя его с 7 до 8.
4.Далее x изменяется на текущее значение y, которое равно 4.
5.Наконец, y изменяется на 5.Заметим, что это не влияет на x.
6.Поэтому в конце, x=4, y=5, а z=8

Имена переменных

Существуют несколько правил, которым следует придерживаться при присвоении имен Вашим переменным.

- Имена переменных могут содержать буквы,числа и подчеркивания.
- Имена переменных *не могут* содержать пробелы.
- Имена переменных *не могут* начинаться с числа.
- Регистр имеет значение - например, *temp* и *Temp* различны.

Поможет сделать Вашу программу более понятной, если Вы выберете имена, которые описательны, но не длинные, чтобы они не загромождали Вашу программу.

1.8 Задания

1.Напечатайте(выведите на печать) подобие в рамке внизу.

```
**********
**********
**********
**********
```

2.Напечатайте(выведите на печать) подобие в рамке внизу.

```
**********
*        *
*        *
**********
```

3.Напечатайте треугольник подобный в рамке внизу.

```
*
**
***
****
```

4.Напишите программу, которая вычисляет и печатает результат от $\frac{512-282}{47*48+5}$. Это примерно 0.1017.

5.Попросите пользователя ввести число. Напечатайте квадрат числа, но используйте необязательный аргумент *sep*, чтобы вывести его в полном предложении, которое заканчивается точкой. Образец вывода показан внизу.

```
Введите число: 5
Квадрат 5 равен 25.
```

6. Попросите пользователя ввести число *x*. Используйте необязательный аргумент *sep* для вывода x, 2x, 3x, 4x и 5x, каждый разделенный тремя тире, подобно внизу.

```
Введите число: 7
7− − −14− − −21− − −28− − −35
```

7.Напишите программу,которая просит пользователя ввести вес в килограммах и переводит его в фунты. В 1 килограмме 2.2 фунта.

8.Напишите программу, которая просит пользователя ввести три числа (используйте три отдельных оператора **input**). Создайте переменные называемые *total* и *average*, которые хранят сумму и среднее значение трех чисел и выведите значения *total* и *average*.

9.Многие телефоны имеют калькулятор чаевых. Напишите такой. Попросите пользователя ввести стоимость еды и процент чаевых, который они хотят оставить. Затем выведите оба, сумму чаевых и общий счет с включенными чаевыми.

Глава 2

Циклы for

Вероятно, самая мощная фишка компьютеров, то что они могут повторять что-то снова и снова очень быстро. Есть несколько способов повторять что-то в Python, самый распространенный из которых - цикл *for*.

2.1 Примеры

Пример 1 Следующая программа выведет *Hello* десять раз.

```
for i in range(10):
    print('Привет')
```

Структура цикла *for* показана ниже:

```
for имя переменной in range(число повторов):
    оператор, который повторяется
```

Здесь важен синтаксис. Слово for должно быть в нижнем регистре, первая строка должна заканчиваться двоеточием, а оператор, который повторяется должен иметь отступ. Использование отступа указывает Python, какие операторы будут повторены.

Программа 2 Программа ниже просит пользователя ввести число и выводит его квадрат, потом спрашивает другое число и выводит его квадрат, и так далее. Она делает это три раза и потом выводит, что цикл выполнен.

```
for i in range(3):
    num = eval(input('Введите число: '))
    print('Квадрат Вашего числа равен ', num*num)
print('Цикл сейчас выполнен.')
```

```
Введите число: 3
Квадрат Вашего числа равен 9
Введите число: 5
Квадрат Вашего числа равен 25
Введите число: 23
Квадрат Вашего числа равен 529
Цикл сейчас выполнен.
```

Так как вторая и третья строка имеют отступы, Python знает, что это выражения, которые нужно повторить. У четвертой строки нет отступа, так как она не часть цикла и выполняется только один раз, после того как цикл закончится.

Смотря на пример выше, мы видим откуда происходит термин *for цикл*: мы можем представить выполнение кода как - начинаясь с оператора for, продолжаясь во второй и третьей строках, потом возвращаясь назад в начало цикла к оператору for.

Пример 3 Программа ниже выведет A, затем B, потом она будет чередовать C и D пять раз и наконец закончит одной буквой E.

```
print('A')
print('B')
for i in range(5):
    print('C')
    print('D')
print('E')
```

Первые два оператора print выполняются по разу, выводя A, с последующей B. Затем C и D чередуются пять раз. Заметим, что мы не получим пять C с последующими пять D. Способ, которым цикл работает, такой, мы выводим C, затем D, далее возвращаемся в начало цикла, выводим C и другую D и так далее. Как только программа закончит цикл с C и D, она выведет одну E.

Пример 4 Если бы мы захотели, чтобы программа выше выводила пять C с последующими пятью D, вместо чередования C и D, мы могли бы сделать следующие:

```
print('A')
print('B')
for i in range(5):
    print('C')
for i in range(5):
    print('D')
print('E')
```

2.2 Переменная цикла

Одна часть for цикла немного странная, это - переменная цикла. В примере ниже переменная цикла, это переменная i. Выводом этой программы будут числа 0,1,...,99, каждое выведено на собственной строке.

```
for i in range(100):
    print(i)
```

Когда цикл запускается, Python устанавливает переменную i равную 0. Каждый раз, когда мы возвращаемся в цикл, Python увеличивает значение i на 1. Программа выполняет цикл 100 раз, каждый раз увеличивая значение i на 1, пока мы не пройдем цикл 100 раз. К этому моменту значение i равно 99.

Вы можете удивляться, почему i начинается с 0 вместо 1. Хорошо, кажется не было веской причины, кроме того, что начинать с нуля было полезно в ранние дни вычислительной техники и это прижилось у нас. В действительности, большинство вещей в компьютерном программировании начинаются с 0, вместо 1. К этому надо немного привыкнуть.

Поскольку, переменная цикла i увеличивается на 1, каждый раз проходя цикл, это может быть использовано для контроля количества прохождений цикла. Рассмотрите пример внизу:

```
for i in range(3):
    print(i+1,'-- Hello')
```

```
1 -- Hello
2 -- Hello
3 -- Hello
```

Имена Нет ничего особенного в имени i для нашей переменной. Программы ниже будут иметь тот же самый результат.

```
for i in range(100:)
    print(i)
```

```
for wacky_name in range(100:)
    print(wacky_name)
```

2.3 Функция range

Значение, которое мы передаем функции **range** определяет, как много раз мы будем повторять цикл. Метод с которым **range** работает, заключается в создании списка чисел от 0 до значения минус единица. Например,**range**(5) создает пять значений: 0,1,2,3, и 4.

Если мы хотим, чтобы список значений начинался со значения отличным от ноля, то мы можем сделать это определением начального значения. Оператор **range**(1,5) создаст список 1,2,3,4. Это выявляет одну особенность функции **range** - она останавливается раньше на единицу, чем мы думаем она должна. Если мы хотим, чтобы список содержал числа от 1 до 5(включая 5), тогда мы должны были сделать **range**(1,6).

Другая вещь, которую мы можем сделать, это давать списку значений повышаться за раз больше, чем на единицу. Делая это, мы можем определить необязательный шаг, как третий аргумент. Оператор **range**(1, 10, 2) будет проходить список с шагом 2, создавая 1,3,5,7,9.

Чтобы получить список со значениями в обратном порядке, мы можем использовать шаг -1. Например,**range**(5,1,-1) создаст значения 5,4,3,2, в порядке убывания. (Заметим,что функция **range** остановится на единицу раньше конечного значения 1). Далее немного больше примеров:

Оператор	Сгенерированные значения
range(10)	0,1,2,3,4,5,6,7,8,9
range(1,10)	1,2,3,4,5,6,7,8,9
range(3,7)	3,4,5,6
range(2,15,3)	2,5,8,11,14
range(9,2,-1)	9,8,7,6,5,4,3

Вот программа, которая делает обратный отсчет от 5 и потом выводит сообщение.

```
for i in range(5, 0, -1):
    print(i, end=' ')
print('Запуск!!! ')
```

```
5 4 3 2 1 Запуск!!!
```

Опция *end* = ' ' способствует тому, чтобы вывод осуществлялся на одной строке.

2.4 Более сложный пример

Давайте взглянем на задачу, в которой мы будем использовать переменную цикла. Программа ниже выводит прямоугольник из звезд, высотой 4 и шириной 6 колонок.

```
for i in range(4):
    print(i, end='* '* 6)
```

Прямоугольник, который создается этим кодом, показан ниже слева. Код ('* '* 6)- это то, что мы рассмотрим в разделе 6.2; он просто повторяет значок звездочка 6 раз.
(Примечание: Вместо символа '*' используйте звездочку на клавиатуре '* '.)

```
******                  *
******                  **
******                  ***
******                  ****
```

Предположим, мы хотим сделать треугольник. Мы можем достигнуть этого с очень маленьким изменением в программе для прямоугольника. Глядя на программу, мы можем увидеть, что цикл **for** будет повторять оператор **print** четыре раза, делая форму высотой 4 ряда. Это число 6, которое нужно будет изменить.

Смысл заключается в том, чтобы изменить число 6 на i+1. Каждый раз проходя цикл, программа сейчас будет печатать i+1 звездочек вместо 6. Переменная счетчика цикла i проходит значения 0,1,2, и 3. Использование этого позволяет нам варьировать число звездочек. Здесь программа создающая треугольник:

```
for i in range(4):
    print('* '* (i+1))
```

2.5 Задания

1.Напишите программу, которая выводит Ваше имя 100 раз.

2.Напишите программу, которая заполняет экран Вашим именем горизонтально и вертикально.[Подсказка: добавьте опцию $end = ''$ в оператор **print**, чтобы заполнить экран горизонтально.]

3.Напишите программу, которая выводит 100 строк, пронумерованных от 1 до 100, с Вашим именем на них. Вывод должен выглядить подобно выводу внизу.

```
1 Ваше имя
2 Ваше имя
3 Ваше имя
4 Ваше имя
. . .
100 Ваше имя
```

4.Напишите программу, которая распечатает список целых чисел от 1 до 20 и их квадраты. Вывод должен выглядеть как ниже:

```
1 – – – 1
2 – – – 4
3 – – – 9
. . .
20 – – – 400
```

5.Напишите программу, которая использует цикл **for**, чтобы вывести числа 8,11,14,17,20,...,83,86,89.

6.Напишите программу, которая использует цикл **for**, чтобы вывести числа 100,98,96,...,4,2.

7.Напишите программу, которая использует только 4 цикла **for**, чтобы вывести последовательность букв, как внизу.

```
AAAAAAAAAABBBBBBBCDCDCDCDEFFFFFFG
```

8.Напишите программу, которая просит пользователя ввести свое имя и сколько раз его напечатать. Программа должна вывести имя пользователя определенное число раз.

9.Числа Фибоначчи это последовательность чисел ниже, где первые два числа равны 1, а каждое следующее сумма двух предыдущих чисел. Напишите программу, которая спрашивает пользователя сколько чисел Фибоначчи вывести, а потом печатает то количество.

1,1,2,3,5,8,13,21,34,55,89...

10.Используя цикл **for** выведите на печать фигуру из звездочек в рамке внизу. Позвольте пользователю определить, какой высоты и ширины фигура должна быть. [Подсказка:**print**('$\star$'*10) напечатает 10 звездочек.]

```
★★★★★★★★★★★★★★★★★★★★
★★★★★★★★★★★★★★★★★★★★
★★★★★★★★★★★★★★★★★★★★
★★★★★★★★★★★★★★★★★★★★
```

11.Используя цикл **for** выведите на печать фигуру из звездочек в рамке внизу. Позвольте пользователю определить, какой высоты и ширины фигура должна быть.

```
********************
*                  *
*                  *
********************
```

12.Используя цикл **for** выведите на печать треугольник подобный в рамке внизу. Позвольте пользователю определить, какой высоты и ширины треугольник должен быть.

```
*
**
***
****
```

13.Используя цикл **for** выведите на печать перевернутый треугольник подобный в рамке внизу. Позвольте пользователю определить, какой высоты и ширины треугольник должен быть.

```
****
***
**
*
```

14.Используя цикл **for** выведите на печать фигуру, в виде бриллианта, как в рамке внизу. Позвольте пользователю определить, какой высоты и ширины фигура должна быть.

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

15.Напишите программу, которая печатает огромную букву A, как ниже. Позвольте пользователю определить, какого размера буква должна быть.

```
      *
     * *
    *****
   *     *
  *       *
```

Глава 3

Числа

Эта глава акцентирует внимание на числах и простой математике в Python.

3.1 Целые и десятичные числа

Из-за того, как компьютерные чипы разработаны, целые и десятичные числа представлены поразному на компьютерах. Десятичные числа представлены так называемыми числами с плавающими точками. Важно про них помнить,что Вы обычно получаете около 15 цифр точности(после точки). Это было бы прекрасно, если бы не было ограничений точности, но вычисления идут на много быстрее, если Вы отсечёте числа в какой-то момент.

С другой стороны, целые числа в Python не имеют ограничений. Они могут быть сколь угодно большими.

Для десятичных чисел,последняя цифра иногда немного отличается благодаря тому факту, что компьютеры работают в двоичной системе(основана на 2), в то время как наша человеческая система счисления основана на 10. Как пример, с математической точки зрения,мы знаем,что десятичное расширение 7/3 есть 2.333..., с повторяющимися бесконечно тройками. Но когда мы вводим 7/3 в оболочку Python, мы получим 2.3333333333333335. Это называется ошибкой округления. Для большинства практических задач это не имеет большого значения, но в действительности может вызвать проблемы для некоторых математических и научных вычислений. Если Вам реально понадобится большая точность, то есть решение. Смотри раздел 22.5

3.2 Math операторы

Вот список стандартных операторов в Python.

Оператор	Описание
+	сложение
–	вычитание
*	умножение
/	деление
**	возведение в степень
//	целочисленное деление
%	модуль(остаток)

Возведение в степень Python использует ** для возведения в степень. Знак вставки,^, применяется для другого.

Целочисленное деление Оператор целочисленного деления,//, требует некоторого объяснения. По сути, для положительных чисел он ведет себя подобно обычному делению, за исключением того, что он отбрасывает десятичную часть результата. Например,хотя 8/5 равно 1.6, мы имеем 8//5 равно 1. Использование этого оператора ма увидим позднее. Заметим, что во многих других языках программирования и в старых версиях Python, обычный оператор деления / фактически выполняет целочисленное деление на целых числах.

Деление по модулю Оператор деления по модулю, %, возвращает остаток от деления. Например, результат 18%7 равен 4, потому что 4 это остаток,когда 18 делится на 7. Этот оператор удивительно полезен. Например, число точно делится на n, когда оно оставляет в остатке 0, при делении на n. Таким образом, чтобы проверить является ли число n четным, посмотрите равно ли n%2 = 0.

Одно из его применений, если Вы хотите запланировать, чтобы что-то происходило в цикле каждый раз при его прохождении, Вы могли бы проверить равна ли переменная цикла по модулю 2 нулю i%2 = 0, и если да, тогда делайте что-то.

Оператор деления по модулю удивительно часто появляется в формулах. Если Вам нужно "обернуться"и возратиться к началу, то оператор деления по модулю полезен для этого. Например, что-то вроде часов. Если Вы проходите 6 часов после 8, то результат равен 2.(8+4=12(0) 0+2=2). Математически, это может быть достигнуто применением деления по модулю на 12, то есть (8+6)%12 равно 2.

Как другой пример, берем игру с игроками от 1 до 5. Скажем, у Вас есть переменная *игрок*, которая отслеживает текущего игрока. После 5 игрока снова следует игрок 1. Оператор деления по модулю может быть использован для этого:

$$игрок = игрок\%5 + 1$$

Когда *игрок* равен 5, *игрок*%5 будет 0 и выражение установит *игрок* равное 1.

3.3 Порядок операций

Операция возведение в степень выполняется первой, следом идут умножение и деление(включая // и %), а сложение и вычитание в последнюю очередь.

Это вступает в действие при вычислении среднего. Скажем, у Вас имеется три переменные x, y и z, и Вы хотите вычислить среднее их значений. Запись x+y+z/3 бы не сработала. Потому,что деление выполняется прежде сложения, в действительности Вы бы вычисляли $x + y + \frac{z}{3}$ вместо $\frac{x+y+z}{3}$. Это легко исправить применяя круглые скобки:(x+y+z)/3.

В общем, если Вы в чем-то не уверены, добавление скобок возможно поможет и обычно не причиняет какого-либо вреда.

3.4 Случайные числа

Чтобы сделать интересную компьютерную игру, неплохо добавить в неё некоторую случайность. Python идет с модулем, который называется *random*, позволяющий использовать случайные числа в наших программах.

Перед тем, как мы возьмём случайные числа, мы должны сначала объяснить, что такое *модуль*. Основная часть языка Python состоит из вещей типа - циклов for, операторов if, математических операторов и некоторых функций, типа **print** и **input**. Все остальное содержится в модулях, и

если мы хотим использовать что-то из модулей нам нужно сначала *import*(импортировать) это - то есть, сказать Python, что мы хотим применить это.

К этому моменту, есть только одна функция, называемая *randint*, которая нам будет нужна из модуля *random*. Чтобы загрузить эту функцию, мы используем следующее выражение:

```
from random import randint
```

Использовать *randint* просто: *randint (a, b)* возратит случайное целое число между *a* и *b*, включая оба *a* и *b*. (Обратите внимание, что *randint* включает правую конечную точку *b* в отличии от функции **range**). Далее короткий пример:

```
from random import randint
x = randint(1, 10)
print('Случайное число между 1 и 10: ', x)
```

```
Случайное число между 1 и 10: 7
```

Случайное число будет каждый раз разным, когда мы запускаем программу.

3.5 Math функции

Модуль math В Python имеется модуль называемый *math*, который содержит хорошо знакомые математические функции,включающие *sin, cos, tan, exp, log, log10, factorial, sqrt, floor* и *ceil*. Есть также обратные тригонометрические функции,гиперболические функции и постоянные *pi* и *e*.Здесь короткий пример:

```
from math import sin, pi
print('Число Пи приблизительно ', pi)
print('sin(0) = ', sin(0))
```

```
Число Пи приблизительно 3.14159265359
sin(0) = 0.0
```

Встроенные математические функции Есть две встроенные математические функции,**abs** (абсолютное значение) и **round**, которые доступны без импортирования модуля *math*. Ниже несколько примеров:

```
print(abs(-4.3))
print(round(3.336, 2))
print(round(345.2, -1))
```

```
4.3
3.34
350.0
```

Функция **round** принимает два аргумента: первый - округляемое число и второй - число десятичных знаков, до которых нужно округлить. Второй аргумент может быть отрицательным.

3.6 Получение помощи от Python

В Python имеется встроенная документация. Чтобы получить помощь о модуле *math*, для примера, зайдите в оболочку Python и введите следующие две строки:

```
>>>import math
>>>dir(math)
```

```
['_doc_', '_name_', '_package_', 'acos ', 'acosh ', 'asin ', 'asinh ', 'atan ', 'atan2 ',
'atanh ', 'ceil ', 'copysign ', 'cos ', 'cosh ', 'degrees ', 'e ', 'exp ', 'fabs ', 'factorial ', 'floor ',
'fmod ', 'frexp ', 'fsum ', 'hypot ', 'isinf ', 'isnan ', 'ldexp ', 'log ', 'log10 ', 'log1p ', 'modf
', 'pi ', 'pow ', 'radians ', 'sin ', 'sinh ', 'sqrt ', 'tan ', 'tanh ', 'trunc ']
```

Это дает список всех функций и переменных в модуле *math*. Вы можете игнорировать все те, которые начинаются с подчеркивания. Чтобы получить помощь о конкретной функции, скажем функции *floor*, Вы можете ввести **help**(math.floor). Ввод **help**(math) предоставит Вам помощь про содержимое модуля *math*.

3.7 Используя Shell как калькулятор

Оболочка Python может быть использована, как очень удобный и мощный калькулятор. Вот пример сеанса.

```
>>> 23**2
529
>>> s = 0
>>> for n in range(1,10001):
s = s + 1/n**2
>>> s
1.6448340718480652
>>> from math import *
>>> factorial(10)
3628800
```

Второй пример, здесь, суммирует числа $1 + 1/4 + 1/9 + \cdots + 1/10000^2$. Результат хранится в переменной *s*. Чтобы проверить значение этой переменной, просто введите её имя и нажмите ввод. Проверка переменных полезна для отладки Ваших программ. Если программа не работает правильно, Вы можете ввести имя Вашей переменной в оболочке после того, как программа завершится, чтобы проверить какие значения она имеет.

Выражение **from** math **import** * импортирует все функции из модуля *math*, который может сделать оболочку очень похожей на научный калькулятор.

Обратите внимание В меню оболочки выберите *Restart shell*, если Вы хотите удалить значения всех переменных.

3.8 Задания

1. Напишите программу, которая создает и выводит 50 случайных целых чисел, каждое между 3 и 6.

2. Напишите программу, которая создает случайное число x между 1 и 50, случайное число y между 2 и 5, и вычисляет x^y.

3. Напишите программу, которая создает случайное число между 1 и 10, и выводит Ваше имя то количество раз.

4. Напишите программу, которая создает случайное десятичное число между 1 и 10 с двумя десятичными знаками точности. Как примеры 1.23, 3.45, 9.80 и 5.00.

5. Напишите программу, которая создает 50 случайных чисел, так что первое число между 1 и 2, второе между 1 и 3, третье между 1 и 4,$\cdots$, и последнее между 1 и 51.

6. Напишите программу, которая просит пользователя ввести два числа x и y, и вычисляет$\frac{|x-y|}{x+y}$.

7. Напишите программу, которая просит пользователя ввести угол между $-180°$ и $180°$. Используя выражение с оператором, деление по модулю, преобразуйте угол в его эквивалент между $0°$ и $360°$.

8. Напишите программу, которая спрашивает пользователя количество секунд и выводит, сколько это в минутах и секундах. Например, 200 секунд, это 3 минуты и 20 секунд.[Подсказка: Используйте оператор //, чтобы получить минуты и оператор %, чтобы получить секунды.]

9. Напишите программу, которая спрашивает пользователя час между 1 и 12, и какое количество часов вперед от этого часа. Выведите какой час будет. Пример показан ниже.

```
Введите час: 8
Сколько часов вперед? 5
Новый час: 1
```

10.(а) Одним из способов узнать последнюю цифру числа, использовать деление по модулю числом 10. Напишите программу, которая просить пользователя ввести степень в виде числа. Затем найдите последнее число у двойки возведенной в ту степень.

(б) Одним из способов узнать последние две цифры числа, применить деление по модулю числом 100. Напишите программу, которая просить пользователя ввести степень в виде числа. Затем найдите последние 2 числа у двойки возведенной в ту степень.

(с) Напишите программу, которая просить пользователя ввести степень в виде числа и количество последних цифр. Найдите указанные цифры у двойки возведенной в степень.

11. Напишите программу, которая просить пользователя ввести вес в килограммах. Программа должна конвертировать его в фунты, выводя ответ округленный до ближайшего десятка фунтов.

12. Напишите программу, которая просить пользователя ввести число и выводит факториал того числа.

13. Напишите программу, которая просить пользователя ввести число и затем выводит синус, косинус и тангенс того числа.

14. Напишите программу, которая просить пользователя ввести угол в градусах и выводит синус того значения.

15. Напишите программу, которая выводит синус и косинус углов в диапазоне от 0 до 345° с шагом 15°. Каждый результат должен быть округлен до 4 знаков. Пример вывода показан внизу:

```
0     - - -     0.0       1.0
15    - - -     0.2588    0.9659
30    - - -     0.5       0.866
...
345   - - -    -0.2588    0.9659
```

16. Ниже описывается, как найти дату Пасхи в любой год. Несмотря на неопределенность появления, это не трудная проблема. Заметим, что $\lfloor \mathbf{x} \rfloor$ есть функция *floor*, которая для положительных чисел просто отбрасывает десятичную часть числа. Например, $\lfloor 3.14 \rfloor = 3$. Функция *floor* часть модуля *math*.

C = столетие($1900' \rightarrow C = 19$)
Y = год(все четыре цифры)
m = $(15 + \text{C} - \lfloor \frac{C}{4} \rfloor - \lfloor \frac{8C+13}{25} \rfloor) \bmod 30$
n = $(4 + C - \lfloor \frac{C}{4} \rfloor) \bmod 7$
a = Y mod 4
b = Y mod 7
c = Y mod 19
d = (19c + m) mod 30
e = (2a + 4b + 6d + n) mod 7

Пасха бывает, или март(22+d+e) или апрель(d+e-9). Имеется исключение, если d = 29 и e = 6.В этом случае, Пасха выпадает на одну неделю раньше(апрель 19). Есть другое исключение, если d = 28, e = 6, и m = 2,5,10,13,16,21,24 или 39. В этом случае, Пасха выпадает на одну неделю раньше(апрель 18). Напишите программу, которая просит пользователя ввести год и выводит дату Пасхи в этот год.

17. Год является високосным, если он делится на 4, за исключением, что года кратные 100 не високосные, если они также не делятся на 400. Попросите пользователя ввести год и используя оператор //, определите количество високосных годов между 1600 и этим годом.

18. Напишите программу, которой даётся количество сдачи, меньше чем $1, чтобы вывести сколько точно нужно будет монет номиналом 25,10,5 и 1 центов для той сдачи.[Подсказка: оператор // может быть полезен.]

19. Напишите программу, которая рисует ″модульный прямоугольник″, подобному внизу.Пользователь определяет ширину и высоту прямоугольника, а элементы начинаются с 0 и возрастают по стилю печатной машинки - слева направо и сверху вниз, только все сделано с применением оператора деления по модулю 10. Ниже примеры 3x5 и 4x8 прямоугольников.

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

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

Глава 4

Операторы if

Довольно часто в программах, мы хотим делать что-то только при условии, что верно что-то еще. Оператор if в Python, то что нам нужно.

4.1 Простой пример

Давайте попробуем программу на отгадывание числа. Компьютер выбирает случайное число, игрок пытается отгадать и программа говорит ему, если он угадал. Чтобы увидеть, что игрок угадал, нам нужно что-то новое, так называемое - *оператор if*

```
from random import randint
num = randint(1,10)
guess = eval(input('Введите свою догадку: '))
if guess == num:
print('Вы отгадали!')
```

Синтаксис оператора if очень похож на синтаксис оператора for в том, что в конце условия if имеется двоеточие, и следующие строка или строки вводятся с отступом. Строки с отступом будут выполняться только, если условие верно. С окончанием отступа if блок завершается.

Игра 'Отгадай число' работает, но она довольно проста. Если игрок отгадывает неправильно, ничего не происходит. Мы можем добавить к оператору if следующее.

```
if guess == num:
    print('Вы отгадали! ')
else:
    print('Извините. Число равно', num)
```

Мы добавили оператор else, который подобен ″иначе ″.

4.2 Условные операторы

Операторы сравнения ==, >, <, >=, <=, и !=. Последний обозначает *не равно*. Вот несколько примеров:

Выражение	Описание
`if x>3:`	если x больше чем 3
`if x>=3`	если x больше чем 3 или равно 3
`if x==3`	если x равно 3
`if x!=3`	если x не равно 3

Имеются три дополнительных оператора, используемых для построения более усложненных условий: **and**, **or** и **not**. Вот несколько примеров:

```
if grade>=80 and grade<90:
        print('Ваш уровень равен Б.')
if score>1000 or time>20:
        print('Игра окончена.')
if not (score>1000 or time>20):
        print('Игра продолжается.')
```

Порядок операций В плане порядка выполнения операций, **and** выполняется прежде **or**, так если у Вас есть усложненное условие, которое содержит оба, Вам могут понадобиться скобки вокруг условия **or**.

Представьте **and**, как подобие произведения, а **or**, как подобие сложения. Далее пример:

```
if (score<1000 or time>20) and turns_remaining == 0:
        print('Игра закончена.')
```

4.3 Общие ошибки

Ошибка 1 Оператор для равенства состоит из двух знаков равно. Это действительно общая ошибка забывать один из знаков равно.

Неправильно	Правильно
`if x=1:`	`if x==1:`

Ошибка 2 Общая ошибка, это использовать **and**, где необходимо **or** или наоборот. Рассмотрите следующие выражения if:

```
if    x>1    and    x< 100:
if    x>1    or     x< 100:
```

Первое выражение корректно. Если x, любое значение, между 1 и 100, тогда выражение будет верно. Идея этого в том, что значение x должно быть, *одновременно*, больше чем 1 **и** меньше чем 100. С другой стороны, второе выражение не то, что мы хотим, потому что, чтобы это было верно, **или x** должно быть больше чем 1, *или* x должно быть меньше чем 100. Но каждое число удовлетворяет этому.Вывод из этого здесь, если Ваша программа работает неправильно, проверяйте Ваши **and** и **or**.

Ошибка 3 Другая, очень общая ошибка, писать что-то подобное, как ниже:

```
if grade >= 80 and <90:
```

Это приведет к синтаксической ошибке. Мы должны быть точны. Правильное выражение

```
if grade >= 80 and grade <90:
```

С другой стороны, есть отличная короткая запись, которая работает в Python(хотя не работает во многих других языках программирования):

```
if 80<= grade <90:
```

4.4 elif

Простое использование инструкции if, это присваивание буквенной оценки. Предположим, что баллы 90 и выше - это 'А', баллы в 80х - 'Б', 70х - 'В', 60х - 'Г', и все что ниже это 'Д'. Здесь один из способов сделать это:

```
grade = eval (input('Введите Ваш балл: '))
if grade >= 90:
        print('А')
if grade >= 80 and grade<90:
        print('Б')
if grade >= 70 and grade<80:
        print('В')
if grade >= 60 and grade<70:
        print('Г')
if grade<60:
        print('Д')
```

Код выше достаточно простой и он работает. Однако, более элегантный способ сделать это показан ниже.

```
grade = eval (input('Введите Ваш балл: '))
if grade >= 90:
        print('А')
elif grade >= 80 :
        print('Б')
elif grade >= 70 :
        print('В')
elif grade >= 60 :
        print('Г')
else:
        print('Д')
```

С отдельными инструкциями if, каждое условие проверяется независимо от того, нужно ли оно на самом деле. То есть, если счет равен 95, первая программа выведет 'А', однако затем продолжит проверять условия, чтобы увидеть, если счет равен 'Б','В'и так далее, что немного затратно. Используя elif, как только мы найдем где счет соответствует условию, мы остановим проверку и пропустим весь путь до конца всего блока условий. Дополнительный выигрыш этого, то что условия, которые мы используем в инструкциях elif проще, чем в их if аналогах. Например,когда

используя **elif**, вторая часть второй инструкции **if** условия, grade<90, становится лишней, потому что соответствующий **elif** не должен переживать о счете 90 или выше, поскольку такой счет уже будет перехвачен первым выражением **if**.

Мы можем прекрасно обходиться без инструкции **elif**, но она часто может сделать Ваш код проще.

4.5 Задания

1. Напишите программу, которая просит пользователя ввести длину в сантиметрах. Если он вводит отрицательное значение длины, то программа должна сказать пользователю, что ввод неправильный. Иначе, программа должна преобразовать длину в дюймы и вывести результат. В дюйме 2.54 сантиметра.

2. Спросите у пользователя температуру. Потом спросите, в каких единицах измерения температура, Цельсиях или Фаренгейтах. Ваша программа должна преобразовать температуру в другое измерение. Преобразование $F = \frac{9}{5}C + 32$ и $C = \frac{5}{9}(F - 32)$.

3. Попросите пользователя ввести температуру в Цельсиях. Программа должна напечатать сообщение основанное на температуре:

- Если температура меньше, чем -273.15, напечатать, что температура неправильная, потому что она ниже абсолютного нуля.
- Если температура равна -273.15, напечатать, что она равна абсолютному нулю.
- Если температура между -273.15 и 0, напечатать, что температура ниже точки замерзания.
- Если она равна 0, напечатать, что она равна температуре точки замерзания.
- Если она между 0 и 100, напечатать, что температура в нормальном диапазоне.
- Если она равна 100, напечатать, что она равна температуре точки кипения.
- Если она выше 100, напечатать, что температура выше точки кипения.

4. Напишите программу, которая спрашивает пользователя, как много зачетных единиц он набрал.Если он набрал 23 или меньше, напечатать, что студент первокурсник. Если он набрал между 24 и 53, напечатать, что он второкурсник.Уровень для третьекурсника равен от 54 до 83, и для четверокурсника от 84 и выше.

5. Создайте случайное число между 1 и 10. Попросите пользователя отгадать число и выведите сообщение основанное на: угадал или нет.

6. Магазин продаёт товар по 12$ за предмет, если Вы покупаете меньше, чем 10 предметов. Если Вы покупаете от 10 до 99 предметов, цена составляет 10$ за предмет. Если Вы покупаете 100 или больше предметов, цена составляет 7$ за предмет. Напишите программу, которая спрашивает пользователя сколько он покупает предметов и выводит общую стоимость.

7. Напишите программу, которая просит пользователя ввести два числа и выводит *Близко*, если числа совпадают в пределах 0.001 или *Не близко* иначе.

8. Год является високосным, если он делится на 4, исключая что годы кратные 100 не являются високосными, если они также не делятся на 400. Напишите программу, которая просит пользователя ввести год и выводит, является ли он високосным или нет.

9. Напишите программу, которая просит пользователя ввести число и выводит все делители того числа. [Подсказка: оператор % используется для определения того, делится ли число на что-либо. см. раздел 3.2]

10. Напишите программу-игру для детей 'Умножение'. Программа должна давать игроку для решения десять, случайно сгенерированных, задач на умножение. После каждой, она должна говорить им, правильно ли они решили или нет, и какой ответ является правильным.

```
Вопрос 1: 3 x 4 = 12
Правильно!
Вопрос 2: 8 x 6 = 44
Неправильно. Ответ равен 48.
...
...
Вопрос 10: 7 x 7 = 49
Правильно.
```

11. Напишите программу, которая спрашивает у пользователя час между 1 и 12, просит ввести *am* или *pm*, и на сколько хочет идти вперед от этого часа. Выведите какой час это будет, указывая *am* или *pm* соответственно.

```
Введите час: 8
am (1) или pm (2)? 1
Сколько часов вперёд: 5
Новый час: 1 pm
```

12. Банка конфет на Хэллоуин содержит неизвестное количество конфет, и если Вы можете узнать точно сколько конфет в банке, тогда Вы выиграли все конфеты. Вы спрашиваете ответственное лицо следующее: если конфеты разделить поровну среди 5 людей, сколько штук осталось бы? Ответ 2 штуки. Потом Вы просите разделить поровну конфеты среди 6 людей, и оставшееся количество равно 3 штуки. Наконец, Вы просите разделить поровну конфеты среди 7 людей, и количество, которое осталось равно 2 штуки. Глядя в банку Вы можете сказать, что там имеется меньше, чем 200 штук. Напишите программу, которая определяет сколько конфет в банке.

13. Напишите программу, которая позволяет пользователю играть в 'Камень - Бумага - Ножницы' против компьютера. Должно быть 5 раундов, и после них, программа должна вывести, кто выиграл и проиграл или ничья.

Глава 5

Разные темы I

Эта глава состоит из нескольких распространенных приемов и некоторой другой полезной информации.

5.1 Подсчет

Очень часто мы хотим, чтобы наши программы считали, как много раз что-то происходит. Например, в видео игре может потребоваться отслеживать, сколько ходов игрок использовал, или математической программе потребуется подсчитать, сколько чисел имеют особые свойства. Ключом для подсчета является использование переменной для счета.

Пример 1 Эта программа принимает от пользователя 10 чисел и подсчитывает, как много из этих чисел больше, чем 10.

```
count = 0
for i in range(10):
    num = eval(input('Введите число: '))
    if num > 10:
        count = count + 1
print('Имеется', count,'чисел больше, чем 10.')
```

Представим переменную *count*, как если мы ведем счет на листе бумаги. Каждый раз, когда мы берем число больше, чем 10, мы добавляем 1 к нашему счету. В программе, это достигается строкой *count = count + 1*. Первая строка программы,count = 0, важная. Без неё, интерпретатор Python дошел бы к строке *count = count + 1* и выдал ошибку, говоря, что он не знает, что такое *count*. Это потому, что когда программа первый раз выполняет эту строку, она пытается сделать,то что строка говорит: взять прежнее значение *count*, добавить к нему 1, и сохранить результат в *count*. Но в первый раз, когда программа попадает туда, прежнего значения для использования нет, таким образом интерпретатор Python не знает что делать. Чтобы избежать ошибки нам нужно определить *count*, а это то, что первая строка делает. Мы устанавливаем переменную к 0, чтобы указать, что в начале программы числа больше, чем 10 не найдены.

Подсчет это очень распространённая вещь. Две вещи, которые вовлечены:

1. count = 0 — Начало счета с 0.
2. count = count + 1 — Увеличение счета на 1.

Пример 2 Эта модификация предыдущего примера подсчитывает, как много чисел, которые пользователь ввел, больше, чем 10, а также сколько равны 0. Чтобы подсчитать два параметра,

мы используем две переменные для счета.

```
count1 = 0
count2 = 0
for i in range(10):
    num = eval(input('Введите число: '))
    if num > 10:
        count1 = count1 + 1
    if num == 0:
        count2 = count2 + 1
print('Имеется', count1,'чисел больше, чем 10.')
print('Имеется', count2,'нулей.')
```

Пример 3 Далее, мы имеем слегка усложненный пример. Эта программа подсчитывает сколько квадратов чисел от 1^2 до 100^2 заканчиваются на 4.

```
count = 0
for i in range(1,101):
    if (i**2)% 10 == 4:
        count = count + 1
print( count)
```

Несколько напоминаний здесь: первое, из-за вышеупомянутой причуды функции **range**, нам нужно использовать **range**(1,101), чтобы выполнить цикл, пройти от 1 до 100.Переменная цикла i принимает эти значения, так квадраты от 1^2 до 100^2 представлены *i**2*. Затем, чтобы проверить оканчивается ли число на 4, хороший математический трюк должен проверить, оставляет ли число остаток 4, после деления на 10. Оператор деления по модулю, %, используется для получения остатка.

5.2 Суммирование

Тесно связанным с подсчетом является суммирование, когда мы хотим сложить группу чисел.

Пример 1 Эта программа будет суммировать числа от 1 до 100. Способ, которым это работает, заключается в том, что каждый раз, когда мы сталкиваемся с числом, мы добавляем его к нашему текущему результату s.

```
s = 0
for i in range(1,101):
        s = s + 1
print('Сумма равна ', s)
```

Пример 2 Эта программа, которая будет просить пользователя 10 чисел, а потом вычислять их среднее значение.

```
s = 0
for i in range(10):
```

```
    num = eval(input('Введите число: '))
    s = s + 1
print('Среднее равно', s/10)
```

Пример 3 Распространённым применением суммирования является ведение счета в игре. В начале игры, мы устанавливаем переменную счета равную 0. Потом, когда мы захотим увеличить счет, будем делать что-то подобное, как внизу:

```
score = score + 10
```

5.3 Обмен местами

Очень часто, мы захотим поменять местами значения двух переменных, x и y. Было бы заманчиво попытаться сделать следующее:

```
x = y
y = x
```

Только это не будет работать. Предположим, $x = 3$ и $y = 5$. Первая строка будет присваивать x значение 5, что нормально, но затем вторая строка также будет присваивать y значение 5, потому что x сейчас 5. Хитрость заключается в использовании третьей переменной, чтобы сохранить значение x.

```
hold = x
x = y
y = hold
```

Во многих языках программирования, это обычный способ поменять местами переменные. Python, однако, предоставляет хороший сокращенный способ.

```
x,y = y,x
```

Позднее, мы узнаем точно, почему это работает. А сейчас, смело используйте любой метод, какой предпочитаете. Последний метод, однако, имеет преимущество быть короче и легко понятным.

5.4 Переменные flag

Переменная flag может быть использована, чтобы дать знать одной части программы, когда что-то происходит в другой части программы. Здесь пример, который определяет, является ли число простым.

```
num = eval(input('Введите число: '))
flag = 0
for i in range(2,num):
    if num % i == 0:

            flag = 1
if flag == 1:
   print('Не простое ')
else:
   print('Простое ')
```

Вспомним, что число является простым, если оно не имеет делителей, кроме 1 и самого себя. Способ, которым программа выше работает,есть то, что *flag* стартует с 0.Затем, мы выполняем цикл от 2 до *num*-1. Если одно из этих значений окажется делителем, тогда *flag* станет равным 1. Как только цикл завершится, мы проверяем переключился ли флаг.Если да, мы знаем, имеется делитель, и *num* не простое. Иначе, число должно быть простым.

5.5 Максимумы и минимумы

Частая задача в программировании найти самое большое или самое маленькое значение в ряде значений. Здесь пример, где мы просим пользователя ввести десять положительных чисел, а затем выводим самое большое.

```
largest = eval(input('Введите положительное число: '))
for i in range(9):
    num = eval(input('Введите положительное число: '))
    if num > largest:
        largest = num
print('Самое большое число: ',largest)
```

Ключевым моментом здесь является переменная *largest*, которая отслеживает самое большое число, найденное до сих пор. Мы начинаем, устанавливая её равной первому числу пользователя. Затем, каждый раз, когда мы получаем новое число от пользователя мы проверяем, является ли это число больше, чем текущее самое большое число(которое хранится в *largest*). Если да, тогда мы устанавливаем *largest* равную этому значению.

Если, наоборот, мы захотим найти самое маленькое значение, единственным необходимым изменением является то, что $>$ становится $<$, хотя было бы неплохо переименовать переменную *largest* в *smallest*.

Позднее, когда мы придем к спискам, мы увидим короткий способ найти наибольшее и наименьшее значения, но прием выше полезно знать, поскольку Вы можете случайно оказаться в ситуации, где способ для списков не будет делать все, что Вам нужно сделать.

5.6 Комментарии

Комментарий—это сообщение тому, кто читает Вашу программу. Комментарии часто используются для описания, что делает секция кода или как она работает, особенно с запутанными секциями кода. Комментарии не влияют на Вашу программу.

Однострокoвые комментарии Для однострокового комментария используйте # символ.

```
# слегка хитрый способ получить два значения сразу
num1, num2 = eval(input('Введите два числа разделенными запятыми: '))
```

Вы можете поместить комментарии в конце строки:

```
count = count + 2 # each divisor contributes two the count
```

Многострокoвые комментарии Для комментариев, которые охватывают несколько строк, Вы можете использовать тройные кавычки.

```
"""Имя программы: Здравствуй мир
Автор: Brian Heinold
Дата: 1/9/11 """
print('Здравствуй мир')
```

Одним хорошим применением для тройных кавычек является закомментирование части Вашего кода. Порой Вам захочется изменять Вашу программу, но не удалять старый код, в случае если Ваши изменения не сработают.Вы сможете закомментировать старый код так, что он останется, если он понадобится Вам, и он будет проигнорирован, когда новая программа запустится.Здесь простой пример:

```
"""
print('Эта строка и следующая внутри комментария.')
print('Эти строки не будут выполнены.')
"""
print('Эта строка не в комментарии и она выполнится.')
```

5.7 Простая отладка

Здесь два простых метода, чтобы узнать почему программа не работает:

1.Используйте оболочку Python. После того, как программа запустилась, Вы можете вводить имена Ваших переменных программы, чтобы проверить их значения и увидеть, какие имеют значения, которые Вы ожидаете от них, а которые нет. Вы также можете применять оболочку для ввода небольших частей Вашей программы и увидеть работают ли они.

2.Добавьте инструкцию **print** в Вашу программу. Вы можете добавить их в любую точку программы, чтобы увидеть, какие значения переменных есть.Вы можете,также добавить инструкцию **print**, чтобы увидеть достигается еще ли точка в Вашем коде. Например,если Вы думаете, что сможете получить ошибку в условии инструкции **if**, Вы можете вставить инструкцию **print** в блок **if**, чтобы узнать вызывается ли условие.

Здесь пример части программы для определения простых чисел рассмотренной ранее. Мы вставляем инструкцию **print** в цикл **for**, чтобы точно увидеть, когда переменная флага устанавливается:

```
flag = 0
num = eval(input('Введите число: '))
for i in range(2,num):
    if num % i == 0:
        flag = 1
    print(i, flag)
```

3.Пустая инструкция **input**, подобной внизу, может быть использована для остановки программы в конкретный момент:

```
input()
```

5.8 Примеры программ

Способность читать код - это ценный навык. В этом разделе, мы заглянем в суть некоторых простых программ и попытаемся понять, как они работают.

Пример 1 Следующая программа выводит *Привет* случайное число раз в интервале от 5 до 25.

```
from random import randint
rand_num = randint(5, 25)
for i in range(rand_num):
    print('Привет')
```

Первая строка в программе является важной инструкцией. Это должно появиться всего лишь однажды, обычно ближе к началу Вашей программы. Следующая строка создает случайное число между 5 и 25. Чтобы повторить что-то 50 раз мы будем использовать **range**(50) в нашем цикле *for*. Чтобы повторить что-то 100 раз мы будем использовать **range**(100) в нашем цикле *for*. Чтобы повторить что-то случайное число раз, мы можем использовать **range**(*rand_num*),где *rand_num* является переменной, содержащей случайное число.Хотя если, мы захотим, можем пропустить переменную и вложить инструкцию *randint* прямо в функцию **range**, как показано ниже.

```
from random import randint
for i in range(randint(5,25)):
    print('Привет')
```

Пример 2 Сравните следующие две программы.

```
from random import randint

rand_num = randint(1,5)
for i in range(6):
    print('Привет'*rand_num)
```

```
from random import randint

for i in range(6):
    rand_num = randint(1,5)
    print('Привет'*rand_num)
```

```
Привет Привет
Привет Привет
Привет Привет
Привет Привет
Привет Привет
Привет Привет
```

```
Привет Привет Привет
Привет
Привет Привет Привет Привет
Привет Привет Привет
Привет Привет
Привет
```

Единственной разницей между программами является размещение инструкции rand_num. В первой, она она находится снаружи цикла *for*, и сохраняет то же самое значение во время выполнения программы. Таким образом, каждая инструкция **print** будет выводить *'Привет'* то же самое количество раз. Во второй программе, *rand_num* находится внутри цикла.Прямо перед каждой инструкцией, *rand_num* присваивается новое случайное число и поэтому выводимое количество *'Привет'* будет изменяться от строке к строке.

Пример 3 Давайте напишем программу, которая создает 10000 случайных чисел от 1 до 100, и подсчитает сколько из них кратно 12. Здесь, то что нам понадобится:

- Из-за того, что мы используем случайные числа, первая строка программы должна импортировать модуль *random*.
- Нам потребуется запустить цикл *for* 10000 раз.
- Внутри цикла нам нужно создать случайное число, проверить на кратность 12 и если так, добавить 1 к переменной счета *count*
- Перед тем, как мы начнем считать, нам нужно будет установить переменную счета *count* равной нулю.
- Чтобы проверить делимость на 12, мы используем оператор деления по модулю %.

Когда мы соберем все это вместе, получим следующее:

```
from random import randint
count = 0
for i in range(10000):
    num = randint(1, 100)
    if num% == 0:
        count = count + 1

print('Число чисел кратных 12:', count)
```

Отступ имеет значение

Общей ошибкой является неправильный отступ. Предположим, мы возмем вышеописанную программу и сделаем отступ последней строки. Программа по-прежнему будет запускается, но не так, как ожидалось.

```
from random import randint
count = 0
for i in range(10000):
    num = randint(1, 100)
    if num% == 0:
        count = count + 1
    print('Число чисел кратных 12:', count)
```

Когда мы запускаем её, она выводит целую группу чисел. Причиной этого является то, что отступом инструкции **print**, мы сделали её частью цикла **for**, поэтому инструкция **print** будет выполняться 10000 раз.

Предположим, мы отступим инструкцию **print** на один шаг дальше, как внизу.

```
from random import randint
count = 0
```

```
for i in range(10000):
    num = randint(1, 100)
    if num% == 0:
        count = count + 1
        print('Число чисел кратных 12:', count)
```

Сейчас она не только часть цикла **for**, но также и часть инструкции **if**. Что будет происходить каждый раз, когда мы найдем новое кратное 12, это то ,что мы будем печатать переменную *count*. Ни это, ни предыдущий пример не то, что мы хотим. Мы хотим сразу напечатать количество в конце программы, поэтому нам совсем не нужно делать отступ к инструкции **print**.

5.9 Задания

1. Напишите программу, которая считает как много квадратов чисел от 1 до 100 заканчиваются на 1.

2. Напишите программу, которая считает как много квадратов чисел от 1 до 100 заканчиваются на 4 и как много заканчиваются на 9.

3. Напишите программу, которая просит пользователя ввести значение n, а затем вычисляет $(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) - ln(n)$. Функция ln - это *log* в модуле *math*.

4. Напишите программу, которая вычисляет сумму $1 - 2 + 3 - 4 + \cdots + 1999 - 2000$

5. Напишите программу, которая просит пользователя ввести число и выводит сумму делителей того числа. Сумма делителей числа важная функция в теории чисел.

6.Число называется *идеальным числом*, если оно равно сумме всех его делителей, не включая само число. Например, 6 является идеальным числом, потому что делители 6 - это, 1,2,3,6 и $6 = 1+2+3$. Другой пример, 28 является идеальным числом, потому что его делители - 1,2,4,7,14,28 и $28 = 1 + 2 + 4 + 7 + 14$. Однако, 15 не будет идеальным числом, потому что его делители - 1, 3, 5, 15 и $15 \neq 1 + 3 + 5$. Напишите программу, которая находит все 4 идеальных числа, которые меньше чем 10000.

7. Целое число называется бесквадратным, если оно не делится никакими квадратными числами(идеальными квадратами - 1,4,9,16 и т.д.), кроме 1. Например, 42 бесквадратное, потому что его делители - 1,2,3,6,7,21 и 42, и не одно из этих чисел(исключая 1) не являются квадратными числами. С другой стороны, 45 квадратное, потому что оно делится на 9, которое является идеальным квадратом. Напишите программу, которая просит пользователя ввести целое число и говорит ему, является ли оно бесквадратным или нет.

8. Напишите программу, которая меняет местами значения трех переменных x,y и z, так чтобы x получала значение y, y получала значение z, а z получала значение x.

9. Напишите программу, которая подсчитывает сколько целых чисел от 1 до 1000 не являются - идеальными квадратами, идеальными кубами, или идеальными в пятой степени.

10. Попросите пользователя ввести 10 результатов теста. Напишите программу, которая делает следующее:

(a) Выводит самый высокий и самый низкий результат.

(б) Выводит среднее значение результатов.

(в) Выводит второй самый большой результат.

(г) Если какой-либо результат больше, чем 100, тогда после того, как все результаты будут введены, выводит сообщение предупреждающее пользователя, что было введено значение свыше 100 .

(д) Отбрасывает два самых низких результата и выводит среднее из оставшихся значений.

11. Напишите программу, которая вычисляет факториал числа. Факториал, n, числа - это число n — произведение всех его целых чисел между 1 и n, включая n. Например, $5 = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. [Подсказка: Пытайтесь использовать эквивалент умножения техники суммирования.]

12. Напишите программу, которая просит пользователя угадать случайное число в интервале от 1 до 10. Если он угадает правильно, то получит 10 очков, добавленных к его счету, и потеряет 1 очко за неправильную догадку. Предоставьте пользователю отгадать пять чисел и выведите его счет после того, когда все попытки отгадать выполнены.

13. В последнем разделе было задание, которое просило Вас создать игру на умножение для детей. Улучшите Вашу программу из того задания, отслеживанием числа правильных и неправильных ответов. В конце программы выведите сообщение, которое изменяется в зависимости от того, сколько правильных ответов игрок дал.

14. Это задание посвящено хорошо известной Монти Холл (Monty Hall) задаче. В ней, Вы участник игрового шоу. Ведущий показывает три двери.За одной из этих дверей приз, а за двумя другими козы. Вы выбираете дверь. Ведущий, который знает за какой дверью лежит приз, затем распахивает одну из дверей за которой нет приза. Остались две двери, и ведущий дает Вам возможность изменить свое решение. Должны Вы оставить ту же самую дверь, поменять дверь или это не имеет значение?

(а) Напишите программу, которая имитирует процесс этой игры 10000 раз и вычисляет, сколько раз в процентах, Вы бы выиграли, если поменяли решение и наоборот, если бы не меняли.

(б) Попробуйте то же самое, но с четырьмя дверями вместо трех.По прежнему имеется только один приз, и ведущий по прежнему распахивает одну дверь, а потом даёт Вам возможность поменять решение.

Глава 6

Строки

Строки - это тип данных для работы с текстом. Python имеет ряд мощных функций для управления строками.

6.1 Основы

Создание строк Строка создается заключением текста в кавычки. Вы можете использовать, или одиночные кавычки,('), или двойные кавычки ("). Тройные могут быть применены для многострочных строк. Вот несколько примеров:

```
s = 'Привет'
t = "Привет"
m = '''Это длинная строка, которая
расположена на двух строчках.'''
```

Ввод Вспомним из главы 1, что когда получаем числовой ввод, мы используем инструкцию **eval** с инструкцией **input**, но когда получаем текст, мы не используем **eval**. Различия показаны ниже:

```
num = eval(input('Введите число : '))
string = input('Введите строку')
```

Пустая строка Пустая строка '' - это строка эквивалентная числу 0. Это строка с пустыми кавычками. Мы её видели прежде в необязательном аргументе инструкции **print**, *sep* = ''.

Длина Чтобы получить длину строки(как много в ней символов), используйте встроенную функцию **len**. Например, **len**('Привет') равняется 6.

6.2 Объединение и повторение

Операторы + и * (умножение) могут быть применены на строки. Оператор + объединяет две строки. Этот оператор называется *конкатенация* (слияние). * повторяет строку определенное число раз. Далее несколько примеров.

Выражение	Результат
'AB' + 'cd'	'Abcd'
'A' + '7' + 'B'	'A7B'
'Hi' * 4	'HiHiHiHi'

Пример 1 Если мы захотим напечатать длинный ряд тире, то можем сделать следующее

```
print('-' * 75)
```

Пример 2 Оператор + может быть использован для построения строки, шаг за шагом, аналогично способу, которым мы подсчитывали и суммировали в разделах 5.1 и 5.2. Далее пример, который многократно просит пользователя ввести букву и выстраивает строку состоящую только из гласных, которые он вводит.

```
s = '!'
for i in range(10):
        t = input('Введите букву: ')
        if t == 'a' or t == 'e' or t == 'и' or t == 'о' or t == 'и' or t == 'у':
            s = s + t
print(s)
```

Этот способ очень полезен.

6.3 In оператор

Оператор in используется, чтобы сказать, что строка содержит что-либо. Например:

```
if 'a' in string:
        print('Ваша строка содержит букву а.')
```

Вы можете объединить in с оператором not, чтобы сказать, что строка не содержит что-либо:

```
if ';' not in string:
        print('Ваша строка не содержит точку с запятой.')
```

Пример В предыдущем разделе у нас было длинное условие if

```
if t == 'а' or t == 'е' or t == 'и' or t == 'о' or t == 'ю' or t == 'у':
```

Используя оператор in мы можем заменить эту инструкцию следующей:

```
if t in 'аоеиую':
```

6.4 Индексация

Нам часто захочется выделить отдельные символы из строки. Python использует квадратные скобки, чтобы сделать это. Таблица ниже дает некоторые примеры индексирования строки s = 'Python'.

Инструкция	Результат	Описание
s[0]	P	Первый символ строки s
s[1]	y	второй символ строки s
s[-1]	n	последний символ строки s
s[-2]	o	второй с конца символ строки s

- Первым символом s является s[0], не s[1]. Запомните, что в программировании, счет обычно начинается с 0, не 1.
- Отрицательные индексы считаются в обратном порядке, от конца строки.

Общая ошибка Предположим, s = 'Python' и мы пытаемся выполнить s[12]. В строке только 6 символов и Python выдаст следующее сообщение об ошибке:

```
IndexError: string index out of range
```

Вы будете видеть это сообщение снова. Запомните, что это происходит, когда Вы пытаетесь читать за концом строки. (Всего можно индексировать 6 символов, индексы от 0 до 5)

6.5 Срезы

Срез используются,чтобы выбрать часть строки. Он работает как комбинация индексирования и функция **range**. Ниже, мы имеем несколько примеров со строкой s = 'abcdefdhij'.

index:	0	1	2	3	4	5	6	7	8	9
letters:	a	b	c	d	e	f	g	h	i	j

Код	Результат	Описание
s[2:5]	cde	символы с индексами 2,3,4
s[:5]	abcde	первые пять символов
s[-2:]	ij	последние два символа
s[:]	abcdefghij	вся строка
s[1:7:2]	bdf	символы с индексами 1-6, через два
s[: :-1]	jihgfedcba	отрицательный шаг меняет на обратный порядок

- Основная структура

  имя строки[начальное положение : конечное положение + 1]

  Срезы имеют ту же самую особенность, что и функция **range**, в том что они не включают конечное местоположение. Например, в примере выше, s[2:5] дает символы с индексами 2,3 и 4, но не символ с индексом 5.

- Мы можем оставить пустым, или начальное, или конечное положение. Если мы оставляем начальное пустым, то по умолчанию это начало строки. Так s[:5] дает первые пять символов s. Если мы оставляем конечное пустым, по умолчанию это конец строки.Так s[5 :] будет давать символы от индекса 5 до конца. Если мы используем отрицательные индексы, мы можем получать конечные символы строки. Например, s[-2 :] дает последние два символа.

- Имеется третий необязательный аргумент, точно как у инструкции **range**, который может определять шаг. Например, s[1:7:2] проходит строку шагом два, выбирая символы с индексами 1,3 и 5(только не 7, из-за вышеупомянутой особенности). Самым полезным шагом является -1, который проходит через строку наоборот, изменяя порядок символов.

6.6 Изменение отдельных символов строки

Предположим, у нас строка называемая *s* и мы хотим изменить символ этой строки с индексом 4 на 'X'. Заманчиво попробовать s[4] = 'X', но это к сожалению не будет работать. Строки в Python *неизменяемы*,это означает,что мы не можем изменять никакую часть из них. Больше об этом в разделе 19.1. Если мы хотим изменить символ строки s,нам придётся создать новую из s и переназначить её к s. Здесь код, который изменит символ с индексом 4 на 'X':

```
s = s[:4] + 'X' + s[5:]
```

Смысл этого в том, что мы берем все символы до индекса 4, затем X, а потом все символы после индекса 4.

6.7 Итерация

Очень часто, мы захотим просканировать строку символ за символом. Цикл for, который показан ниже может быть использован, чтобы сделать это. Он проходит строку, называемую s, выводя символ за символом, каждый на отдельной строке:

```
for i in range(len(s)):
    print(s[i])
```

В инструкции **range** мы имеем **len**, которая возвращает длину s. Так, если s имеет длину 5 символов, это было бы то же самое, что **range**(5) и переменная цикла i проходила бы от 0 до 4. Это означает, что s[i] будет проходить символы s. Этот способ прохождения цикла полезен, если нам нужно отслеживать наше местонахождение в строке в течение цикла.

Если нам не нужно отслеживать нашу локацию в строке, тогда мы можем использовать простой тип цикла:

```
for c in s:
    print(c)
```

Этот цикл будет проходить строку s, символ за символом, с *c*, которая хранить текущий символ. Вы можете приблизительно прочитать это, как предложение: 'Для каждого символа *c* в s, вывести этот символ.'

6.8 Методы строк

Строки идут с массой *методов*, функциями, которые возвращают информацию о строке или возвращают новую строку, которая является измененной версией оригинала. Здесь, некоторые из самых полезных:

Метод	Описание
lower()	возвращает строку с каждой буквой оригинала в нижнем регистре
upper()	возвращает строку с каждой буквой оригинала в верхнем регистре
replace(x,y)	возвращает строку с каждым встреченным x замененным на y
count(x)	считает число встреченных x в строке
index(x)	возвращает местонахождение первого встреченного x
isalpha()	возвращает **True**, если каждый символ строки является буквой

Важное замечание Одним очень важным замечанием о *lower*, *upper* и *replace* является, то что они не изменяют оригинальную строку. Если Вы хотите изменить строку s, перевести все в нижний регистр, недостаточно просто использовать s.lower(). Вам нужно сделать следующее:

```
s = s.lower()
```

Короткие примеры Здесь несколько примеров использования методов строк.

Инструкция	Описание
print(s.count(' '))	выводит число пробелов в строке
s = s.upper()	переводит строку в верхний регистр
s = replace('Hi','Hello')	заменяет каждый 'Hi' в s на 'Hello'
print(s.index('a'))	выводит индекс места первой 'a' в s

isalpha Метод *isalpha* применяется, чтобы сказать является ли символ строки буквой или нет. Он возвращает **True**, если символ буква и **False** иначе. Когда применяется ко всей строке, он будет возвращать **True** только, если каждый символ строки буква. Значения **True** и **False** называются булевыми и рассматриваются в разделе 10.2. Пока же, просто запомните, что Вы можете применять *isalpha* в условиях *if*. Здесь простой пример:

```
s = input('Введите строку ')
if s[0].isalpha():
    print('Ваша строка начинается с буквы ')
if not s[0].isalpha():
    print('Ваша строка не содержит буквы ')
```

Замечание о index Если Вы попытаетесь найти индекс чего-либо, чего нет в строке, Python покажет ошибку. Например, если s = 'abc' и Вы попробуете s.index('z'), то получите ошибку. Один способ обойти это, надо проверить сначала, как внизу:

```
if 'z' in s:
    location = s.index('z')
```

Другие методы строк Имеется гораздо больше методов строк. Например, есть методы *isdigit* и *isalnum*, которые аналогичны *isalpha*. Некоторые другие полезные методы, о которых мы узнаем позднее, есть *join* и *split*. Чтобы увидеть список всех методов строк, наберите **dir**(**str**) в оболочке Python. Если Вы сделали это, то увидите набор имен, которые начинаются с _. Вы можете игнорировать их. Чтобы прочитать документацию Python для одного из методов, скажем методе *isdigit*, наберите *help(str.isdigit)*.

6.9 Символы перехода

Обратная косая черта используется для получения определенных специальных символов, называемых символы перехода, в Вашей строке. Имеются различные символы перехода, и здесь самые полезные:

- \n *символ новой строки.*Он используется для перехода на новую строку. Здесь пример:

```
print('Привет\n\nВсем!')
```

```
Привет
Всем!
```

- \' для вставки апострофов в строки.Скажем, у Вас есть следующая строка:

```
s = 'I can't go '
```

Это создаст ошибку, потому что апостроф фактически закончит строку. Вы можете использовать \', чтобы обойти это:

```
s = 'I can\'t go
```

Другой опцией является использование двойных кавычек для строки:

```
"s = I can\'t go"
```

- \" аналогично \'.
- \\Это применяется, чтобы получить сам обратный слеш. Например:

```
filename = 'c:\\programs\\file.py'
```

- \t символ табуляции.

6.10 Примеры

Пример 1 Легким способом напечатать пустую строку является - **print**(). Однако, если мы захотим напечатать десять пустых строк, нижеследующий быстрый способ делает это:

```
print('\n' * 9)
```

Обратите внимание, что мы получаем одну из десяти строк из самой функции **print**.

Пример 2 Напишите программу, которая просит пользователя ввести строку и выводит местоположение каждой 'a' в строке.

```
s = input('Введите текст ')
for i in range(len(s)):
    if s[i] == 'a':
        print(i)
```

Мы используем цикл, чтобы просканировать строку по одному символу за раз. Переменная цикла *i* отслеживает наше местоположение в строке, и s[i] дает символ в том месте. Таким образом, третья строка проверяет каждый символ на соответствие 'a' и если это так, то выведет *i*, местоположение той 'a'.

Пример 3 Напишите программу,которая просит пользователя ввести строку и создает новую строку, которая удваивает каждый символ оригинальной строки. Например, если пользователь введет *Hello*, вывод должен быть *HHeelloo*.

```
s = input('Введите текст ')
doubled_s = ''
for c in s:
        doubled_s = doubled_s + c * 2
```

Здесь мы можем использовать второй тип цикла из раздела 6.7. Переменная *c* пройдет символы *s*. Мы применяем оператор повторения, *, чтобы удвоить каждый символ. Мы выстраиваем строку *s*, способом описанным в конце раздела 6.2

Пример 4 Напишите программу,которая просит пользователя ввести свое имя и выводит его в следующей шутливой форме:

```
Э   Эл   Элв   Элви   Элвис
```

Нам потребуется цикл, потому что мы должны периодически выводить часть строки, и чтобы вывести часть строки, мы будем использовать срез:

```
s = input('Введите свое имя: ')
for i in range(len(name)):
        print(name[:i+1], end='')
```

Одна хитрость заключается в использовании переменной цикла *i* в срезе. Так как число символов, которое нам нужно выводить изменяется, нам нужно переменное количество в срезе. Это напоминает программу для треугольника из раздела 2.4. Мы хотим выводить один символ имени за цикл в первый раз, два символа второй раз и так далее.Переменная цикла *i* начинается с 0 в первый проход цикла, потом увеличивается на 1 во второй проход цикла и т.д. Таким образом, мы используем *name[:i+1]*, чтобы напечатать первые *i+1* символы имени. Наконец, чтобы полученные срезы вывести на той же самой строке, мы используем опциональный аргумент функции **print** *end* =''.

Пример 5 Напишите программу, которая удаляет все заглавные буквы и общие знаки пунктуации из строки *s*.

```
s = s.lower()
for s in ', . ; : - ? !() \' " ':
        s = s.replace(c,'')
```

Способ, которым это работает для каждого символа в строке знаков препинания, является то, что мы заменяем каждое его вхождение в *s* на пустую строку ''. Здесь одно техническое замечание: Нам нужен ' символ в строке. Как описано в предыдущем разделе, мы получаем его в строке, использованием символом перехода \'.

Пример 6 Напишите программу, которая получает строку содержащую десятичное число, выводит десятичную часть числа. Например, если получает 3.14159, программа должна вывести 0.14159.

```
s = input('Введите Ваше десятичное число: ')
print(s[s.index('.'):])
```

Ключевым моментом здесь является метод *index*, который определяет, где находится десятичная часть. Она начинается там и идет к концу строки, поэтому мы используем срез, который начинается с *s.index('.')*

Здесь другой, более математический способ делать это:

```
from math import floor
num = eval(input('Введите Ваше десятичное число: '))
print(num - floor(num))
```

Одним отличием между способами является то, что первая создает строку, тогда как вторая число.

Пример 7 Простой и очень старый способ отправки секретных сообщений является шифр замены.

```
alphabet = ('абвгдеёжзийклмнопрстуфхцчшщъыьэюя ')
key = 'дцвфыяжканлюхгиосцбетшуёмчээйпръ '

secret_message = input(('Введите Ваше сообщение: '))
secret_message = secret_message.lower()

for c in secret_message:
    if c.isalpha():
        print(key[alphabet.index(c)], end = '')
    else:
)       print(c, end = ''))
```

Строка *key* - это случайным образом переставленный алфавит.

Хитрость заключается в цикле *for*. Проходя сообщение символ за символом, каждая буква находящая в ней заменяется на соответствующую из *key*. Это достигается использованием метода *index*, который находит позицию в алфавите текущей буквы и заменяет её буквой из *key* с той же самой позицией. Все небуквенные символы копируются как есть. Программа использует метод *isalpha*, который говорит, является ли символ буквой или нет.

Код для дешифрации сообщения почти тот же самый. Просто измените *key[alphabet.index(c)]* на *alphabet[key.index(c)]*.Раздел 19.11 предоставляет различные подходы к шифру замены.

6.11 Задания

1. Напишите программу, которая просит пользователя ввести строку. Затем программа должна вывести следующее:

(а) Общее число символов в строке.

(б) Повтор строки 10 раз.

(в) Первый символ строки(помните, что индексация строки начинается с 0).

(г) Первые три символа строки.

(д) Три последние символа строки.

(е) Строку наоборот.

(ё) Седьмой символ строки, если она достаточной длины или сообщить, что иначе.

(ж) Строку с удаленным первым и последним символами.

(з) Строку в верхнем регистре(Заглавные буквы).

(и) Строку с каждой *a* замененной на *e*.

(й) Строку, в которой каждая буква заменена пробелом.

2. Простой способ оценить количество слов в строке- это подсчитать число пробелов в строке. Напишите программу, которая просит пользователя ввести строку и возвращает подсчет количества слов в строке.

3. Люди часто забывают закрывать круглые скобки когда пишут формулы. Напишите программу, которая просит пользователя ввести формулу, и напечатать, имеет ли формула то же самое количество открывающих и закрывающих скобок.

4. Напишите программу, которая просит пользователя ввести слово и выводит, содержит ли это слово гласные буквы.
5. Напишите программу, которая просит пользователя ввести строку. Программа должна создать новую строку, называемую *new_string*, из строки пользователя, так чтобы второй символ был заменен на звездочку(*) и три восклицательных знака были добавлены в конец строки. Наконец вывести *new_string*. Типичный вывод показан ниже:

```
Введите Вашу строку: Старт
С*арт!!!
```

6. Напишите программу, которая просит пользователя ввести строку s, а потом преобразует её в нижний регистр(прописные буквы), удалит все точки и запятые, и выведет получившуюся строку.

7. Напишите программу, которая просит пользователя ввести слово и определит, является ли оно палиндромом или нет. Палиндром - это слова, которое читается одинаково слева направо справо налево.

8. В определенных школах, email адреса студентов заканчиваются на *@student.college.edu*, в то время как email адреса профессоров заканчиваются на *@prof.college.edu*. Напишите программу, которая сначала спрашивает пользователя сколько email адресов он будет вводит, а потом просит ввести эти адреса. После того, как все адреса будут введены, программа должна вывести сообщение указывающее на то, что все введенные адреса студентов или там были адреса профессоров.

9. Спросите у пользователя число, а потом напечатайте следующее, где образец оканчивается на число, которое пользователь ввёл.

```
1
 2
  3
   4
```

10. Напишите программу, которая просит пользователя ввести последовательность символов(строку), затем выводит каждую удвоенную букву и на отдельной строке. Например, если пользователь ввел

ОКЕЙ, вывод был бы

ОО
КК
ЕЕ
ЙЙ

11. Напишите программу, которая просит пользователя ввести слово и содержит букву а. Программа затем должна напечатать следующие две строки: на первой строке должна быть часть слова до и включающая первую букву а, и на второй строке должна быть оставшаяся часть. Образец вывода показан ниже:

Введите слово: буффало
буффа
ло

12. Напишите программу, которая просит пользователя ввести слово, а затем делает заглавной каждую вторую букву того слова. Так, если пользователь вводит *носорог*, программа должна вывести *нОсОрОг*.

13. Напишите программу, которая просит пользователя ввести две строки одинаковой длины. Программа, затем, должна проверить, имеют ли строки одинаковую длину. Если нет, программа должна напечатать соответствующее сообщение и выйти. Если они одинаковой длины, программа должна чередовать символы двух строк. Например, если пользователь ввел *абвгд* и *АБВГД*, программа должна вывести АаБбВвГгДд.

14. Напишите программу, которая просит пользователя ввести свое имя прописными буквами, а затем сделать заглавной первую букву каждого слова его имени.

15. Когда я был ребенком, мы бывало играли в эту игру, называемую *Mad Libs*. Способ, которым она работала был таким: друг спрашивал меня некоторые слова, и потом они вставлялись в определенные места текста и читали его. Текст, часто бывало, становился достаточно смешным со словами, которые я дал, так как я не имел понятия о чем был текст. Слова были, обычно, из определенной категории, как место, животное и так далее.
Для этой задачи вы напишите *Mad Libs* программу. Сначала, Вы должны составить текст и исключить некоторые слова. Ваша программа должна просить пользователя ввести некоторые слова и сказать какие типы слов ввести. Затем вывести полный текст с вставленными словами.Здесь маленький пример, но Вы должны использовать свой собственный (длинный) пример:

Введите название урока: МАТАНАЛИЗ
Введите прилагательное: ВЕСЕЛЫЙ
Введите занятие: ИГРА В БАСКЕТБОЛ

Сегодня урок МАТАНАЛИЗА был действительно ВЕСЕЛЫМ.Сегодня мы изучали
на уроке, как ИГРАТЬ В БАСКЕТБОЛ. Я не могу дождаться завтрашнего урока.

16. Компании часто пытаются персонализировать свои предложения(оферты), чтобы сделать их более привлекательными. Один несложный способ сделать это, просто вставить имя человека в различные места предложения. Конечно, компании не вручную впечатывают каждое имя человека, все генерируется компьютером. Напишите программу, которая просит пользователя ввести свое имя, а потом генерирует оферту, как показано ниже. Простоты ради, Вы можете предположить, что имя и фамилия человека состоят из одного слова каждое.

Уважаемый Джордж Вашингтон,

Я рад предложить вам нашу новую бонусную карту Platinum Plus со специальной вступительной ставкой 47,99% годовых. Джордж, подобное предложение поступает не каждый день, поэтому я настоятельно призываю вас позвонить сейчас по бесплатному номеру 1-800-314-1592. Мы не можем долго предлагать такую низкую цену, Джордж, поэтому звоните прямо сейчас.

17. Напишите программу, которая генерирует 33 строчный блок букв, частично показан ниже. Используйте цикл содержащий одну или две инструкции **print**.

```
абвгдеёжзийклмнопрстуфхцчшщъыьэюя
бвгдеёжзийклмнопрстуфхцчшщъыьэюяа
вгдеёжзийклмнопрстуфхцчшщъыьэюяаб
. . .
юяабвгдеёжзийклмнопрстуфхцчшщъыьэ
яабвгдеёжзийклмнопрстуфхцчшщъыьэю
```

18. Цель этого задания увидеть, сможете ли Вы имитировать поведение оператора **in** и методов *count* и *index* используя только переменные, циклы **for** и инструкции **if**.

(а) Без использования оператора **in** напишите программу, которая просит ввести пользователя строку и букву, и выводит, появляется или нет буква в строке.

(б) Без использования метода *count* напишите программу, которая просит пользователя ввести строку и букву, и подсчитать, сколько появлений буквы имеется в строке.

(в) Без использования метода *index*, напишите программу, которая просит пользователя ввести строку и букву, и выводит указатель первого появления буквы в строке. Если буквы нет в строке, программа должна сообщить об этом.

19. Напишите программу, которая просит пользователя ввести большое целое число и вставляет запятые в него, в соответствии со стандартным Американским соглашением для запятых в больших числах. Например, если пользователь вводит 1000000, вывод должен быть 1,000,000.

20.Напишите программу, которая переводит время из одной часовой зоны в другую. Пользователь вводит время в стандартном Американском способе, таком как 3:48pm или 11:26am. Первый часовой пояс, который вводит пользователь, соответствует исходному времени, а второй - желаемому часовому поясу. Возможные часовые зоны - это Восточный(UTC-05),Центральный(UTC-06), Горный(UTC-07) и Тихоокеанский(UTC-08).

```
Время: 11:48pm
Начальный пояс: Тихоокеанский
Конечный пояс: Восточный
2:48am
```

21. Анаграмма слова - это слово, созданное перестановкой букв исходного слова. Например, двумя анаграммами слова *idle* являются *deli* и *lied*. Нахождение анаграмм, которые есть реальные слова, за пределами нашей достигаемости до главы 12. Вместо этого, напишите программу, которое просит пользователя ввести слово(строку) и возвращает случайную анаграмму строки - другиим словами, произвольной перестановкой букв той строки(слова).

22. Простым способом шифрования сообщений является перестановка его символов. Один из способов перестановки символов - это, выбрать символы с четными индексами, вставить их в шифрованную строку, а за ними символы с нечетными индексами. Например, строка *message* была бы зашифрована, как *msaeesg*, потому что четные символы - *m,s,a,e(* с индексами 0,2,4,6), а нечетные символы - *e,s,g*(с индексами 1,3,5).

(а) Напишите программу, которая просит пользователя ввести строку и используя этот метод зашифровать её.

(б) Напишите программу, которая расшифровывает строку, зашифрованную этим методом.

23. Более общей версией вышеописанной техники является *шифр ограждения*, где вместо разложения объектов на четные и нечетные, они разбиваются - тройками, четверками или чем то большим. Например, в случае троек, строка *secret message* была бы разбита в три группы. Первая группа - *sr sg*,символы с индексами 0,3,6,9,12. Вторая группа - *eemse*,символы с индексами 1,4,7,10,13. Последняя группа - *ctea*, символы с индексами 2,5,8,11. Зашифрованное сообщение - *sr sgeemsectea*.

(а) Напишите программу, которая просит пользователя ввести строку и использует шифр ограждения в случае тройки, чтобы зашифровать строку.

(б) Напишите программу для расшифровки в случае тройки.

(в) Напишите программу, которая просит пользователя ввести строку и целое число, определяющее, или разбивать объекты тройками, четверками, или чем-то еще. Зашифруйте строку используя шифр ограждения.

(г) Напишите программу для расшифровки в общем случае.

24. В математическом анализе, производная x^4 равняется $4x^3$. Производная x^5 равняется $5x^4$. Производная x^6 равняется $6x^5$. Эта закономерность продолжается. Напишите программу, которая просит пользователя ввести данные типа x^3 или x^{25} и вывести производную. Например, если пользователь вводит x^3, программа должна вывести $3x^2$.

25. В алгебраических выражениях, символ умножения часто пропускают, как в $3x+4y$ или $3(x+5)$. Компьютеры предпочитают, чтобы эти выражения включали символ умножения, как в $3 * x + 4 * y$ или $3 * (x + 5)$. Напишите программу, которая просит пользователя ввести алгебраическое выражение, а затем вставить символ умножения, где это уместно.

Глава 7

Списки

Скажем, нам нужно получить тридцать результатов теста от пользователя и что-то с ними сделать, например, упорядочить. Мы можем создать тридцать переменных - *score1, score2,. . . , score30*, но это было бы очень утомительно. Упорядочить, тогда результаты было бы крайне сложно. Решение для этого - использовать списки.

7.1 Основы

Создание списков Вот простой список:

```
L = [1, 2, 3]
```

Используйте квадратные скобки для обозначения начала и конца списка, и отделяйте элементы запятыми.

Пустой список Пустой список обозначается [ ]. Он эквивалентен 0 или ''

Длинные списки Если Вам нужно ввести длинный список, Вы можете разделить его на несколько строк, как внизу:

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
        32, 33, 34, 35, 36, 37, 38, 39, 40]
```

Ввод Мы можем применить **eval**(**input**()), чтобы позволить пользователю ввести список. Далее пример:

```
L = eval(input('Введите список: '))
print('Первый элемент равен',L[0])
```

```
Введите список: [5, 7, 9]
Первый элемент равен 5
```

Выведение списков Вы можете использовать функцию **print**, чтобы вывести все содержание списка.

```
L = [1, 2, 3]
print(L)
```

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

Типы данных Списки могут содержать все типы объектов, даже другие списки. Например, следующее является допустимым списком:

```
L = [1,  2.718,  'абв ', [5,6,7]]
```

7.2 Сходство со строками

Имеется целый ряд вещей, которые работают для списков так же, как для строк.

- **len** — Число элементов в L определяется **len**(L).
- **in** — Оператор **in** говорит Вам, если список что-то содержит. Ниже примеры:

```
if 2 in L:
    print('Ваш список содержит число 2. ')
if 0 not in L:
    print('Ваш список не имеет нулей.')
```

- Индексация и разделение — Это работает точно также, как со строками. Например, L[0] первый элемент списка, а L[:3] дает первые три элемента.
- *index* и *count* — Эти методы работают также, как и для строк.
- + и * — Оператор + добавляет один список к концу другого. Оператор * повторяет список. Вот примеры:

Выражение	Результат
[7, 8] + [3, 4, 5]	[7, 8, 3, 4, 5]
[7, 8] * 3	[7, 8, 7, 8, 7, 8]
[0] * 5	[0, 0, 0, 0, 0]

Последний пример особенно полезен для быстрого создания списка нулей.

- Итерация(цикл) — Те же самые типы циклов, которые работают со строками, также работают со списками. Оба следующих примера выводят элементы списка, один за одним на отдельных строках.

```
for i in range(len(L)):
    print(L[i])
```

```
for item in L:
    print(item)
```

Левый тип цикла полезен для условий, когда Вам нужно применять переменную цикла *i*, чтобы отследить, где Вы находитесь в цикле. Если это не надо, тогда используйте правый тип цикла, так как он немного проще.

7.3 Встроенные функции

Имеется несколько встроенных функций, которые работают со списками. Здесь некоторые полезные:

Функция	Описание
len	возвращает количество элементов списка
sum	возвращает сумму элементов списка
min	возвращает минимальное значение элемента списка
max	возвращает максимальное значение элемента списка

Например, следующее выражение вычисляет среднее из значений списка L:

```
среднее = sum(L) / len(L)
```

7.4 Методы списков

Здесь некоторые методы списков:

Метод	Описание
append(x)	добавляет x в конец списка
sort()	сортирует список
count(x)	возвращает число появлений x в списке
index(x)	возвращает индекс первого места появления x
reverse()	возвращает список в противоположном порядке

Важное замечание Имеется большое различие между методами списка и методами строки: Методы строки не изменяют оригинальную строку, но методы списка изменяют начальный список. Чтобы отсортировать список L просто используйте *L.sort()*, а не L = L.sort(). По факту, последнее не будет работать совсем.

неправильно	правильно
s.replace('X','x')	s = s.replace('X','x')
L = L.sort()	L.sort()

Другие методы списка Имеются некоторые другие методы списка. Введите **help**(**list**) в оболочке Python, чтобы увидеть документацию по ним.

7.5 Прочее

Создание копий списков Создание копий списков немного сложнее благодаря способу, которым Python обрабатывает списки. Скажем, у нас есть список L и мы хотим сделать копию списка и назвать его M. Выражение M = L не будет работать по причинам, которые рассматриваются в разделе 19.1.На данный момент делаем следующее вместо M = L:

```
M = L[ : ]
```

Изменение списков Изменение конкретного элемента в списке происходит легче, чем в строке. Чтобы изменить значение в местоположение 2(индекс 2) в списке L на 100, мы просто скажем L[2] = 100. Если мы захотим вставить значение 100 в местоположение 2 без перезаписи текущего значения, мы можем использовать метод *insert*. Чтобы удалить запись из списка, мы можем применить оператор del. Некоторые примеры показаны ниже. Принимайте L = [6, 7, 8] для каждой операции.

Операция	Новый L	Описание
L[1] = 9	[6, 9, 8]	замена элемента с индексом 1 на 9
L.insert(1,9)	[6, 9, 7, 8]	вставка 9 с индексом 1 без замены
del L[1]	[6, 8]	удаляет второй элемент
del L[:2]	[8]	удаляет первые два элемента

7.6 Примеры

Пример1 Напишите программу, которая генерирует список 50 случайных чисел между 1 и 100.

```
from random import randint:
L = [ ]
for in range(50):
        L.append(randint(1,100))
```

Мы использовали метод *append*, чтобы наращивать список по элементу за раз, начиная с пустого списка, [ ]. Альтернативой использования *append* является следующее:

```
L = L + [randint(1, 100)]
```

Пример2 Замените каждый элемент списка его квадратом.

```
for i in range(len(L)):
        L[ i ] = L[ i ] ** 2
```

Пример3 Подсчитайте сколько элементов в списке L больше, чем 50.

```
count = 0
for item in L:
        if item > 50:
            count = count + 1
```

Пример4 Дан список L, который содержит числа от 1 до 100, создайте новый список, чей первый элемент равняется количеству единичек в списке L, чей второй элемент равен количеству двоек в списке L, и так далее.

```
frequences = [ ]
for i in range(1, 101):
```

```
    frequences.append(L.count(i))
```

Решающим является метод *count*, который говорит сколько раз что-то появляется в списке.

Пример5 Напишите программу, которая выводит два самых больших и два самых маленьких элементов списка, называемого *scores*

```
scores.sort()
print('Два самых маленьких : ', scores[0], scores[1])
print('Два самых больших : ', scores[-1], scores[-2])
```

Как только мы отсортируем список, самые маленькие значения будут в начале, а самые большие в конце.

Пример6 Здесь программа для игры в простую викторину.

```
num_right = 0
# Вопрос 1
print('Столица Франции ?', end = ' ')
quess = input()
if guess.lower() == 'париж':
    print('Правильно!')
    num_right += 1
else:
    print('Неправильно.Ответ - Париж.')
print('У Вас', num_right,'из 1 правильно')

# Вопрос 2
print('Какой штат в США имеет только одного соседа?', end = ' ')
quess = input()
if guess.lower() == 'мэн':
    print('Правильно!')
    num_right += 1
else:
    print('Неправильно.Ответ - Мэн.')
print('У Вас', num_right,'из 2 правильно')
```

Код работает, но он очень громоздкий. Если мы захотим добавить больше вопросов, нам нужно копировать и вставлять один из этих блоков кода, а потом изменить кучу вещей. Если мы решим изменить один из вопросов или порядок вопросов, тогда там будет включено громадное количество переписываний. Если мы решим изменить дизайн игры, например, не сообщать пользователю правильный ответ, тогда каждый одиночный блок кода должен быть переписан.Чрезмерный код подобный этому может быть,часто, сильно упрощен применением списков и циклов.

```
questions = ['Столица Франции ?','Какой штат в США имеет только одного
соседа? ']
answers = ['Париж','Мэн']
```

```
num_right = 0
for  i  in  range(len(questions)):
    quess = input(questions[ i ])
    if guess.lower() == answer[ i ].lower():
        print('Правильно!')
        num_right = num_right + 1
    else:
        print('Неправильно.Ответ - ', answers[ i ])
    print('У Вас', num_right,'из', i + 1,'правильно')
```

Если Вы внимательно посмотрите на этот код, то Вы увидете, что код в цикле почти тот же самый, как код одного из блоков в предыдущей программе, за исключением, что в инструкциях, где мы выводим вопросы и ответы, мы используем *questions[i]* и *answers[i]* вместо текста самих вопросов.

Это демонстрирует общую технику: если Вы обнаружите, что повторяете тот же самый код снова и снова, пробуйте списки и циклы for. Несколько частей Вашего повторяющегося кода, который изменяется, то место куда пойдет код списка. Преимуществом является,то что изменяя вопрос, добавляя вопрос, или изменяя порядок, необходимо изменить только списки *questions* и *answers*. Также, если Вы захотите сделать изменения в программе, по типу не сообщать пользователю правильный ответ, тогда все что нужно сделать, это изменить одну строку, вместо двадцати копий той строки во всей программе.

7.7 Задания

1. Напишите программу, которая просит пользователя ввести список целых чисел. Сделайте следующее:

(а) Выведите общее число элементов в списке.

(б) Выведите последний элемент списка.

(в) Выведите список в обратном порядке.

(г) Выведите *Да*, если список содержит 5 и *Нет* иначе.

(д) Выведите число пятерок в списке.

(е) Удалите первый и последний элементы из списка, отсортируйте оставшиеся и выведите результат.

(ё) Выведите сколько целых чисел в списке меньше, чем 5.

2. Напишите программу, которая генерирует список из 20 случайных чисел от 1 до 100.

(а) Выведите список.

(б) Выведите среднее элементов в списке.

(в) Выведите самое большое и самое маленькое значения в списке.

(г) Выведите второе самое большое и второе самое маленькое значения в списке.

(д) Выведите сколько четных чисел в списке.

3.Начните со списка $[8, 9, 10]$. Сделайте следующее:

(а) Установите второй элемент (индекс 1) равный 17.

(б) Добавьте 4,5 и 6 в конец списка.

(в) Удалите первый элемент из списка.

(г) Отсортируйте список.

(д) Удвойте список.

(е) Вставьте 25 на место с индексом 3

Окончательный список должен равняться $[4, 5, 6, 25, 10, 17, 4, 5, 6, 10, 17]$

4. Попросите пользователя ввести список содержащий числа от 1 до 12. Затем замените все элементы в списке, которые больше 10, на 10.

5.Попросите пользователя ввести список строк. Создайте новый список, состоящий из этих строк, у которых первые символы удалены.

6. Создайте следующие списки используя цикл for .

(а) Список состоящий из целых чисел от 1 до 49.

(б) Список содержащий квадраты целых чисел от 1 до 50.

(в) Список ['а','бб','ввв','гггг', . . .], который заканчивается с 33 копиями буквы *я*.

7. Напишите программу, которая принимает два списка L и M одного размера, и суммирует их элементы вместе, формируя новый список N, чьи элементы являются суммой соответствующих элементов в L и M. Например, если $L = [3, 1, 4]$ и $M = [1, 5, 9]$ тогда N должен равняться $[4, 6, 13]$.

8. Напишите программу, которая просит пользователя ввести целое число и создает список, состоящий из факторов(делителей), того целого числа.

9. Когда играя игру, где Вы должны бросать кубики(кости), хорошо знать шансы каждого броска. Например, шансы выпадения 12 составляют около 3 %, а шансы выпадения 7 около 17%. Вы можете вычислить их математически, но если не знаете математику, Вы можете написать программу, которая делает это. Чтобы сделать это, Ваша программа должна имитировать бросание двух кубиков около 10000 раз, и вычислить, и вывести процент выпадений, который выходит при $2, 3, 4, \ldots, 12$.

10. Напишите программу, которая перемещает элементы списка так, что элемент с индексом 1 передвигается на место с индексом 2, элемент с индексом 2 на место с индексом 3 и так далее, а элемент с последним индексом на место с первым индексом.

11. Используя цикл for , создайте список как внизу, который состоит из единичек разделенных многими нулями. Последние две единицы в списке должны быть разделены десятью нулями.

$[1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, \ldots]$

12. Напишите программу, которая генерирует 100 случайных целых чисел, которые равны или 0, или 1. Потом найдите самый длинную последовательность нулей, самое большое число нулей в ряду. Например, самая длинная последовательность нулей в $[1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0]$ равна 4.

13. Напишите программу, которая удаляет повторяющиеся элементы из списка так, чтобы каждый элемент появлялся один раз. Например, список $[1, 1, 2, 3, 4, 3, 0, 0]$ бы стал $[1, 2, 3, 4, 0]$.

14. Напишите программу, которая просит пользователя ввести длину в футах. Затем программа должна предоставить пользователю опцию, чтобы конвертировать из футов в дюймы, ярды, мили, миллиметры, сантиметры, метры или километры. Скажем, если пользователь введет 1, программа преобразует в дюймы, если он введет 2, тогда программа конвертирует в ярды и так далее. Хотя это может быть сделано с инструкциями if , со списками будет намного короче, а также легче добавлять новые преобразования, если Вы используете списки.

15. Существует, вероятно невзламываемый шифр, называемый одноразовый блокнот. Способ, которым он работает, является то, что Вы сдвигаете каждый символ сообщения на случайное число от 1 до 33, обходя алфавит если необходимо. Например, если текущий символ *у* и сдвиг равен 5,тогда новый символ равен *ш*. Каждый символ получает собственный сдвиг, поэтому нужно столько случайных сдвигов, сколько символов в сообщении. Как пример, предположим пользователь ввел *секрет*. Программа должна сгенерировать случайный сдвиг от 1 до 33 для каждого символа. Предположим, случайно сгенерированные сдвиги $1, 3, 2, 10, 8$ и 2. Зашифрованное сообщение было бы *тзмъмф*.

(а) Напишите программу, которая просит пользователя ввести сообщение и зашифровать его применяя одноразовый блокнот. Сначала преобразуйте строку в прописные буквы. Любые пробелы и знаки пунктуации в строке должны быть оставлены без изменения. Например, *Секрет!!!* становится *тзмъмф!!!* используя сдвиги выше.

(б) Напишите программу, которая дешифрует строку, зашифрованную способом выше.

Причиной, по которой это называется одноразовым блокнотом, является то, что список сдвигов должен быть использован сразу. Он становится легко взламываемым, если те же самые случайные сдвиги используются для более чем одного сообщения. Более того, он вероятно невзламываемый, если случайные числа являются действительно случайными. Для этой проблемы просто используйте *randint*, а для криптографически безопасных случайных чисел смотри раздел 22.8

Глава 8

Больше со списками

8.1 Списки и random модуль

Существуют полезные функции в модуле *random*, которые работают со списками.

Функция	Описание
choice(L)	выбирает случайный элемент из L
sample(L, n)	выбирает группу n случайных элементов из L
shuffle(L)	перемешивает элементы в L

Замечание Функция *shuffle* изменяет оригинальный список, поэтому если Вы не хотите изменения Вашего списка Вам надо будет сделать его копию.

Пример 1 Мы можем применить *choice*, чтобы выбрать имя из списка имен.

```
from random import choice
names = ['Джо','Боб','Сью','Салли']
current_player = choice(names)
```

Пример 2 Функция *sample* сходна с *choice*. Тогда как *choice* выбирает один элемент из списка, *sample* может быть использована для нескольких.

```
from random import choice
names = ['Джо','Боб','Сью','Салли']
team = sample(names, 2)
```

Пример 3 Функция *choice* также работает со строками, выбирая случайный символ из строки. Здесь пример, который использует *choice*, чтобы заполнить экран множеством случайных символов.

```
from random import choice
s = 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя1234567890!^#$ %&*()'
for  i in range(10000):
    print(choice(s), end = '')
```

Пример 4 Здесь полезное использование *shuffle* для выбора случайного порядка игроков в игре.

```
from random import shuffle
players = ['Джо','Боб','Сью','Салли']
shuffle(players)
for p in players:
    print(p,'Это твоя очередь')
    # здесь идет код для игры...
```

Пример 5 Здесь мы применяем *shuffle*, чтобы разделить группу людей в команды по два. Предположим, нам дали список называемый *names*.

```
shuffle(players)
teams = [ ]
for i in range(0, len(names), 2):
    teams.append([names[ i ], names[ i + 1]])
```

Каждый элемент в *teams* является списком двух имен. Способ, которым код работает следующий - мы перемешиваем имена так, чтобы они встали в случайном порядке. Первые два имени в перемешанном списке становятся первой командой, следующие два имени становятся второй командой, соответственно. Заметим, что мы используем третий необязательный аргумент **range**, чтобы пропустить два имени в списке.

8.2 split

Метод *split* возвращает список слов строки. Метод предполагает, что слова разделены пробелами, которые могут быть или пробелами, табуляцией или символами новой строки. Вот пример:

```
s = 'Привет! Это тест.'
print (s.split())
```

```
['Привет!', 'Это', 'тест.']
```

Как мы можем видеть, поскольку *split* разбивает строку по пробелам, знаки пунктуации будут частью слов. Существует модуль, называемый *string*, который содержит, среди других вещей, строковую переменную, называемую *punctuation*, которая содержит общие знаки пунктуации. Мы можем удалить пунктуацию из строки *s* следующим кодом:

```
from string import punctuation
for c in punctuation:
    s = s.replace(c,'')
```

Пример Здесь программа, которая подсчитывает сколько раз определенное слово появляется в строке.

```
from string import punctuation
s = input ('Введите строку: ')
for c in punctuation:
    s = s.replace(c,'' )
s = s.lower()
L = s.split()

word = input ('Введите слово: ')
print (word,'появляется',L.count(word), 'раз.')
```

Необязательный аргумент Метод *split* принимает необязательный аргумент, который позволяет ему разделять строку в местах отличных от пробелов. Далее пример:

```
s = '1-800-271-8281'
print(s.split('-'))
```

```
['1',  '800',  '271',  '8281']
```

8.3 join

Метод *join* в некотором смысле противоположен *split*. Это строковый метод, который принимает список строк и соединяет их вместе в одну строку. Здесь примеры, использующие список L = ['A', 'B', 'C']

Операция	Результат
' '.join(L)	A B C
''.join(L)	ABC
', '.join(L)	A, B, C
'***'.join(L)	A *** B *** C

Пример Напишите программу, которая создает анаграмму данного слова. Анаграмма слова использует те же самые буквы, как и слово, но в различном порядке. Например, двумя анаграммами слова *there* являются *three* и *ether*. Не переживайте по поводу, реальное ли слово анаграмма или нет.

Это созвучно с тем, что мы можем применить для этого *shuffle*, но *shuffle* работает только со списками. Что нам необходимо сделать, это преобразовать нашу строку в список, применить к нему *shuffle*, а потом конвертировать список обратно в строку. Чтобы превратить строку *s* в список, мы можем использовать **list** (s). (Смотри раздел 10.1). Чтобы перевести список обратно в строку, мы будем использовать *join*.

```
from random import shuffle
word = input ('Введите слово: ')

letter_list = list(word)
shuffle(letter_list)
```

```
anagram = ''.join(letter_list)

print(anagram)
```

8.4 Генераторы списков

Генераторы списков, мощное средство создания списков. Вот простой пример:

```
L = [ i  for  i  in  range(5)]
```

Это создает список [0, 1, 2, 3, 4]. Заметим, что синтаксис генератора списка есть что-то напоминающее запись множества в математике. Здесь больше пары примеров генераторов списков. Для этих примеров принимаем следующее:

```
string = 'Привет'
L = [ 1, 14, 5, 9, 12 ]
M = [ 'один', 'два', 'три', 'четыре', 'пять', 'шесть']
```

Генератор списка	Полученный список
[0 for i in range(10)]	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[i ** 2 for i in range(1,8)]	[1, 4, 9, 16, 25, 36, 49]
[i * 10 for i in L]	[10, 140, 50, 90, 120]
[c * 2 for c in string]	['ПП', 'рр', 'ии', 'вв', 'ее', 'тт']
[m[0] for m in M]	['о', 'д', 'т', 'ч', 'п', 'ш']
[i for i in L if i<10]	[1, 5, 9]
[m[0] for m in M if len(m) == 3]	['д', 'т']

Как мы видим в последних двух примерах, мы можем добавлять if в генератор списка. Сравни последний пример с длинным кодом списка:

```
L = [ ]
for  m  in  M:
    if len(m) == 3:
        L.append(m)
```

Множественные for Мы можем использовать больше, чем один for в генераторе списка:

```
L = [ [ i, j ] for i in range(2) for j in range(2)]
```

```
[ [ 0,  0 ], [ 0, 1 ], [ 1, 0 ], [ 1, 1 ] ]
```

Это эквивалентно следующему коду:

```
L = [ ]
for i in range(2):
    for j in range(2):
        L.append( [ i,  j ] )
```

Далее другой пример:

```
L = [ [ i, j ] for i in range(4) for j in range(i) ]
```

```
[ [ 1,  0 ], [ 2, 0 ], [ 2, 1 ],  [ 3, 0 ], [ 3,  1 ], [ 3,  2 ] ]
```

8.5 Использование генераторов списков

Чтобы далее продемонстрировать мощь генераторов списков, мы сделаем первые четыре примера раздела 7.6 по одной строке за штуку используя генераторы списков.

Пример 1 Напишите программу, которая генерирует список L из случайных чисел от 1 до 100.

```
L = [ [ randint(1 , 100 ) ] for  i  in   range(50): ]
```

Пример 2 Замените каждый элемент в списке L его квадратами.

```
L = [ i ** 2   for  i  in   L ]
```

Пример 3 Подсчитать сколько элементов в списке L больше чем 50.

```
n = len( [ i   for  i  in   L   if   i > 50                                    ])
```

Пример 4 Дан список L, который содержит числа от 1 до 100, создайте новый список, чей первый элемент представляет количество единичек в списке L, чей второй элемент представляет количество двоек в L и так далее.

```
frequencies = [ L.count( i )   for  i  in   range(1 , 100) ]
```

Другой пример Метод *join* может быть часто применяться с генераторами списков для быстрого создания строки. Здесь мы создаем строку, которая содержит случайный набор 1000 букв.

```
from   random  import   choice
alphabet = 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя'
s = ''.join([ choice (alphabet)   for   i   in   range(1000)])
```

Еще один пример Предположим, у нас имеется список, чьи элементы списки размером 2, как ниже:

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

Если мы захотим изменить порядок значений в списках, мы можем использовать следующий генератор списка:

```
M = [ [ y, x ]   for  x, y  in  L]
```

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

Заметка Вы можете определенно обходиться без использования генераторов списков, но как только Вы освоите их, Вы обнаружите, что они быстры в написании и легко читаются, чем длинный способ создания списков.

8.6 Двухмерные списки

Существует ряд общих вещей, которые могут быть представлены двухмерными списками, как доска крестиков - ноликов или пикселей на компьютерном экране. В Python, одним из способов создания двухмерного списка является создание списка чьи элементы сами списки. Ниже пример:

```
L = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9] ]
```

Индексация Мы используем два индекса, чтобы получить доступ к отдельным элементам. Чтобы взять значение в ряду *r*, колонке *c* используйте следующее:

```
L [ r ] [ c ]
```

Вывод двухмерного списка Чтобы напечатать двухмерный список, Вы можете использовать вложенные циклы for. Следующий пример выведет 10 x 5 список:

```
for i in range(10):
    for c in range(5):
        print (L [ r ] [ c ],  end = "  ")
    print()
```

Другой опцией является использование функции *pprint* из модуля *pprint*. Эта функция применяется для структурированного вывода его аргументов. Здесь пример для вывода списка L:

```
from pprint import pprint
pprint ( L )
```

Функция *pprint* может быть использована для красивого вывода обычных списков и других объектов в Python.

Работа с двухмерными списками Вложенные циклы for, подобные тем, которые применяются для вывода двухмерных списков, могут быть также использованы для создания элементов в двухмерном списке. Здесь пример, который подсчитывает, как много четных значений в 10 x 5 списке.

```
count = 0
for r in range(10):
    for c in range(5):
        if L[ r ][ c ] %2 == 0:
            count = count + 1
```

Это также может быть сделано с генератором списка:

```
count = sum([ 1 for r in range(10) for c in range(5) if L[r][c] %2 == 0])
```

Создание больших двухмерных списков Чтобы создать большой список, мы можем использовать генератор списка как внизу:

```
L = [ [ 0 ] * 50 for i in range(100)]
```

Это создает список нулей со 100 рядами и 50 колонками.

Выбор рядов и колонок Чтобы получить ряд *r* списка *L* (начиная с ряда *r = 0*), используйте

следующее:

```
L = [ r ]
```

Чтобы получить строку c списка L (начиная с колонки $c = 0$), используйте генератор списка:

```
[ L[r][c]  for  i  in  range(len(L)) ]
```

Выравнивание списков Чтобы выравнить двухмерный список, то есть возвратить одномерный список его элементов, используйте следующее:

```
[ j for  row  in  L  for j  in  row ]
```

Например, предположим, у нас имеется следующий список:

```
L = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9] ]
```

Выровненный список будет:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

Многомерные Создание и использование трехмерных и выше списков подобно. Здесь мы создаем 5 x 5 x 5 список:

```
L = [ [ [0] * 5  for  i  in  range(5) ] for  j  in  range(5) ]
```

В списке чьи элементы являются списками списков. Первое значение в списке есть

```
L = [ 0 ] [ 0 ] [ 0 ]
```

8.7 Задания

1. Напишите программу, которая просит пользователя ввести текст, а потом подсчитать сколько артиклей имеется в тексте. Артикли это слова 'a', 'an' и 'the'.

2. Напишите программу, которая позволяет пользователю ввести пять чисел(читать как строки). Сделайте строку, которая содержит числа пользователя разделенных знаком плюс. Например, если пользователь вводит 2, 5, 11, 33, и 55, тогда строка должна быть '2 + 5 + 11 + 33 + 55'.

3.(а) Попросите пользователя ввести предложение и выведите третье слово предложения.

(б) Попросите пользователя ввести предложение и выведите каждое третье слово предложения.

4.(а) Напишите программу, которая просит пользователя ввести, а потом случайно переставить слова в предложении. Не переживайте о правильности знаков препинания и заглавных буквах.

(б) Решите задачу выше, но сейчас обеспечьте, чтобы предложение начиналось с заглавной, а первое исходное слово начиналось с прописной, если оно не в начале, и что точка стоит в нужном месте.

5. Напишите простую программу 'Цитата дня'. Программа должна содержать список цитат и когда пользователь запускает программу, случайно выбранная цитата должна быть напечатана.

6. Напишите простую программу для розыгрыша лотереи. Розыгрыш должен состоят из шести различных номеров от 1 до 48.

7. Напишите программу, которая оценивает среднее число выпадений, которые она принимает, прежде чем выпадут номера пользователя в лотерее, которая состоит из правильного выбора шести различных чисел в диапазоне от 1 до 10. Чтобы сделать это, запустите цикл 1000 раз, который случайно сгенерирует набор чисел пользователя и имитирует выпадения, пока числа пользователя не выпадут. Найдите среднее число выпадений необходимых при выполнении цикла 1000 раз.

8. Напишите программу, которая имитирует вытаскивание имен из шляпы. В этом розыгрыше, число значений из шляпы, которые участник получает может меняться. Позвольте пользователю ввести список имен и список, как много значений каждый участник имеет в розыгрыше и выведите, кто победил в розыгрыше.

9. Напишите простую игру викторину, у которой есть список десяти вопросов и список ответов к этим вопросам. Игра должна давать игроку ответить на четыре случайно выбранных вопросов. Она должна спрашивать вопрос за вопросом и говорить игроку правильно или неправильно он ответил на вопрос В конце она должна вывести, на сколько из четырех он ответил правильно.

10. Напишите программу цензор. Позвольте пользователю ввести текст и Ваша программа должна вывести текст со звездочками закрывающими слова-ругательства. Число звездочек должно соответствовать длине этих слов. Для целей этой программы, просто используйте 'плохие'слова-*darn, dang, freakin, heck* и *shoot*. Пример вывода ниже:

```
Enter some text: Oh shoot, I thought I had the dang problem
figured out. Darn it. Oh well, it was a heck of a freakin try.


Oh *****, I thought I had the **** problem figured out.
**** it. Oh well, it was a **** of a ***** try.
```

11. Раздел 8.3 описывает как использовать метод *shuffle* для создания случайной анаграммы строки. Используйте метод *choice* для создания случайной анаграммы строки.

12. Напишите программу, которая принимает строку от пользователя, содержащей возможный телефонный номер. Программа должна напечатать *Верно*, если она решит, что телефонный номер реальный, и *Неверно* иначе. Телефонный номер считается верным при условии, что он написан в форме *абв-где-ёжзи* или *1-абв-где-ёжзи*. Тире должно быть включено, телефонный номер должен содержать только числа и тире, и число цифр в каждой группе должно быть правильным. Проверьте Вашу программу с выводом показанным ниже.

```
Введите телефонный номер:  1-301-447-5820
Верно
Введите телефонный номер:   301-447-5820
Верно
Введите телефонный номер:   301-4477-5820
Неверно
Введите телефонный номер:   3X1-447-5820
Неверно
Введите телефонный номер:   3014475820
Неверно
```

13. Пускай L будет списком строк. Напишите генераторы списков, которые создают новые списки

из L, для каждого из следующего.

(а). Список, который состоит из строк s с удаленными первыми символами.

(б). Список длин строк s.

(в). Список, состоящий из строк s, длиной не менее трех символов.

14. Используйте генератор списка для создания списка, который состоит из всех палиндромических чисел в диапазоне от 100 до 1000.

15. Используйте генератор списка для создания списка, как внизу, который состоит из единиц разделенных возврастающим количеством нулей. Последние две единицы в списке должны быть разделены десятью нулями.

$$[\ 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1 \ldots\ ]$$

16. Пускай L = $[\ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47\ ]$. Используйте генератор списка для создания списка разниц между соответствующими значениями в L. Затем найдите max размер разницы и процент разницы, которая имеет размер 2.

17. Напишите программу, которая находит среднее значение всех элементов в 4 x 4 списке целых чисел.

18. Напишите программу, которая создает 10 x 10 список случайных целых чисел от 1 до 100. Затем сделайте следующее:

(а) Напечатайте список.

(б) Найдите самое большое значение в третьем ряду.

(в) Найдите самое маленькое значение в шестой колонке.

19. Напишите программу, которая создает 10 x 10 список, чьи элементы чередуются между 1 и 2 в шахматном порядке, начиная с 1 в верхнем левом углу.

20. Напишите программу, которая проверяет, является ли список 4 x 4 магическим квадратом. В магическом квадрате, каждый ряд, колонка и две диагонали в сумме дают то же самое значение.

21. Напишите программу, которая просит пользователя ввести длину. Программа должна спросить его, в какой размерности длина и в какую размерность преобразовать. Возможные единицы измерения - дюймы, ярды, мили, миллиметры, сантиметры, метры и километры. Хотя это может быть сделано с 25 инструкциями *if*, короче и проще добавлять, если Вы используете двухмерный список преобразований, поэтому используйте списки для этой задачи.

22. Следующее полезно, как часть программы для игры в морской бой. Предположим, у нас есть 5 x 5 список, который состоит из нулей и единиц. Попросите пользователя ввести строку и столбец. Если значение в списке в той строке и столбце равно единице, программа должна вывести *Попал*, и иначе *Мимо*.

23. Это задание полезно в создании игры 'Память'. Случайным образом сгенерируйте 6 x 6 список различных символов, таких, чтобы имелось точно два каждого символа. Пример показан ниже.

```
@  5  #  A  A  !
5  0  b  @  $  z
$  N  x  !  N  z
0  -  +  #  b  :
-  :  +  c  c  x
```

24. Следующее полезно в реализации компьютерных движков в ряд различных игр. Напишите программу, которая создает 5 x 5 список, состоящий из нулей и единиц. Ваша программа должна затем выбрать случайное местоположение в списке, которое содержит ноль и изменить его на единицу. Если все элементы приняли значение 1, то программа должна сказать это.[Подсказка: одним из способом сделать это является создание нового списка, чьи элементы, это координаты всех единичек в списке и использовать метод *choice* для случайного выбора одного из них. Используйте двухэлементный список, чтобы представить набор координат.]

25. Существует старая головоломка, которую Вы можете решить с компьютерной программы. Существует только одно, пятизначное число n, которое таково, что каждое из следующих десяти чисел имеет ровно одну общую цифру в той же позиции, что и n. Найдите это число.

$$01265, 12171, 23257, 34548, 45970, 56236, 67324, 78084, 89872, 99414$$

26. Мы обычно ссылаемся на значения двухмерного списка их строкой и колонкой, как внизу слева. Другой способ показан снизу справа.

```
( 0, 0 )  ( 0, 1 )  ( 0, 2 )      0  1  2
( 1, 0 )  ( 1, 1 )  ( 1, 2 )      3  4  5
( 2, 0 )  ( 2, 1 )  ( 2, 2 )      6  7  8
```

(а) Напишите код, который переводит отображение слева в отображение справа. Операторы // и % будут полезны. Убедитесь, что Ваш код работает с массивами любых размеров.

(б) Напишите код, который переводит отображение справа в отображение слева.

Глава 9

While циклы

Мы уже узнали о циклах for, которые позволяют нам повторять вещи определенное число раз. Иногда, однако, нам необходимо повторять что-то, но мы не знаем заранее точно, сколько раз то должно повториться. Например, игра 'крестики- нолики' продолжается, пока кто-то не выиграет или нельзя сделать ход, таким образом число ходов будет изменяться от игры к игре. Это положение, которое вызвало бы цикл while.

9.1 Примеры

Пример 1 Давайте вернемся к первой программе, которую мы написали в разделе 1.3, температурный конвертер. Одна раздражающая вещь в нем, то что пользователь не может перезапустить программу для каждого нового значения. Цикл while позволит пользователю неоднократно вводить температуры. Простой способ для пользователя указать, что они выполнены, - это попросить пользователя ввести несуществующую температуру, как -1000(которая ниже абсолютного 0). Это сделано ниже.

```
temp = 0
while  temp != -1000:
    temp = eval (input('Введите температуру в Цельсиях (-1000 для выхода):
'))
    print('Температура в Фаренгейтах равна', 9/5 * temp+32)
```

Сначала посмотрите на инструкцию while. Она говорит нам, что мы будем продолжать цикл, то есть получать и конвертировать температуры, так долго, как введенная температура не будет -1000. Как только -1000 введется, цикл while остановится. Отслеживая, программа сначала сравнивает *temp* с -1000. Если *temp* не равняется -1000, тогда программа спрашивает температуру и конвертирует её. Затем программа начинает цикл снова и вновь сравнивает *temp* с -1000. Если *temp* не равняется -1000, программа спросит другую температуру, конвертирует её, и потом начинает цикл снова и делает другое сравнение. Она продолжает этот процесс пока пользователь не введет -1000.

Нам нужна строка *temp = 0* в начале, так как без неё, мы бы получили ошибку имени. Программа начала бы выполнять инструкцию while, пытаясь увидеть не равняется ли *temp* -1000 и получит проблему, потому что *temp* еще не существует. Чтобы справиться с этим, мы просто объявляем *temp* равную нулю. Не существует ничего особого здесь о значении 0. Мы могли установить её любой, за исключением -1000.(Установка значения -1000 привело бы изначально к неверной установке условия цикла while и цикл никогда бы не запустился.)

Обратите внимание, что естественно думать о цикле while, как о продолжающемся цикле пока

пользователь не введет -1000. Однако, когда мы строим условие, вместо того, чтобы думать когда цикл остановить, нам нужно думать с точки зрения того, что должно быть верно, чтобы продолжать работать.

Цикл **while** сильно похож на инструкцию **if**. Различие в том, что инструкция с отступом в **if** блоке будет выполняться один раз, тогда как инструкция с отступом в цикле **while** выполняется периодически.

Пример 2 В предыдущей программе имеется проблема в том, что когда пользователь вводит -1000, чтобы выйти, программа все равно преобразует значение -1000 и не дает никаких сообщений, чтобы показать, что программа закончилась. Улучшенный вариант программы показан ниже.

```
temp = 0
while  temp != -1000:
      temp = eval (input('Введите температуру в Цельсиях (-1000 для выхода):
'))
      if temp != -1000:
          print('Температура в Фаренгейтах равна', 9/5 * temp+32)
      else:
          print('Пока!'
```

Пример 3 Когда в первый раз встретив инструкцию **if** в разделе 4.1, мы написали программу, с которой играли простую игру на отгадывание случайного числа. Проблема в той программе, то что игрок имеет возможность угадать только один раз. Мы можем, по сути, заменить инструкцию **if** в той программе циклом **while**, чтобы создать программу, которая позволяет пользователю отгадывать, пока не угадает.

```
from random import  randint
secret_num = randint( 1, 10 )
temp = 0
while  guess != secret_num :
      temp = eval (input('Отгадай секретный номер: '))
print('Вы наконец угадали его! '
```

Условие *guess != secret_num* говорит, что пока текущая отгадка неправильная, мы будем продолжать цикл. В этом случае, цикл состоит из одной инструкции, инструкции (**input**, и поэтому программа будет продолжать спрашивать пользователя отгадку, до тех пор, пока их отгадка не будет правильной. Нам требуется строка *guess = 0* перед циклом **while**, так чтобы, когда программа в первый раз достигает цикла, *guess* имело бы какое- нибудь значение, чтобы программа применяла для сравнения. Точное значение *guess* не имеет по настоящему значения в этот момент. Мы просто хотим что-то, что гарантированно будет отличаться от *secret_num*. Когда пользователь, наконец, угадает правильный ответ, цикл закончится и управление программой переходит к инструкции **print** после цикла, которая выводит поздравительное сообщение игроку.

Пример 4 Мы можем использовать цикл **while**, чтобы имитировать цикл **for**, как показано ниже. Оба цикла имеют тот же самый эффект.

```
for i range(10):
print(i)
```

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

Вспомните, что цикл **for** начинается с переменной цикла i равной 0 и заканчивается равной 9. Чтобы использовать цикл **while** для имитации цикла **for**, мы должны вручную создать нашу собственную переменную цикла i. Мы начинаем устанавливая её равной нулю. В цикле **while**, у нас имеется та же самая инструкция **print**, как в цикле **for**, но у нас есть другая инструкция, $i = i + 1$, чтобы вручную увеличить переменную цикла, что-то что цикл **for** делает автоматически.

Пример 5 Ниже наш старый друг, который конвертирует Фаренгейты в Цельсии.

```
temp = eval (input('Введите температуру в Цельсиях: '))
print('Температура в Фаренгейтах равна', 9/5 * temp+32)
```

Программа, которая принимает ввод от пользователя, может захотеть проверить, допустимые ли данные ввел пользователь. Самым маленьким возможным значением температуры является абсолютный ноль,$-273,15°$C. Программа ниже учитывает абсолютный ноль:

```
temp = eval (input('Введите температуру в Цельсиях: '))
if temp <-273.15:
    print('Эта температура невозможна.', 9/5 * temp+32)
else:       print('Температура в Фаренгейтах равна', 9/5 * temp+32)
```

Одним из способов улучшить это является, позволить пользователю вводить температуру, пока он не введет правильную. Вы можете поэкспериментировать что-то подобное используя онлайн форму для ввода телефонного номера или номер кредитной карты. Если Вы ввели неправильный номер, то Вас попросят ввести его вновь. В коде ниже, цикл **while** действует очень похоже инструкции *if* в предыдущем примере.

```
temp = eval (input('Введите температуру в Цельсиях: '))
while temp <-273.15:
    temp = eval (input('Невозможно. Введите допустимую температуру: '))
print('Температура в Фаренгейтах равна', 9/5 * temp+32)
```

Заметим, что здесь нам не нужна инструкция **else**, в отличии от инструкции **if**. Условие в цикле **while** гарантирует, что мы перейдем к инструкции **print** сразу, как только пользователь введет допустимую температуру.

Пример 6 Как упоминалось прежде, ценным качеством является способность читать код. Одним из способов сделать это - стать подобно интерпретатору Python и пройти код, строку за строкой. Давайте попытаемся это с кодом ниже.

```
i = 0
while  i<50:
        print(i)
       i = 1 + 2
print('Пока')
```

Переменная *i* в начале устанавливается равной 0. Затем, программа проверяет условие в цикле *while*. Из-за того, что *i* равна 0, что есть меньше чем 50, код с отступом под инструкцией **while** будет выполнен. Этот код выводит текущее значение *i*, а затем выполняется инструкция $i = i + 2$, которая прибавляет 2 к *i*.

Переменная *i* сейчас равняется 2, и программа возвращается назад к инструкции **while**. Она проверяет меньше ли *i* чем 50 и, так как *i* равно 2, что меньше 50, код с отступом должен быть выполнен снова. Таким образом, мы выводим *i* снова, прибавляя к нему 2, и затем цикл возвращается назад, чтобы проверить условие цикла **while** снова. Мы продолжим это пока наконец *i* не станет равной 50.

К этому моменту, условие **while** наконец не будет выполняться и программа перепрыгнет вниз к первой инструкции после **while**, которое выводит *Пока!*. Конечным результатом программы являются числа $0, 2, 4, \ldots, 48$, заканчивающиеся сообщением *Пока!*.

9.2 Бесконечные циклы

Работая с циклами **while**, рано или поздно, Вы случайно пошлете Python в бесконечный цикл. Вот пример:

```
i = 0
while i<10:
      print( i )
```

В этой программе, значение *i* никогда не измениться и таким образом, условие $i < 10$ всегда верно. Python будет постоянно выводить нули. Чтобы остановить программу, которая попала в бесконечный цикл, используйте *Restart Shell* в меню *Shell*. Вы можете применить это, чтобы остановить программу, прежде чем она закончит выполняться.

Иногда бесконечный цикл - это то, что вам нужно. Простой способ создать его показан ниже:

```
while True:
        # инструкции, которые повторяются, идут здесь
```

Значение **True** называется булевым значением и обсуждается далее в разделе 10.2

9.3 break оператор

Инструкция **break** может быть использована для выхода из циклов **while** и **for**, прежде чем цикл завершится.

Пример 1 Вот программа, которая позволяет пользователю вводить до 10 чисел. Он может остановить её раньше вводом отрицательного числа.

```
for  i in range(10):
        num = eval(input('Введите число: '))
        if num < 0:
            break
```

Это также может быть достигнуто с циклом while.

```
i = 0
num = 1
while  i < 10 and  num > 0:
        num = eval(input('Введите число: ')
```

Любой метод хорош. Во многих случаях инструкция break может помочь сделать Ваш код легче для понимания и менее громоздким.

Пример 2 Ранее в главе, мы использовали цикл while, чтобы позволить пользователю периодически вводить температуры для конвертирования. Здесь, более или менее, оригинальная версия, с левой стороны, по сравнению с другим подходом использующим инструкцию break.

```
temp = 0
while temp != - 1000:
    temp = eval( input(': ')):
        if temp == -1000:
                print(9/5*temp+32)
        else:
                print('Пока')
```

```
while True:
        temp = eval( input(': ')):
        if temp == -1000:
            print('Пока')
                break
        print(9/5*temp+32)
```

9.4 else оператор

Существует необязательный else, который Вы можете использовать с инструкцией break. Код с отступом под else выполняется только, если цикл заканчивается без выполнения break.

Пример 1 Это простой пример на основе Примера 1 предыдущего раздела.

```
for  i in range(10):
        num = eval(input('Введите число: '))
        if num<0:
                print('Остановился рано')
                break
else:
        print('Пользователь ввел все десять значений')
```

Программа позволяет пользователю ввести до 10 чисел. Если он вводит отрицательное, тогда программа выводит *Остановился рано* и больше не просит чисел. Если пользователь не вводит

отрицательных, тогда программа выведет *Пользователь ввел все десять значений.*

Пример 2 Здесь два способа проверить является ли целое число *num* простым. Простое число - это число, чьи делители 1 и само число. Решение слева использует цикл **while**, в то время как решение справа цикл **for\while**:

```
i = 2
while i < num and num%i !==0:
    i = i + 1
if i == num:
    print('Простое')
else:
    print('Не простое')
```

```
for i in range(2, num):
    if num% ==0:
        print('Не простое')
        break
else:
    print('Простое')
```

Смыслом обоих решений, является сканирование всех целых чисел от 2 до *num*-1, и если любое из них есть делитель, тогда мы знаем, *num* не *простое* число. Чтобы узнать является ли значение *i* делителем *num*, мы просто должны проверить равно ли *num*% 0.

Идея версии с циклом **while** заключается в том, что мы продолжаем цикл пока не найдем делитель. Если мы пройдем весь цикл, не найдя делителя, то i будет равно *num*, и в этом случае число должно быть *простым*.

Идеей версии **for\break** является то, что мы проходим цикл со всеми возможными делителями, и как только мы находим один, мы знаем, что число не простое и мы выводим *Не простое*, и останавливаем цикл. Если мы проходим весь цикл без прерывания, тогда мы не нашли делитель. В этом случае блок **else** будет выполнен и выведет, что число простое.

9.5 Игра 'угадайка', улучшенная

Стоит пройти шаг за шагом как разрабатывать программу. Мы изменим программу для отгадывания из раздела 9.1 делая следующее:

- Игрок получает только пять ходов.
- Программа говорит игроку после каждой попытки, если число больше или меньше.
- Программа выводит соответствующие сообщения о том, когда игрок выигрывает или проигрывает.

Ниже - это то, как мы хотим чтобы программа выглядела:

```
Введите свою отгадку (1 - 100): 50
МЕНЬШЕ. Осталось попыток 4.

Введите свою отгадку (1 - 100): 25
МЕНЬШЕ. Осталось попыток 3.

Введите свою отгадку (1 - 100): 12
МЕНЬШЕ. Осталось попыток 2.

Введите свою отгадку (1 - 100): 6
БОЛЬШЕ.Осталось попыток 1.

Введите свою отгадку (1 - 100): 9
МЕНЬШЕ. Осталось попыток 0.

Вы Введите свою отгадку (1 - 100): 50
проиграли. Правильное число равно 8
```

Сначала подумаем о том, что нам нужно в программе:

- Нам нужны случайные числа, поэтому будет инструкция **import** в начале программы и функция *randint* где-нибудь еще.
- Чтобы позволить пользователю угадывать до тех пор, пока он либо не угадает правильно, либо не закончатся ходы, одним из решений является использование цикла **while** с условием, которое учитывает обе эти возможности.
- Будет инструкция **input**, чтобы принять от пользователя отгадку. Поскольку это то, что периодически делается, инструкция будет идти в цикле.
- Там будет инструкция **if**, которая учитывает уровень больше\меньше.Так как сравнение будет делаться периодически и будет зависеть от отгадки пользователя, она будет идти в цикле после инструкции **input**.
- Будет переменная счета, которая отслеживает как много ходов игрок сделал. Каждый раз, когда пользователь вводит отгадку, счет будет увеличиваться на один, поэтому эта инструкция будет также идти внутри цикла.

Затем начните программировать эти вещи, которые легко сделать:

```
from random  import randint

secret_number = randint(1, 100)
num_guesses = 0

while # здесь идет условие
    guess = eval(input( 'Введите Вашу отгадку (1 - 100):'))
    num_guesses = num_guesses + 1
    # больше\меньше инструкция if идет здесь
```

Для цикла **while**, мы хотим продолжать цикл пока пользователь не угадает секретное число и пока игрок не использует все попытки отгадать:

```
while guess != secret_num  and  num_guesses <= 4:
```

Выше\ниже инструкция **if** может быть сделана как ниже:

```
if guess < secret_num:
    print('БОЛЬШЕ.', 5 - num_guesses,'осталось попыток.\n' )
elif guess > secret_num:
    print('МЕНЬШЕ.', 5 - num_guesses,'осталось попыток.\n' )
else:
    print('Вы отгадали!')
```

Наконец, было бы прекрасно вывести сообщение для игрока, если закончились попытки отгадать. Когда попытки закончаться, цикл **while** остановится и управление программой перейдет к коду, что идет за циклом. К этому моменту, мы можем вывести сообщение, но мы хотим сделать это только в случае, если причина остановки цикла в том, что у игрока закончились попытки, а не в том, что он угадал правильно. Мы можем достигнуть этого с инструкцией **if** после цикла. Это показано ниже, вместе с остальной частью законченной программы.

```
from random import randint
secret_number = randint(1, 100)
num_guesses = 0
guess = 0

while guess != secret_num and num_guesses <= 4:
    guess = eval(input( 'Введите Вашу отгадку (1 - 100):'))
    num_guesses = num_guesses + 1
    if guess < secret_num:
            print('БОЛЬШЕ.', 5 - num_guesses,'осталось попыток.\n' )
    elif guess > secret_num:
            print('МЕНЬШЕ.', 5 - num_guesses,'осталось попыток.\n' )
    else:
            print('Вы отгадали!')
if num_guesses == 5 and guess != secret_num:
    print('Вы проиграли. Правильное число',secret_num)
```

Вот альтернативное решение используя цикл **for****break**:

```
from random import randint
secret_number = randint(1, 100)

for num_guesses in range(5):
    guess = eval(input( 'Введите Вашу отгадку (1 - 100):'))
    if guess < secret_num:
        print('БОЛЬШЕ.', 5 - num_guesses,'осталось отгадок.\n' )
    elif guess > secret_num:
        print('МЕНЬШЕ.', 5 - num_guesses,'осталось отгадок.\n' )
    else:
        print('Вы отгадали!')
        break:
else:
    print('Вы проиграли. Правильное число',secret_num)
```

9.6 Задания

1. Код ниже выводит числа от 1 до 50. Перепишите код используя цикл **while**, чтобы достичь то же самого результата.

```
for i in range(1, 51):
    print( i )
```

2.(а) Напишите программу, которая использует цикл **while** (не цикл **for**), чтобы прочитать строку и вывести символы строки один за одним на отдельных строках.

(б) Измените программу выше, чтобы вывести каждый второй символ строки.

3. Хорошая программа будет проверять, что данные, которые вводят её пользователи правильные. Напишите программу, которая просит пользователя ввести вес и конвертировать его из килограммов в фунты. Всякий раз, когда пользователь вводит вес меньше 0, программа должна сказатьть, что ввод неправильный, и затем просить его снова ввести вес.[Подсказка: используйте цикл **while**, не инструкцию **if**].

4. Напишите программу, которая просит пользователя ввести пароль. Если он вводит правильный пароль,то программа должна говорить ему, что он вошел в систему. Иначе, она должна просить его повторно ввести пароль. У пользователя должно быть только пять попыток для ввода пароля, после чего программа должна сказать ему, что он удален из системы.

5. Напишите программу, которая позволяет пользователю вводить любое количество результатов теста. Пользователь указывает, что ввод закончен, введением отрицательного числа. Выведите, сколько результатов являются A (90 или выше). Также выведите среднее значение.

6. Измените больше\меньше программу так, что когда остается только одна попытка, она говорит *1 попытка*, не *1 попыток*.

7. Напомним, что при заданной строке s, $s.index('x')$ возвращает индекс первого x в s и ошибку, если x не имеется.

(а) Напишите программу, которая просит пользователя ввести строку и букву. Используя цикл**while**, программа должна вывести индекс первого вхождения этой буквы и сообщение, если строка не содержит эту букву.

(б) Напишите программу выше используя цикл **for****break** вместо цикла **while**.

8. НОД (наибольший общий делитель) двух чисел - это самое большое число, на которое оба делятся. Например, нод(18, 42) - это число 6, потому что самое большое число, на которое оба 18 и 42 делятся - это 6. Напишите программу, которая просит пользователя ввести два числа и вычисляет их нод.Показанный внизу способ, который вычисляет НОД, называется алгоритмом Евклида.

- Сначала вычислите остаток деления большого числа меньшим числом.
- Затем, замените большое число меньшим, а меньшее число остатком.
- Повторите этот процесс, пока меньшее число не станет равно 0. НОД равняется последнему значению большого числа.

9. 4000-летний способ вычисления квадратного корня из 5 следующий: начнем с первоначального предположения, скажем 1. Затем вычислите

$$\frac{1+\frac{5}{1}}{2} = 3$$

Далее, берем эту 3 и заменяем единички, в предыдущей формуле, тройками. Это дает

$$\frac{3+\frac{5}{3}}{2} \approx 2.33.$$

Следующая замена 3 в предыдущей формуле на 7\3. Это дает

$$\frac{\frac{7}{3}+\frac{5}{\frac{7}{3}}}{2} = \frac{47}{21} \approx 2.24$$

Если Вы продолжаете делать этот процесс вычисления формулы, получая результат и вставляя его назад, значения в итоге будут ближе и ближе к $\sqrt{5}$. Этот метод работает для чисел отличных от 5. Напишите программу, которая просит пользователя ввести число и используя этот метод вычислить квадратный корень числа с точностью 10^{-10}. Вычисление будет корректным с точностью 10^{-10}, когда абсолютное значение разницы между последовательными значениями меньше чем 10^{-10}.

10. Напишите программу, у которой есть список из десяти слов, у некоторых из которых буквы повторяются, а у некоторых нет. Напишите программу, которая выбирает случайные слова из списка не имеющих повторяющихся букв.

11. Напишите программу, которая начинает с 5 x 5 списка нулей и случайным образом изменяет точно десять из этих нулей на единицы.

12. Напишите программу, в которой у Вас имеется список, содержащий семь целых чисел, которые могут быть 0 или 1. Найдите первое ненулевое значение в списке и измените его на 1. Если ненулевых значений не имеется, выведите сообщение об этом.

13. В разделе 4 была задача, в которой Вас просили написать программу, позволяющую пользователю играть 'камень-бумага-ножницы' против компьютера.В ней было только пять раундов. Перепишите программу так, чтобы было лучшие 3 из 5. Первым становится игрок выигравший подряд 3 раза.

14.Напишите программу, чтобы играть в следующую простую игру. Игрок начинает с 100$. На каждом ходу подбрасывается монета и игрок должен отгадать - орел или решка.Он выигрывает 9$ за каждый правильный ответ и теряет 10$ за неправильный. Игра заканчивается, или когда игрок теряет деньги, или выигрывает 200$.

15. Напишите программу, чтобы играть в следующую игру. Имеется список с несколькими названиями стран и программа случайным образом выбирает одно. Затем, игрок должен отгадать буквы в слове, по одной за раз. Перед каждым отгадыванием, в название страны вставляются правильно отгаданные буквы, а остальные представлены тире. Например, если страна *Канада* и игрок правильно угадал *а, д* и *н*, программа бы вывела *-анада*. Программа должна выполняться пока игрок не отгадает все буквы слова или неправильно отгадает пять букв.

16. Напишите текстовую версию игры Memory. Игра должна генерировать 5 x 5 доску (смотри задание из раздела 8). Первоначально программа должна отображать доску, в виде сетки звездочек размером 5 x 5 . Затем пользователь вводит координаты клетки. Программа должна показать сетку с символом в этих координатах, которые сейчас отображаются. Затем пользователь вводит координаты другой клетки. Программа должна сейчас демонстрировать сетку с предыдущим сим-

волом и отображаемым новым символом. Если два символа совпадают, то они должны заменены на постоянные звездочки в этих положениях. Иначе, когда пользователь вводит следующий набор координат, эти символы должны быть заменены звездочками. Игра продолжается таким образом, пока у игрока все не совпадет или не закончатся ходы. Вы можете решить сколько ходов получит игрок.

17. Попросите пользователя ввести числитель и знаменатель дроби, и номер цифры после запятой, которую он хочет знать. Например, если пользователь вводит числитель 1 и знаменатель 7, и хочет знать 4-ю цифру, Ваша программа должна вывести 8, потому что $\frac{1}{7} = 0.142856\ldots$ и 8 является 4 цифрой. Одним из способов сделать это - имитировать длинный процесс деления, о котором Вы могли узнать в начальной школе. Это может быть сделано в пределах пяти строк, используя оператор // в какой-то момент программы.

18. Случайным образом создайте список размером 6 x 6, у которого имеются 12 единиц, размещенных в разных местах списка. Остальные значения должны быть нули.

19. Случайным образом создайте список размером 9 x 9, где значения равны целым числам от 1 до 9, которые не повторяются в ряду или колонках.

Глава 10

Различные темы II

В этой главе мы рассмотрим множество полезных вещей, которые следует знать.

10.1 str, int, float и list

str,**int**,**float** и **list** функции, которые используются для преобразования одного типа данных в другой.

str Довольно часто нам захочется превратить число в строку, чтобы воспользоваться преимуществом строковых методов для разбиения числа на части. Встроенная функция *str* используется для преобразования объектов в строки. Вот некоторые примеры:

Инструкция	Результат
str(37)	'37'
str(3.14)	'3.14'
str([1, 2, 3])	'[1, 2, 3]'

int и float Функция **int** преобразует что-то в целое число. Функция **float** превращает что-то в число с плавающей точкой. Далее примеры:

Инструкция	Результат
int('37')	37
float('3.14')	3.14
int(3.14)	3

Чтобы преобразовать число с плавающей точкой в целое, функция **int** отбрасывает все после десятичной точки.

list Функция *list* принимает что-то, что может быть превращено в список и делает список. Вот два её применения.

```
list(range(5))    [0, 1, 2, 3, 4]
list('abc')       ['a', 'b', 'c']
```

Примеры

Пример 1 Вот пример, который находит все палиндромические числа от 1 до 10000. Палиндро-

мическое число - это число, которое одинаково читается, как слева направо, так и справа налево, подобно 1221 или 64546.

```
for i in range(1, 10001):
    s = str( i )
    if s == s[ : : -1]:
        print( s )
```

Здесь мы применили функцию **str** для перевода целого числа *i* в строку, поэтому мы можем использовать срезы для её реверса.

Пример 2 Здесь пример, говорящий человеку, который родился 1 января 1991 года, сколько ему лет в 2010.

```
birthday = 'January 1, 1991'
year = int(birthday [-4 : ])
print('Вам будет ', 2010 - year, 'лет.')
```

Годом являются последние четыре символа *birthday*. Мы используем **int** для преобразования этих символов в целое число, поэтому мы можем вычислить год.

Пример 3 Напишите программу, которая берет число *num* и суммирует составляющие его цифры. Например, дано число 47, программа должна возвратить 11(которое равно 4 + 7). Давайте начнем с двухзначного примера.

```
digit = str(num)
answer = int(digit[ 0 ]) + int(digit[ 1 ])
```

Идея заключается в том, что мы переводим *num* в строку, поэтому мы можем применить индексацию для получения двух цифр отдельно. Затем, мы превращаем каждое обратно в целое число используя функцию *int*. Вот версия, которая обрабатывает числа с произвольным количеством цифр.

```
digit = str(num)
answer = 0
for i in range(len(digit)):
    answer = answer + int(digit[ i ])
```

Мы можем сделать в одну строку, программу приведенную выше, применяя генератор списка

```
answer = sum([ int(c) for c in str(num)])
```

Пример 4 Чтобы разбить десятичное число, *num*, на его целую и дробную части, мы можем сделать следующее:

```
ipart = int(num)
dpart = num - int(num)
```

Например, если *num* равняется 12.345, тогда *ipart* равна 12, а *dpart* равна 12.345 - 12 = 0.345

Пример 5 Если мы захотим проверить, является ли число простым, мы можем сделать так, проверяя имеет ли оно какие-либо делители, отличные от самого числа и 1. В разделе 9.4 мы видели код для этого и у нас был следующий цикл **for**:

```
for i in range(2, num):
```

Это проверяет делимость на целые числа $2, 3, \ldots,$ *num*-1. Однако оказывается, что на самом деле вам нужно только проверить целые числа от 2 до квадратного корня из числа. Например, чтобы проверить является ли число 111 простым, Вам нужно только проверить делиться ли оно целыми числами от 2 до 10, так как $\sqrt{111} \approx 10.5$. Мы могли также попробовать следующий цикл **for**:

```
for i in range(2, num ** 0.5):
```

Однако, это даст ошибку, потому что *num* ** 0.5 не может быть целым числом, а функции **range** нужны целые числа. Мы можем применить **int**, чтобы исправить это.

```
for i in range(2, int(num ** 0.5) + 1):
```

+1 в конце нужно из-за того, что функция **range** не включает последнее значение.

10.2 Булевы переменные

Булевы переменные в Python - это переменные, которые могут принимать два значения, **True** и **False**. Здесь два примера установки булевых переменных:

```
game_over = True
highlight_text = False
```

Булевы значения могут помочь сделать Ваши программы более читаемыми. Они часто используются, как переменные флага или для индикации опций. Булевы значения часто применяются как условия в инструкциях *if* и циклах *while*:

```
if game_over:
    print('Пока!')
```

Обратите внимание на следующие эквиваленты:

```
if game_over:          ⇔ if game_over == True:
while not game_over:   ⇔ while game_over == False:
```

заметка Условные выражения преобразуются в логические значения, и вы даже можете присвоить их переменным. Например, следующее присваивает **True** к *x*, потому что 6 == 6 преобразуется к **True**.

```
x = (6 == 6)
```

Мы уже видели булевы значения прежде. Строковый метод *isalpha* возвращает **True**, если каждый символ строки является буквой и **False** иначе.

10.3 Сокращения

- **Операторы сокращений** Операции типа *count* = *count* + 1 случаются так часто, что для

них существует сокращенная запись. Вот пара примеров:

Инструкция	Сокращение
count = count + 1	count += 1
total = total - 5	total -= 5
prod = prod * 2	prod *= 2

Существуют также операторы сокращений /=, %=, //=, и **=.

- **Сокращения присвоения**

Взгляните на код ниже.

```
a = 0
b = 0
c = 0
```

Отличное сокращение:

```
a = b = c = 0
```

- **Другое сокращение присвоения**

Скажем, у нас список L с тремя элементами в нем, и мы хотим присоединить эти элементы к именам переменных. Мы могли сделать следующее:

```
x = L[ 0 ]
y = L[ 1 ]
z = L[ 2 ]
```

Вместо, мы можем сделать это:

```
x,  y,  z,  = L
```

Аналогично, мы можем присоединить три переменные сразу, как внизу:

```
x,  y,  z,  = y,  z,  x
```

- **Сокращения с условиями**

Вот некоторые удобные сокращения:

Инструкция	Сокращение
if a == 0 and b == 0 and c = 0	if a == b == c == 0
if 1<a and a<b and b<5	if 1<a<b<5

10.4 Укороченный цикл

Скажем, мы пишем программу, которая ищет списки слов чьи пятые элементы символ 'z'. Мы могли бы попробовать следующее:

```
for w in words:
    if w[4] == 'z':
        print(w)
```

Но с этим, мы иногда будем получать *string index out of range* ошибку. Проблема заключается в том, что некоторые слова в списке могут быть длиной меньше, чем 5 символов. Следующая инструкция if, однако, будет работать:

```
if len(w)>=5 and w[4] == 'z':
```

Может показаться, что мы до сих пор будем получать ошибку, потому что мы все еще проверяем w[4], но ошибки не будет. Смысл в том, почему это работает является *укороченный цикл.*Python начинает проверкой первой части условия, len(w)>=5. Если это условие окажется неверным, тогда целое условие and гарантировано будет неверным и поэтому нет смысла даже рассматривать второе условие. Таким образом, Python не переживает за второе условие. Вы можете полагаться на это действие.

Укороченный цикл также бывает с условиями or. В этом случае, Python проверяет первую часть условия or и если оно верно, тогда все or будет гарантировано верно, и поэтому Python не будет беспокоится о проверке второй части or.

10.5 Продолжение

Иногда, Вы будете писать длинную строку кода, которая была бы лучше читаема, если ее разделить на две строки. Чтобы сделать это, используйте обратный слеш \символ в конце строки показывая, что инструкция продолжается на следующей строке. Далее пример:

```
if 'а' in string or 'б' in string or 'в' in string \
or 'г' in string or 'д' in string:
```

Убедитесь, что после обратного слеша нет пробелов или Вы получите сообщение об ошибке.

Если Вы вводите список, словарь, или аргументы функции, обратный слеш может быть опущен:

```
L = [ 'Джо', 'Боб', 'Сью', 'Джимми', 'Тодд', 'Френк',
'Майк', 'Джон', 'Эми', 'Эдгар', 'Сэм' ]
```

10.6 pass

Инструкция pass ничего не делает. Верьте этому или нет, такой объект действительно имеет несколько применений, которые мы увидим позднее.

10.7 Форматирование строк

Предположим, мы пишем программу, которая вычисляет 25% чаевых от счета $23.60. Когда мы умножим, то получим 5.9, но нам хочется отобразить результат, как $5.90 не $5.9. Вот как это сделать:

```
a = 23.60 * 0.25
print('Чаевые равны {:.2f}'.format(a))
```

Это использует строковый метод format. Здесь другой пример:

```
счет = 23.60
чаевые = 23.60 * 0.25
print('Чаевые: ${:.2f}, Total: ${:.2f}'.format((чаевые, счет + чаевые))
```

Способ работы метода format заключается в том, что мы помещаем пару фигурных скобок в любом месте,где мы хотим отформатировать значение. Аргументами к функции *format* являются значения, которые мы хотим отформатировать, с первым аргументом совпадающим с первым набором скобок, вторым аргументом со вторым набором и так далее. Внутри каждого набора скобок, Вы можете указать форматирующий код, который определяет как соответствующий аргумент будет отформатирован.

Форматирование целых чисел Чтобы отформатировать целые числа используется форматирующий код {:d}. Подставляя число впереди *d* позволяет нам выравнивать вправо целые числа. Вот пример:

```
print('{:3d}'.format(2))
print('{:3d}'.format(25))
print('{:3d}'.format(138))
```

```
  2
 25
138
```

Число 3 в этих примерах говорит, что значению выделено три места. Значение размещено в этих трех местах так далеко вправо, как возможно, а остальные места будут заполнены пробелами. Такого рода вещь полезна для красивого форматирования таблиц.

Чтобы разместить целые числа в центре вместо правого расположения, используйте ^символ, а чтобы расположить влево, применяйте < символ.

```
print('{:^5d}'.format(2))
print('{:^5d}'.format(25))
print('{:^5d}'.format(138))
```

```
  2
 122
13834
```

Каждый из них выделяет пять пробелов для целого числа и центрирует его внутри этих пяти пробелов.

Ставя запятую в форматирующий код, будет форматировать целые числа запятыми. Пример ниже выведет 1,000,000:

```
print('{:,d}'.format(1000000))
```

Форматирование чисел с плавающей точкой Чтобы форматировать число с плавающей точкой применяется форматирующий код {:f}. Чтобы отобразить число с двумя десятичными знаками используйте {:2f}. 2 может быть изменена, чтобы изменить число десятичных знаков.

Вы можете определить выравнивание по правому краю числа с плавающей точкой. Например, {:8.2f} будет выделять восемь мест для своего значения - одно из них для десятичной точки и

два для части значения после десятичной точки. Если значение равно 6.42, тогда нужны только четыре места и оставшиеся места заполняются пробелами, заставляя значение быть выровненным по правому краю.

Символы ^и < центруют и выравнивают по левому краю числа с плавающей точкой.

Форматирование строк Чтобы форматировать строки используется форматирующий код { : s }. Вот пример, который центрирует текст:

```
print( '{  :^10s } ' . format('Hi'))
print( '{  :^10s } ' . format('there!'))
```

```
  Hi
there
```

Чтобы выровнять вправо строку, используйте > символ:

```
print( '{  :>6s } ' . format('Hi '))
print( '{  :>6s } ' . format('there!'))
```

```
    Hi
there!
```

Имеется много больше, что может быть сделано с форматированием.

10.8 Вложенные циклы

Мы можем помещать циклы внутрь других циклов. Цикл внутри другого цикла называется *вложенным*, и Вы можете, больше или меньше, ложить циклов так глубоко, как Вы хотите.

Пример 1 Напечатайте 10 x 10 таблицу умножения.

```
for i in range(1, 11):
    for j in range(1, 11):
        print('{  :3d } ' . format(i * j , end = '  '))
    print()
```

Таблица умножения - двухмерный объект. Чтобы работать с ним, мы применяем два цикла, один для горизонтального, а другой для вертикального направлений. Инструкция **print** выравнивает вправо произведения, делая их вид презентабельным. *end* = '' позволяет нам выводить несколько объектов на каждом ряду. Когда мы закончим выводить строку, мы применяем **print**, чтобы перейти на следующую строку.

Пример 2 Обычной математической задачей является решение системы уравнений. Иногда, Вам захочется найти только целочисленные решения, а это может быть слегка сложно математически. Однако, мы можем написать программу, которая делает полный перебор для поиска решений. Здесь мы находим все целочисленные решения(x, y) в системе 2x + 3y = 4, x - y = 7, где x и y оба между -50 и 50.

```
for x in range(-50, 50):
    for y in range(-50, 50):
        if 2 * x + 3 * y == 4  and  x - y == 7:
            print(x, y)
```

Пример 3 Пифагоровы тройки - это тройка чисел (x, y, z). такие как $x^2+y^2 = z^2$. Например, (3, 4, 5) являются пифагоровами тройками, потому что $3^2+4^2 = 5^2$. Пифагоровы тройки соответствуют треугольникам, чьи стороны равны всем целым числам (как 3-4-5 треугольник). Вот программа, которая находит все пифагоровы тройки (x, y, z), где x, y и z положительные и меньше чем 100.

```
for x in range(1, 100):
    for y in range(1, 100):
        for z in range(1, 100):
            if x ** 2 + y ** 2 == z ** 2:
                print(x, y, z)
```

Если Вы запустите программу, то заметите что имеются избыточные решения. Например, (3, 4, 5) и (4, 3, 5) оба перечислены. Чтобы избавиться от этих чрезмерностей, измените второй цикл, так чтобы он запускался от x до 100. Таким образом , когда x равняется 4, например, первое значение для y, которое будут искать, равно 4, а не 1, и поэтому мы не получим дублирующий список (4, 3, 5). Также измените третий цикл, так чтобы он запускался от y до 100.

Просматривая решения, Вы также можете заметить, что есть много решений, которые кратны другим, как (6, 8, 10) и (9, 12, 15) кратны (3, 4, 5). Следующая программа находит только простые пифагоровы тройки, которые не кратные другим тройкам. Способ, которым она это делает, заключается в том, что каждый раз, когда обнаруживается новая тройка, она проверяет, чтобы убедиться, что x, y и z, все не делятся на то же самое число.

```
for x in range(1, 100):
    for y in range(x, 100):
        for z in range(y, 100):
            if x ** 2 + y ** 2 == z ** 2:
                for i in range(2, x):
                    if x % i ==0 and y % i == 0 and z % i == 0:
                        break
                else:
                    print((x, y, z), end = ' ')
```

Пример 4 В разделе 15.7, мы напишем игру, чтобы играть 'крестики-нолики'. Доска состоит из сетки 3 x 3, и мы будем использовать вложенные циклы для её создания.

Пример 5 Ваш экран компьютера - это сетка пикселей. Чтобы нарисовать изображения на экране, мы часто применяем вложенные циклы for - один цикл для горизонтального, а один для вертикального направлений. Смотри разделы 18.2 и 22.6 для примеров.

Пример 6 Генераторы списков могут содержать вложенные циклы for. Пример ниже возвращает список всех гласных в списке слов.

```
[ char  for  item in  L  for  char  in  item  if  char  in  'а, е, и, о, у, э, ы, я, ё, ю ']
```

10.9 Задания

1. Напишите программу, которая использует **list** и **range** для создания списка [3, 6, 9, . . . , 99].

2. Напишите программу, которая просит пользователя ввести вес в килограммах, форматируя результат до одного десятичного знака.

3. Напишите программу, которая просит пользователя ввести слово. Расставьте все буквы слова в алфавитном порядке и выведите результирующее слово.

4. Напишите программу, которая берет список из десяти цен и десяти товаров, применяет 11% скидку к каждой цене, отображая вывод как ниже, выровненной вправо и красиво отформатированной.

Яблоки	$	2.45
Апельсины	$	18.02
. . .		
Груши	$	120.03

5. Используйте следующие два списка и метод format, чтобы создать список названий карт в формате *значение карты имя масти* (например, 'Два Трефы').

```
suits = [ 'Черви'♡, 'Бубны'♢, 'Трефы'♣, 'Пики'♠ ]
values = [ 'Один', 'Два', 'Три', 'Четыре', 'Пять', 'Шесть', 'Семь',
'Восемь', 'Девять', 'Десять', 'Валет', 'Дама', 'Король', 'Туз', ]
```

6. Напишите программу, которая использует логическую переменную flag для определения, имеют ли два списка какие-либо общие элементы

7. Напишите программу, которая создает список [1, 11, 111, 1111, . . . , 111 . . . 1], где значениями являются постоянно возрастающее число из единиц, с последним значением состоящим из 100 единиц.

8. Напишите программу, которая находит все числа от 1 до 1000, которые делятся на 7 и заканчиваются на 6.

9. Напишите программу, которая определяет сколько чисел от 1 до 10000 содержат цифру 3.

10. Складывая определенные числа с их значениями в обратном порядке, иногда, получаются палиндромические числа. Например, 241 + 142 = 383. Иногда, нам приходиться повторять процесс. Например, 84 + 48 = 132 и 132 + 231 = 363. Напишите программу, которая находит оба двухзначных числа для которых этот процесс должен быть повторен более чем 20 раз, чтобы получить палиндромическое число.

11. Напишите программу, которая находит все пары шестизначных палиндромических чисел, которые отличаются друг от друга на значение не более чем 20. Одной такой парой является 199991 и 200002

12. Число 1961 читается так же и в перевернутом положении и обратном порядке. Выведите все числа от 1 до 100000, которые читаются также.

13. Число 99 имеет свойство, которое заключается в том что, когда мы умножаем его составляющие цифры, и потом складываем их, а затем суммируем все так, что получаем назад 99. То есть, $(9 \star 9) + (9 + 9) = 99$. Напишите программу, которая находит все числа меньше чем 10000 с этим свойством.(Существуют только 9 из них.)

14. Напишите программу, которое находит самое маленькое положительное целое число, которое удовлетворяет следующему свойству: если Вы возьмете цифру с левой стороны числа и переставите его на правую сторону, то число таким образом станет точно в 3,5 раза больше, чем первоначальное число. Например, если мы возьмем 2958 и переставим 2 вправо, мы получим 9852, которое почти в 3,2 раза начального числа.

15. Напишите программу, которая определяет сколько нулей в конце 1000!.

16. Напишите программу, которая переводит десятичную высоту в футах в футы и дюймы. Например, ввод 4.75 футов должен стать 4 фута, 9 дюймов.

17. Напишите программу, которая периодически просит пользователя ввести высоту в формате *фут'дюймы*"(как 5'11"или 6'3"). Пользователь покажет, что он закончил ввод значений, введением *сделано*. Программа должна возвратить результат, сколько было введено 4-футовых, 5-футовых, 6-футовых и 7-футовых значений.

18. Напишите программу, которая периодически просит пользователя ввести футбольный счет в формате *счет побед - счет поражений* (подобно 27-13 или 21-3). Пользователь показывает, что он закончил ввод счетов, введением *сделано*. Затем программа должна вывести самый высокий и самый низкий результат из всех введенных данных.

19. Напишите программу, которая периодически просит пользователя ввести день рождения в формате *месяц/день*(как 12/25 или 2/14) Пользователь показывает, что он закончил ввод дней рождений, введением *сделано*. Программа должна возвратить, количество этих дней рождений в феврале, и количество на 25 число какого-либо, любого месяца.

20. Напишите программу, которая просит пользователя ввести дату в формате *мм/дд/гг* и перевести её в более подробный формат. Например, 02/04/77 должно быть преобразовано в *февраль 4, 1977*.

21. Напишите программу, которая просит пользователя ввести дробь в форме строки, как '1/2' или '8/24'. Программа должна сократить дробь до минимального значения и вывести результат.

22. Напишите программу, чтобы найти все четыре решения к следующей задаче: если карамбола стоит \$5, манго стоит \$3, а три апельсина вместе стоят \$1, сколько карамболы, манго и апельсинов, общим количеством 100 штук, могут быть куплены на \$100?

23. Наличность незнакомой страны имеет монеты номиналом 7 и 11 центов. Напишите программу, которая определяет самую большую стоимость покупки, которая не может оплачена используя эти две монеты.

24. Вот старая загадка, которую Вы можете решить используя перебор, применением компьютерной программы для проверки всех возможностей: в вычислении $43 + 57 = 207$, каждая цифра на единицу отличается от своего правильного значения. Какое правильное вычисление?

25. Напишите программу, которая находит все решения в целых числах в уравнение Пелля $x^2 - 2y^2 = 1$, где x и y находятся в диапазоне от 1 до 100.

26. Напишите программу, которая просит пользователя ввести число и вывести все способы написания числа, как разницу двух полных квадратов, $x^2 - y^2$, где x и y находятся между 1 и 1000. Написание числа, как разницы двух квадратов, приводит к продуманной технике для разложения больших чисел.

27. Напишите программу, которая имитирует все возможные броски четырех кубиков и для каждого сымитированного броска находит суммы пар кубиков. Например, если в броске выпало 5 1 2 4, суммы равны 6, 8, 9, 3, 5 и 6. Для каждой из возможных сумм от 2 до 12 найдите общее число сымитированных бросков в которых сумма появляется, и проценты сымитированных бросков в которых эти суммы появились. Вывод общего числа и процентов, красиво отформатированных, с процентами отформатированными до одного десятичного знака. Чтобы проверить работу, Вы должны найти, сумма 2 появляется в 171 броске, который составляет 13.2% бросков.

28. В магическом квадрате, каждая строка, каждая колонка и обе диагонали в сумме дают то же самое число. Частично заполненный магический квадрат показан ниже. Напишите программу, которая проверяет все возможности заполнения магического квадрата.

```
5 _ _
_ 6 2
3 8 _
```

29.Следующее полезно, как часть программы для игры 'Сапер'. Предположим, у нас имеется список размером 5 x 5, состоящий из нулей и M'к. Напишите программу, которая создает новый 5 x 5 список, содержащий M'ки в тех же местах, только нули заменены числом, составляющим количество M'к находящихся в смежных клетках (смежные по горизонтали, вертикали или диагонали). Пример показан ниже.[Подсказка: укороченный цикл может быть полезен, чтобы избежать ошибок *index-out-of-range*.]

```
0  M  0  M  0        1  M  3  M  1
0  0  M  0  0        1  2  M  2  1
0  0  0  0  0        2  3  2  1  0
M  M  0  0  0        M  M  2  1  1
0  0  0  M  0        2  2  2  M  1
```

30. Треугольник Паскаля показан ниже. Снаружи имеются 1, а каждое другое число - это сумма двух чисел прямо над ним. Напишите программу создающую треугольник Паскаля. Позвольте пользователю определить число рядов. Обеспечьте, чтобы он был красиво отформатирован, как внизу.

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

31. Даны две даты введенные,как строки в форме мм/дд/гггг, где годы в промежутке 1901 и 2099, определите на сколько дней они отстоят друг от друга. Вот информация, которая может быть полезна: високосный год между 1901 и 2099 случается точно каждые четыре года, начиная с 1904. Февраль имеет 28 дней, 29 в високосный год. В ноябре, апреле, июне и сентябре 30 дней. В других месяцах 31 день.

32. Методы Монте Карло могут быть использованы для оценки всех видов объектов, включая подбрасывание монеты и выпадение игральных костей. Как пример, чтобы оценить возможность выпадения пары шестерок у двух костей(кубиков), мы должны использовать случайные целые

числа, чтобы имитировать кости и запустить моделирование тысячи раз, считая какой процент количества раз выпадения шестерок.

(а) Оцените возможность выпадения *Yahtzee* в одном бросании пяти костей. То есть оцените возможность, что при бросании пяти костей, все они выходят с одним и тем же числом.

(б) Оцените возможность выпадения большого стрита в одиночном бросании пяти костей. Большой стрит - это бросок, при котором кости выпадают 1-2-3-4-5 или 2-3-4-5-6 в любом порядке.

(в) Оцените средний самый длинный ряд 'орлов' или 'решек' при подбрасывании монеты 200 раз.

(г) Оцените среднее число подбрасываний монеты перед тем, как выпадет подряд пять 'решек'.

(д) Оцените среднее число подбрасываний монеты, которое требуется для выпадения строки s, где s - строка 'орлов' и 'решек', типа ООРРО.

Глава 11

Словари

Словарь является более общей версией списка. Вот список, который содержит число дней в месяцах года:

```
days = [ 31, 28, 31, 30, 31,30, 31, 31, 30, 31, 30, 31 ]
```

Если нам понадобится число дней в январе, используем *days*[11] или *days*[-1].

Вот словарь дней по месяцам года:

```
days = {  'Январь': 31, 'Февраль':28 , 'Март':31 , 'Апрель':30 ,
'Май':31 ,'Июнь':30 , 'Июль':31 ,'Август':31 ,
'Сентябрь':30 ,'Октябрь':31 ,'Ноябрь':30 ,'Декабрь':31 }
```

Чтобы получить число дней в январе, мы используем *days*['Январь']. Одним из преимуществ использования словарей, здесь является, код, который более читабельный, и мы не должны знать с каким индексом в списке идет месяц. Словари также имеют ряд других применений.

11.1 Основы

Создание словарей Вот простой словарь:

```
d = {  'A':100,  'B':200 }
```

Чтобы объявить словарь, мы заключаем его в фигурные скобки, {} . Каждое значение состоит из пары, разделенных двоеточием. Первая часть пары называется *ключ*, вторая *значение*. Ключ выступает как индекс. Поэтому в первой паре, 'A':100, ключ равен 'A', значение 100, а d['A'] дает 100. Ключи часто бывают строками, но они могут быть целыми числами, числами с плавающей точкой, а также многими другими объектами. Вы можете смешивать разные типы ключей в том же самом словаре, а также различные типы значений.

Изменение словарей Давайте начнем с этого словаря:

```
d = {  'A':100,  'B':200 }
```

- Чтобы изменить d['A'] на 400, делаем

```
d['A'] = 400
```

- Чтобы добавить новое значение в словарь, мы можем просто присвоить его, как ниже:

```
d['C'] = 500
```

Заметим, что такая вещь не работает со списками. Выполнение L[2] = 500 со списком с двумя элементами привело бы к ошибке *index out of range*. Но это работает со словарями.

- Чтобы удалить значение из словаря, используйте оператор del:

```
del d['A']
```

Пустой словарь Пустой словарь - это { } , который является словарным эквивалентом [] для списков или ' ' для строк.

Важная заметка Порядок элементов в словаре не обязательно будет в том порядке в котором их поместили в словарь. Внутренне, Python переставляет объекты в словаре для того, чтобы оптимизировать производительность

11.2 Примеры словарей

Пример 1 Вы можете использовать словарь, как фактический словарь определений:

```
d = {  'собака' : 'собака имеет хвост и говорит гав!',
       'кошка' : 'говорит мяу',
       'мышь' : 'преследуется кошками' }
```

Вот пример словаря в действии:

```
word = input('Введите слово :')
print('Определение является: ', d[word])
```

```
Введите слово : мышь
Определение является : преследуется кошками
```

Пример 2 Следующий словарь полезен в программах, которые используют римские цифры.

```
numerals = {  'I' : 1,   'V' : 5,   'X' : 10,   'L' : 50,   'C' : 100,   'D' : 500,   'M' : 1000 }
```

Пример 3 В игре 'Scrabble', у каждой буквы имеется бальное значение ассоциированное с ним. Мы можем использовать следующий словарь для буквенных значений:

```
points = {  'A' : 1,   'B' : 3,   'C' : 3,   'D' : 2,   'E' : 1,   'F' : 4,   'G' : 2,
            'H' : 4,   'I' : 1,   'J' : 8,   'K':5,   'L' : 1,   'M' : 3,   'N' : 1,
            'O' : 1,   'P' : 3,   'Q' : 10,   'R' : 1,   'S' : 1,   'T' : 1,   'U' : 1,
            'V' : 4,   'W' : 4,   'X' : 8,   'Y' : 4,   'Z' : 10 }
```

Чтобы оценить слово, мы можем сделать следующее:

```
score = sum( [ points[ c ] for   c  in   word ])
```

Или, если Вы предпочитаете длинный способ:

```
total = 0
for   c   in   word:
      total += points[ c ]
```

Пример 4 Словарь обеспечивает хороший способ представления колоды карт:

```
deck = [ { 'значение ' : i, 'масть ', c}
      for c in [ 'пики ', 'трефы ', 'черви ' , 'бубны ' ]
      for i in range(2, 15)]
```

Колода на самом деле представляет список из 52 словарей. Метод *shuffle* может быть использован для перемешивания колоды:

```
shuffle(deck)
```

Первая карта в колоде *deck[0]*. Чтобы получить значение и масть карты, мы обычно используем следующее:

```
deck [0][ 'значение ' ]
deck [0][ 'масть ' ]
```

11.3 Работа со словарями

Копирование словарей Точно как для списков, создание копий словарей немного сложно по причинам, которые мы рассмотрим позднее. Чтобы сделать копию словаря, используйте его метод *copy*. Вот пример:

```
d2 = d.copy()
```

in Оператор in используется, чтобы сказать, если что-то является ключом в словаре. Например, скажем у нас имеется следующий словарь:

```
d = { 'A' : 100, 'B' : 200 }
```

Ссылаясь на ключ, которого нет в словаре, приведет к ошибке. Чтобы предотвратить эту ошибку, мы можем применить оператор in для первоначальной проверки нахождения ключа в словаре, прежде чем пытаться его использовать. Здесь пример:

```
letter = input('Введите букву : ' )
if letter in d:
      print('Значение равно ', d[letter] )
else:
      print('Нет в словаре ' )
```

Мы также можем применить not in, чтобы увидеть есть ли ключ в словаре.

Итерация(Циклы) Итерация словарей подобна итерации списков. Здесь пример, который выводит ключи словаря:

```
for key in d:
      print(key)
```

Вот пример, который выводит значения:

```
for key in d:
      print(d[key])
```

Списки ключей и значений Следующая таблица иллюстрирует способы получения списков ключей и значений из словаря. Он использует словарь $d =$ { 'A' : 1, 'B' : 3 } .

Инструкция	Результат	Результат
list(d)	['A', 'B']	ключи *d*
list(d.values())	[1, 3]	значения *d*
list(d.items())	[('A', 1), ('B', 3)]	(ключ,значение пары) *d*

Пары возвращаемые *d.items* называются *кортежами*. Кортежи сильно похожи на списки. Они рассматриваются в разделе 19.2.

Вот применение *d.items*, чтобы найти все ключи в словаре *d*, которые соотносятся к значению 100:

```
d = { 'A' : 100, 'B' : 200, 'C' : 100 } .
L = [x[0] for x in d.items() if x[1] == 100]
```

```
[ 'A', 'C']
```

dict Функция **dict**, другой метод создания словаря. Одно из применений для нее, в некотором роде противоположно методу *items*:

```
d = dict([('A', 100), ('B', 300)])
```

Это создает словарь { 'A' : 100, 'B' : 300 } . Этот способ построения словаря полезен, если Вашей программе нужно создать словарь во время ее запуска.

Генераторы словарей Генераторы словарей работают аналогично генераторам списков. Следующий простой пример создает словарь из списка слов, где значения являются длинами слов:

```
d = { s : len(s) for s in words }
```

11.4 Подсчет слов

Мы можем использовать словари, чтобы подсчитать, как часто определенные слова появляются в тексте.
В разделе 12.1, мы узнаем, как читать из текстового файла. На данный момент, вот строчка кода, которая читает весь контент из файла содержащий текст *Ромео и Джульетта* Шекспира и хранит его в строке называемой *text*:

```
text = open('romeoandjuliet.txt ' ).read()
```

Чтобы получить отдельные слова, мы применим метод *split* к переводу строки в список с его отдельными словами. Также, из-за того, что некоторые слова могут быть с заглавной буквы, мы преобразуем всю строку к прописным буквам. Мы также должны удалить пунктуацию.

```
from string import punctuation
text = text.lower()
for p in punctuation:
    text = text.replace(p, '' )
words = text.split()
```

Далее идет код словаря, который совершает подсчет. Ключами словаря будут слова из текста, а значениями будет подсчет, сколько раз каждое слово появляется. Мы начинаем с пустого словаря. Далее для каждого слова в списке, если мы видели слово прежде, мы добавляем единицу к его счету, и иначе мы устанавливаем счет для того слова равным единице. Вот код:

```
d = { }
for w in words:
    if w in d:
        d[w] = d[w] + 1
    else:
        d[w] = 1
```

Как только создали словарь, мы можем использовать следующий код для вывода элементов в алфавитном порядке:

```
items = list(d.items())
items.sort()
for i in items:
    print(i)
```

Способ, которым это работает немного сложный. Вспомните, что *d.item()* возвращает список пар(называемых кортежем), которые сильно похожи на списки.Когда мы сортируем список кортежей, сортировка делается по первой записи, которое в этом случае является словом. Поэтому сортировка выполняется по алфавиту.

Если вместо этого, мы захотим, чтобы объекты располагались по распространенности, мы поменяем порядок объектов в кортеже, а потом отсортируем:

```
items = list(d.items())
items = [ (i[1], i[0]) for i in items]
items.sort()
for i in items:
    print(i)
```

Вот, весь код вместе.

```
from string import punctuation
# чтение из файла, удаление заглавных букв и знаков пунктуации, и разделение
на слова
text = open('romeoandjuliet.txt ' ).read()
from string import punctuation
text = text.lower()
for p in punctuation:
    text = text.replace(p, '' )
words = text.split()

# создание словаря распространения по частоте
d = { }
for w in words:
    if w in d:
        d[w] = d[w] + 1
    else:
        d[w] = 1
```

```
# вывод в алфавитном порядке
items = list(d.items())
items.sort()
for i in items:
    print(i)

# вывод в порядке от наименьшего до самого распространенного
items = list(d.items())
items = [ (i[1], i[0]) for i in items]
items.sort()
for i in items:
    print(i)
```

Смотри раздел 24.5 для другого подхода к распространенности слов.

11.5 Задания

1. Напишите программу, которая периодически просит пользователя ввести наименование товаров и цен. Сохраняйте все это в словаре, чьи ключи являются наименованием товаров, а значения ценами. Когда пользователь закончит ввод товаров и цен, обеспечьте ему периодический ввод наименования товара и вывод соответствующей цены или сообщения, если товара нет в словаре.

2. Используя словарь созданный в предыдущей задаче, обеспечьте пользователю ввод суммы в долларах и напечатайте все товары чьи цены меньше чем та сумма.

3. Для этой задачи используйте словарь из начала этой главы, чьи ключи являются названиями месяцев, а значения числами к соответствующим месяцам.

(а) Попросите пользователя ввести название месяца и используйте словарь, чтобы сообщить ему количество дней в месяце.

(б) Выведите все ключи в алфавитном порядке.

(в) Выведите все месяцы в которых 31 день.

(г) Выведите (ключ-значение) пары, отсортированные числом дней в каждом месяце.

(д) Измените программу из части(а) и словарь так, чтобы пользователю не нужно было знать, как пишется точно название месяца. То есть все, что он должен сделать - это написать правильно первые три буквы названия месяца.

4. Напишите программу, которая использует словарь, содержащий десять имен пользователей и паролей. Программа должна просить пользователя ввести его логин и пароль. Если логин отсутствует в словаре, программа должна отобразить, что человек не допустимый пользователь системы. Если логин присутствует в словаре, но пользователь ввел неправильный пароль, программа должна сообщить, что пароль неверный. Если пароль правильный, тогда программа должна сообщить пользователю, что он авторизован в системе.

5. Периодически просите пользователя ввести имя команды и сколько игр команда выиграла и сколько проиграла. Храните эту информацию в словаре, где ключами являются имена команд, а значениями списки в виде *победы, проигрышы*.

(а) Применяя словарь,созданный выше, позвольте пользователю ввести имя команды и вывод выигрышей команды в процентах.

(б) Используя словарь создайте список, чьи значения являются число побед каждой команды.

(в) Применяя словарь, сделайте список из всех этих команд, которые имеют записи о победе.

6. Периодически просите пользователя ввести счет игр в формате *команда1 счет 1 - команда2 счет 2*. Храните эту информацию в словаре, где ключи являются именами команд, а значения списки в форме [*победы, проигрышы*].

7. Сделайте список чисел размером 5 x 5. Затем напишите программу, которая создает словарь, чьи ключи являются числами, а чьи значения, число раз, когда оно появляется. Затем выведите три самых распространенных числа.

8.Используя словарь карт из этого раздела ранее, создайте простую карточную игру, которая раздает двум игрокам по три карты каждому. Игрок с самой большой картой выигрывает. Если случается ничья, тогда сравнивается вторая самая большая карта, если необходимо третья. Если у всех трех карт то же самое значение, то тогда игра в ничью.

9. Используя словарь карт из этого раздела ранее, раздайте три карты. Определите следующее:

(а) Если три карты образуют флэш(все одной масти)

(б) Если имеется три одинаковых(все одного значения)

(в) Если имеется пара, но не три одинаковых

(г) Если три карты образуют стрит(все подряд, как (2,3,4) или (10,Валет, Королева))

10. Используя словарь карт из этого раздела ранее, запустите Монте Карло имитацию, чтобы оценить вероятность выпадения флэша в комбинации из пяти карт. Смотри задание 32, раздела 10 подробнее о моделировании Монте Карло.

11. В разделе 6.10, мы встретили шифр замещения. Этот шифр заменяет каждую букву другой буквой. Например, каждая *a* могла быть заменена *e*, каждая *b* могла быть заменена *a* и так далее. Напишите программу, которая просит пользователя ввести две строки. Затем определите, может ли вторая строка быть закодированной версией первой с помощью шифра замещения. Например, CXYZ не является закодированной версией BOOK, потому что K сопоставляется самой себе. С другой стороны, CXXZ была бы кодировкой BOOK. Эта задача может быть сделана с или без словаря.

12. Ниже ноты, которые употребляются в музыке:

до до# ре ре# ми фа фа# соль соль# ля ля# си

Нотами для *до мажор* являются *до,ми,соль*.

Ноты для аккорда *до мажор* являются *до*, *ми*, *соль*. Математический способ, чтобы получить это состоит в том, что *ми* находится на 4 шага дальше *до*, а *соль* - на 7 шагов дальше *до*. Это работает для любой базы. Например, нотами для *ре мажора* являются *ре*, *фа диез*, *ля*. Мы можем представить шаги мажорного аккорда в виде списка с двумя элементами: [4,7].Соответствующие списки для некоторых других типов аккордов показаны ниже:

минор	[3, 7]	доминантсептааккорд	[4, 7, 10]
увеличенная квинта	[4, 8]	малая септима	[3, 7, 10]
большая кварта	[4, 6]	большой септаккорд	[4, 7, 11]
большая секста	[4, 7, 9]	уменьшённая септима	[3, 6, 10]
малая секста	[3, 7, 9]		

Напишите программу, которая просит пользователя ввести ключ и тип аккорда, и выводит ноты аккорда. Используйте словарь, ключи которого являются (музыкальные)ключи, а значения - списком сдвигов(шагов).

13. Предположим, Вам дан следующий список строк:

```
L = [ 'aabaabac ',  'cabaabca ',  'aaabbcba ',  'aabacbab ',  'acababba ' ]
```

Образцы, как эти встречаются во многих местах, включая последовательность ДНК. У пользователя есть собственная строка, в которой заполнены только некоторые буквы, а остальные обозначены звездочками. Примером является *a**a****. Пользователь хотел бы знать, какая строка в списке подходит к его образцу. В примере только что данным, подходящими строками являются первая и четвертая. Одним из способов решить эту проблему, является создание словаря, ключами которого являются индексы символов в строке пользователя, отличных от звездочки, а значениями которых являются эти символы. Напишите программу реализующую этот способ(или какой-нибудь другой подход), чтобы найти строки, которые подходят к введеной пользователем строке.

14. Словари предоставляют удобный способ хранения упорядоченной информации. Вот пример словаря:

```
d = [ {  'имя' : 'Тодд',  'телефон' : '555-1414',  'email' : 'todd@mail.net'} ,
      { 'имя' : 'Хельга',  'телефон' : '555-1618',  'email' : 'helga@mail.net'} ,
      { 'имя' : 'Принцесс',  'телефон' : '555-3141',  'email' : ''} ,
      { 'имя' : 'ЛДЖ',  'телефон' : '555-2718',  'email' : 'lj@mail.net'} ]
```

Напишите программу, которая читает любой словарь, типа этого и выводит следующее:

(а) Всех пользователей, чьи номера заканчиваются на 8

(б) Всех пользователей, которые не имеют в списке электронного адреса

15. Следующая задача из раздела 6. Попытайтесь решить ее снова, на этот раз применяя словарь, ключи которого являются имена часовых поясов, а значения - это сдвиг от Восточного часового пояса.

Напишите программу, которая переводит время от одного часового пояса к другому. Пользователь вводит время в обычном американском способе, таким как *3:48pm* или *11:26am*. Первый часовой пояс, который пользователь вводит соответствует исходному времени, а второй - желаемому часовому поясу. Возможными часовыми поясами являются - Восточный, Центральный, Горный или Тихоокеанский.

Время: 11:48pm
Начальный пояс: Тихоокеанский
Конечный пояс: Восточный
2:48am

16.(а) Напишите программу, которая переводит римские цифры в обычные. Вот преобразования: M = 1000, D = 500, C = 100, L = 50, V = 5, I = 1. Не забывайте о таких вещах, как IV-4, а XL-40.

(б) Напишите программу, которая переводит обычные числа в римские.

Глава 12

Текстовые файлы

Существует масса интересной информации, которую можно найти в интернете, хранящаяся в тестовых файлах. В этой главе мы узнаем, как работать с данными, которые хранятся в текстовых файлах.

12.1 Чтение из файлов

Предположим, у нас есть текстовой файл называемый *example.txt*, содержание которого показано ниже, и мы хотим прочитать его в Python. Существуют несколько способов сделать это. Мы рассмотрим два из них.

Привет.
Это текстовой файл.
Пока!

1. Первым способом прочитать текстовой файл является использование генератора списка для загрузки файла, строку за строкой в список:

```
lines = [line.strip() for line in open('example.txt' )]
```

Список *lines* сейчас

```
[ 'Привет.', 'Это текстовой файл.', 'Пока!']
```

Строковый метод *strip* удаляет любые символы пробелов из начала и конца строки. Если мы не применим его, то каждая строка содержала бы символ новой строки в конце каждой строки. Обычно, это не то что мы хотим.

Заметка: *strip* удаляет оба из начала и конца строки. Используйте *rstrip*, если Вам нужно сохранить пробел в начале строки.

2. Второй способ чтения текстового файла заключается в загрузке всего файла в строку:

```
s = open('example.txt' ).read()
```

Сейчас строка

```
'Привет. \ nЭто текстовой файл.\ nПока!'
```

Каталоги

Скажем, Ваша программа открывает файл, как ниже:

```
s = open('file.txt' ).read()
```

Файл предполагается быть в том же самом каталоге, как и сама Ваша программа . Если он в другом каталоге, тогда Вам нужно указать то, как внизу:

```
s = open('c:/users/desktop/file.txt' ).read()
```

12.2 Запись в файлы

Существуют несколько способов записи в файлы. Мы взглянем здесь на один способ. Мы будем записывать в файл, называемый *writefile.txt*.

```
f = open('writefile.txt',  'w')
print('Это строка 1.',  file=f)
print('Это строка 2.',  file=f)
f.close()
```

Вначале мы должны открыть файл. Это то, что делает первая строка, с 'w' указывая, что мы хотим иметь возможность записывать в файл. Python создает так называемый файловый объект, представляющий файл, и мы даем этому объекту имя *f*. Это то, что мы используем для ссылки на файл. Чтобы записать файл, мы используем инструкцию **print** с опциональным аргументом file, который указывает на файл для записи. Когда мы закончим записывать, мы должны закрыть файл, чтобы убедиться, что все наши изменения приняты. Будьте осторожны здесь, потому что если *writefile.txt* уже существует, его содержание будет переписано.

12.3 Примеры

Пример 1 Напишите программу, которая читает список температур из файла, называемого *temps.txt*, переводит эти температуры в Фаренгейты, и записывает результаты в файл, называемый *ftemps.txt*

```
file1 = open('ftemps.txt',  'w')
temperatures = [ line.strip()  for  line  in  open('temps.txt']
for  t in  temperatures:
      print(int(t) * 9/5 + 32,  file = file1)
file1.close()
```

Пример 2 В разделе 7.6 мы написали простую головоломку. Вопросы и ответы содержались в списках, четко записанных в программе. Вместо этого, мы можем хранить вопросы и ответы в файлах. Таким образом, если Вы решите изменить вопросы или ответы, Вы только должны изменить их файлы. Более того, если Вы решите дать программу кому-то еще, кто не знает Python, они могут легко изменить свои собственные списки вопросов и ответов. Чтобы сделать это, мы просто изменяем строки, которые создают списки на следующие:

```
questions = [ line.strip()  for  line  in  open('questions.txt']
answers = [ line.strip()  for  line  in  open('answers.txt']
```

Пример 3 Скажем, у Вас есть текстовой файл, который содержит результаты каждой 2009-2010 NCAA баскетбольной игры.(Вы можете найти такой файл на www.kenpom.com). Обычная строка файла выглядит подобно этому:

02/27/2010, Robert Morris, 61, Mount St. Mary's, 63

Ниже программа, которая сканирует файл, чтобы найти самую большую неравномерную игру, игра в которой выигравшая команда имеет самое большое количество побед.

```
lines = [ line.strip()  for  line  in  open('scores.txt']
games = [ line.split(',' )  for  line  in  lines]
print(max([ abs(int(g[2] - int(g[4]))  for  g  in  games)]))
```

Мы используем метод *split*, чтобы разбить каждую строку в списки составляющих ее частей. Результаты идут с индексами 2 и 4. Чтобы найти максимальную разницу, мы можем использовать генератор списка, для создания списка всех побед и применяя **max**, чтобы найти максимум.

Максимум получился 84. К сожалению, способ выше не говорит нам ничего еще об игре. Для того, чтобы сделать то, мы обращаемся к длинному способу нахождения максимума, описанному в разделе 5.5. Это позволит нам хранить информацию об игре, когда мы ищем самое большое количество побед.

```
lines = [ line.strip()  for  line  in  open('scores.txt']
games = [ line.split(',' )  for  line  in  lines]

biggest_diff = 0
for  g  in  games:
    diff = abs(int(g[2] - int(g[4])
     if  diff>biggest_diff:
          diff>biggest_diff = diff
          game_info = g
print(game_info)
```

```
[ '12/03/2009',  'SalemInternational',  '35',  'Marshall',  '119']
```

12.4 Игра слов(Каламбур)

Если Вам нравятся слова, Вы можете весело провести время со списком слов, который представляет собой текстовой файл, где каждая строка содержит другое слово.Быстрый поиск в Интернете выдаст множество различных списков слов, начиная от списков распространенных английских слов и заканчивая списками, содержащими практически все английские слова.

Предполагая, что файлом со списком слов является *wordlist.txt*, мы можем загрузить слова в список используя строку ниже.

```
wordlist = [ line.strip()  for  line  in  open('wordlist.txt']
```

Пример 1 Выведите все слова из трех букв.

```
for  word  in  wordlist:
    if  len(word) == 3:
        print(word)
```

Заметим, что этот и большинство предстоящих примеров может быть сделано с генераторами списков:

```
print([ word  for  in  in  wordlist  if  len(word) == 3])
```

Пример 2 Выведите все слова, которые начинаются с *гн* или *кн*.

```
for  word  in  wordlist:
    if  word[:2] == 'гн'  or  word[:2] == word[:2] == 'кн':
        print(word)
```

Пример 3 Определите, какой процент слов начинается с гласной.

```
count = 0
for  word  in  wordlist:
    if  word[0]  in  'аоеиюэяыёу':
        count = count + 1
        print(100*count/len(wordlist))
```

Пример 4 Выведите все слова, состоящие из 7 букв, которые начинаются на *т* и заканчиваются на *во*. Вещи подобные этому хороши для создания кроссвордов.

```
for  word  in  wordlist:
    if  len(word) == 7  and  word[:2] == 'т '  and word[-2:] == 'во ':
        print(word)
```

Пример 5 Выведите первые десять слов, которые начинаются с *р*.

```
i = 0
while  wordlist [i] [0] != 'р':
    i = i + 1
print(wordlist [i : i + 10])
```

Обратите внимание, что это не очень эффективный способ работы, так как мы должны просматривать большую часть списка. Бинарный поиск был бы более эффективным, но предыдущий метод по прежнему работает почти мгновенно, даже для больших файлов.

Пример 6 Найдите самое длинное слово, которое может быть составлено применяя только буквы *а, б, в, г* или *д*.

```
largest = 0
for word in wordlist:
    for c in word:
        if c not in 'абвгд':
            break
    else:
        if len(word)>largest:
            largest = len(word)
            largest_word = word
print(largest_word)
```

Способ, которым эта программа работает следующий - для каждого слова в списке слов, мы используем цикл for/else(раздел 9.4), чтобы просканировать слово, проверяя каждый символ на соответствие его *а, б, в, г* или *д*. Если буква в слове не является одной из них, тогда мы прерываем цикл и переходим к другому слову. С другой стороны, если мы проходим цикл, тогда мы идем в блок else. В том блоке, мы применяем модификацию техники из раздела 5.5 для нахождения максимума.

12.5 Задания

1. Вам дан файл называемый *class_ scores.txt*, где каждая строка файла содержит одно слово имени пользователя и результат теста разделенных пробелами, как внизу:

GWashington 83
JAdams 86

Напишите код, который сканирует файл, добавляет 5 очков к каждому результату, и выводит имя пользователя и новые результаты в новом файле *scores2.txt*.

2. Вам дан файл называемый *class_ grades.txt*, где каждая строка файла содержит одно слово имени студента и три результата теста разделенных пробелами, как ниже:

GWashington 83 77 54
JAdams 86 69 90

Напишите код, который сканирует файл и определяет сколько студентов прошли все три теста.

3. Вы получили файл, называемый *logfile.txt*, который содержит список времени входа и времени выхода пользователей системы. Типичная строка файла выглядит, как:

Van Rossum, 14:22, 14:37

Каждая строка имеет три значения разделенные запятыми: имя пользователя, время входа, и время выхода. Время дано в 24 часовом формате. Вы можете предположить, что все входы и выходы происходят в течении рабочего дня.

Напишите программу, которая сканирует файл и выводит имена всех пользователей, кто был онлайн хотя бы час.

5. Вам дан файл называемый *students.txt*. Обычная строка в файле выглядит как:

walter melon melon@email.msmary.edu 555-3141

В строке имеется имя, email адрес и телефонный номер, разделенные табуляцией. Напишите программу, которая читает файл строку за строкой, и для каждой строки, устанавливает заглавной первую букву имени и фамилии, и добавляет телефонный код к номеру телефона. Ваша программа должна записать это в новый файл, называемый *students2.txt*. Вот как должна выглядеть первая строка нового файла.

Walter Melon melon@email.msmary.edu 301-555-3141

5. У Вас есть файл *namelist.txt*, который содержит группу имен. Некоторые имена состоят из имени и фамилии разделенных пробелами, как *George Washington*, тогда как другие имеют отчество, как *John Quincy Adams*. Не имеется имен состоящих только из одного слова или больше чем трех слов. Напишите программу, которая просит пользователя ввести инициалы, как *GM* или *JQA*, и вывести все имена, которые подходят к этим инициалам. Обратите внимание, что инициалы подобные *JA* должны подходить к *John Adams* и *John Quincy Adams*.

6. У Вас есть файл *namelist.txt*, который содержит группу имен. Выведите все имена в списке, в которых гласные *a, t, i, o* и *u* появляются по порядку(с возможностью повторения). Первая гласная в имени должна быть *a*, а после первой *u* могут быть другие гласные. Например, *Ace Elvin Coulson*.

7. Вам дан файл называемый *baseball.txt*. Типичная строка файла начинается как ниже:

Ichiro Suzuki SEA 162 680 74 . . . [больше данных]

Каждое значение разделено табуляцией, \t. Первое значение имена игроков, а второе их команда. Далее следуют 16 статистических данных. Домашние раны в седьмой, а украденные базы в одиннадцатой. Распечатайте всех игроков, кто имеет не менее 20 домашних ранов и не менее 20 украденных баз.

8. Для этой задачи используйте файл NCAA баскетбольных результатов, как описано в разделе 12.3

(а) Найдите среднее значение очков, набранных за все игры в файле.

(б) Выберите свою любимую команду и просканируйте файл, чтобы определить сколько игр она выиграла и сколько проиграла.

(в) Найдите команду (команды), которая чаще всего проигрывала на 30 и более очков.

(г) Найдите все команды, которые в среднем набирали за игру не менее 70 очков.

(д) Найдите все команды, которые имели выигрышные записи, но были коллективно обойдены своими соперниками. Команда является коллективно обойденной своими соперниками, если общее число очков, которые команда набрала во всех играх меньше, чем общее число очков их соперников набранных в своих играх против этой команды.

9. Закон Бенфорда устанавливает, что в реальных данных, где значения распределены среди нескольких порядков величин, около 30% значений начинаются с числа 1, в то время как только около 4,6% значений начинаются с числа 9. Это вопреки тому, что мы могли ожидать, а именно, что значения начинающиеся с 1 и 9 были бы примерно равны. Используя файл *expenses.txt*, который содержит количество затрат из расходного счета, определите какой процент начинается

с каждой цифры от 1 до 9. Этот метод используется бухгалтерами для выявления мошенничества.

10. *Игра слов* - Используйте файл *wordlist.txt* для этой задачи. Найдите следующее:

(а) Все слова оканчивающиеся на *ime*

(б) Все слова, у которых вторая, третья и четвертая буквы *ave*

(в) Сколько слов содержит не менее одной из букв *r, s, t, l, n, e*

(г) Процент слов, который содержит не менее одной из букв *r, s, t, l, n, e*

(д) Все слова без гласных.

(е) Все слова, которые состоят из гласных

(ё) Имеются ли слова из десяти или семи букв

(ж) Самое длинное слово в списке

(з) Все палиндромы

(и) Все слова одинаковые, как в прямом так и обратном порядке, типа *rat* и *tar*

(й) То же самое как выше, но только выведите одно слово из каждой пары

(к) Все слова, которые содержат удвоенные буквы следующие друг за другом, типа *aardvark* или *book*, за исключением слов, которые оканчиваются на *lly*

(л) Все слова содержащие *q*, за которой не следует *u*

(м) Все слова, которые содержат *zu* в любом месте слова

(н) Все слова, которые содержат *ab* в нескольких местах, подобно *habitable*

(о) Все слова с четыремя или более гласных подряд

(п) Все слова, которые содержат обе буквы *z* и *w*

(р) Все слова у которых первая буква *a*, третья буква *e* и пятая буква *i*

(с) Все двухбуквенные слова

(т) Все четырехбуквенные слова, которые начинаются и заканчиваются с той же самой буквой

(у) Все слова, которые содержат не менее девять гласных букв

(ф) Все слова, которые содержат каждую из букв *a, b, c, d, e* и *f*, в любом порядке. Могут быть другие буквы в слове. Два примера - *backfield* и *feedback*

(х) Все слова у которых первые четыре и последние четыре буквы те же самые

(ц) Все слова в виде *abcd*dcba*, где * произвольной длины последовательность букв

(ч) Все группы из 5 слов, как *pat, pet, pit, pot, put*, где каждое слово состоит из 3 букв, у всех слов те же самые первая и последняя буквы, а через среднюю букву проходят все 5 гласных

(ш) Слова имеющие самое большое количество *i*

11. Напишите программу, которая помогает со словесными играми. Пользователь вводит слово, а программа используя список слов определяет, является ли слово пользователя реальным или нет.

12. Предположим, мы записываем все слова в списке слов задом наперед, а затем расставляем эти слова в алфавитном порядке. Напишите программу, которая выводит последнее слово в этом измененном списке

13. Напечатайте все комбинации строки 'Python' плюс слово из трех английских букв. Сделайте заглавной первую букву этого слова. Примеры комбинаций - 'PythonCat', 'PythonDog' и 'PythonTag'. Это правильные комбинации, потому что *cat, dog* и *tag* английские слова. С другой стороны, 'PythonQqz', не было бы верной комбинацией, потому что *qqz* не является английским словом. Используйте список слов, чтобы определить, какая комбинация из трех букв является словом.

14. Напишите простую программу для проверки правописания. Пользователь должен ввести строку, а программа должна вывести список всех слов, которые по ее мнению неправильные. Это были бы слова, которые она не может найти в списке слов

15. Кроссворд читер(программа помогающая в играх): когда отгадываешь кроссворд, часто Вам будут попадаться слова, где Вы знаете несколько букв, но не все из них. Вы можете написать компьютерную программу, чтобы помочь себе. Для программы, пользователь должен иметь возможность ввести слово с буквами, которые он знает и звездочками вместо букв, которые не знает. Программа должна вывести список всех слов, которые подходят под то описание. Например, ввод *th***ly* должен возвратить все слова, которые могли подойти, а именно *thickly* и *thirdly*

16. Попросите пользователя ввести несколько букв. Затем найдите все слова, которые могут быть составлены с этими буквами, повторения допускаются.

17. Используя список слов, создайте словарь, у которого ключи являются буквами от *a* до *z*, а значения процент слов, которые используют ту букву.
18. Используя список слов, создайте словарь, у которого ключи являются буквами от *a* до *z*, а значения процент от общего количества букв в списке слов, которые являются этой буквой.

19. Напишите программу, которая просит пользователя ввести слово и находит все меньшие слова, которые могут быть составлены из букв того слова. Количество появлений буквы в меньшем слове не может быть больше чем в слове пользователя.

20.(а) Напишите программу, которая читает файл состоящий из email адресов, каждый на собственной строке. Ваша программа должна вывести строку, состоящую из этих адресов, разделенных точкой с запятой.

(б) Напишите ту же самую программу, как выше, но новая строка должна содержать только те адреса, которые не заканчиваются *@prof.college.edu*

21. Файл *high_ temperatures.txt* содержит средние высокие температуры для каждого дня года в определенном городе. Каждая строка файла содержит дату, написанную в формате месяц/день, далее пробел и средняя высокая температура для той даты. Найдите 30 дневный период, в течение которого наблюдается наибольшее повышение средней высокой температуры.

22. В главе 6, было задание 15 посвященное игре *Mad Libs*. Оно просило Вас придумать историю и убрать некоторые слова из нее. Ваша программа должна попросить пользователя ввести слова и сказать ему какие типы слов ввести. Затем вывести полную историю вместе с вставленными словами. Перепишите Вашу программу из того задания, чтобы читать историю из файла. Чтение

истории из файла позволяет людям, кто не знает как программировать, использовать свои собственные истории в программе без изменения кода.

23.Акроним - это сокращение, которое использует первую букву каждого слова в фразе. Мы видим их везде. Например, NCAA(National Collegiate Athletic Association - Национальная ассоциация студенческого спорта) или NBC(National Broadcasting Company - Национальная широковещательная компания). Напишите программу, куда пользователь вводит акроним и программа случайным образом выбирает слова из списка слов, так чтобы слова соответствовали бы акрониму. Ниже типичный вывод,сгенерированный, когда я запустил программу:

Введите акроним:ABC
['addressed', 'better', 'common']

Введите акроним: BRIAN
['bank', 'regarding', 'intending', 'army', 'naive']

24. Задача посвященная версии игры *Jotto*. Компьютер выбирает случайным образом слово из пяти букв с неповторяющимися буквами. Игрок получает несколько попыток отгадать компьютерное слово. На каждом ходу игрок угадывает слово и ему сообщают количество букв, которые по его предположению совпадают со словом компьютера.

25. У слова *part* есть интересная особенность в том, что если Вы удалите буквы одну за другой, на каждом шаге результатом является реальное слово. Например, *part* $\rightarrow$*pat* $\rightarrow$*pa* $\rightarrow$*a*. Вы можете удалять буквы в любом порядке, и последнее(однобуквенное) слово должно быть реальным словом. Найдите все слова из восьми букв с этим свойством.

26. Напишите программу для подбора слов (cheat - мошенничество) в игре *Scrabble*. Пользователь вводит строку. Программа должна возвратить список всех слов, которые могут быть составлены из этих семи букв.

Глава 13

Функции

Функции полезны для разбиения большой программы на части, чтобы сделать ее более читаемой и легкой в обслуживании. Они также полезны, когда Вы пишите тот же самый код в нескольких разных местах Вашей программы. Вы можете поместить тот код в функцию и вызывать функцию в любое момент когда Вы захотите выполнить тот код. Вы также можете применять функции, чтобы создать свои собственные утилиты, математические функции и так далее.

13.1 Основы

Функции определяют с инструкцией def. Инструкция заканчивается двоеточием, а код, который является частью функции, располагается с отступом ниже инструкции def. Здесь, мы создадим простую функцию, которая просто что-то выводит.

```
def  print hello():
    print('Привет!' )
print_hello()
print('1234567' )
print_hello()
```

```
Привет!
1234567
Привет!
```

Первые две строки определяют функцию. В последних трех строках мы вызываем функцию дважды.

Одним из применений функций является, то что если Вы используете тот же самый код снова и снова в различных частях Вашей программы, то можете сделать свою программу короче и легче понимаемой, располагая код в функции. Например, предположим по какой-то причине Вам нужно вывести прямоугольник из звезд, как внизу в нескольких точках Вашей программы.

```
**************************
*                        *
*                        *
**************************
```

Поместите код в функцию, а затем, когда Вам понадобится прямоугольник просто вызовите функцию вместо набора нескольких строк лишнего кода. Вот функция.

```
def draw_square():
    print('*' * 15 )
    print('*', ' ' * 11, '*')
    print('*', ' ' * 11, '*')
    print('*' * 15 )
```

Одним из преимуществ этого является, то что если Вы решите изменить размер прямоугольника Вам просто нужно доработать код в функции, тогда как, если бы скопировали и разместили код везде где нужно, то Вам пришлось бы изменить его весь.

13.2 Аргументы

Мы можем передавать значения в функции. Вот пример:

```
def print_hello(n):
    print('Привет!' * n)
    print()
print_hello()
print('1234567' )
print_hello(3)
print_hello(5)
times = 2
print_hello(times)
```

```
Привет
Привет Привет Привет
Привет Привет Привет Привет Привет
Привет Привет
```

Когда мы вызываем функцию *print_hello* со значением 3, то значение хранится в переменной *n*. Затем мы можем ссылаться на ту переменную в нашем коде функции.

Мы можем передавать больше одного значения в функцию:

```
def multiple_print(string, n):
    print(string * n)
    print()

multiple_print('Привет', 5)
multiple_print('A', 10)
```

```
ПриветПриветПриветПриветПривет
AAAAAAAAAA
```

13.3 Возвращаемые значения

Мы можем писать функции, которые выполняют вычисления и возвращают результат.

Пример 1 Здесь простая функция, которая переводит температуры из Цельсия в Фаренгейты.

```
def convert(t):
    return t * 9/5 + 32

print(convert(20))
```

```
68
```

Инструкция return используется, чтобы послать результат вычислений функции назад вызывающему.

Обратите внимание, что сама функция ничего не выводит. Вывод осуществляется снаружи функции. Таким образом, мы можем выполнить математические вычисления с результатом, как показано ниже.

```
print(convert(20) + 5)
```

Если бы мы просто вывели результат в функции вместо его возвращения, результат был бы выведен на экран и забыт, и мы ничего бы не смогли сделать с ним.

Пример 2 Как другой пример, Python модуль *math* содержит тригонометрические функции, но они работают только в радианах. Давайте напишем свою собственную функцию синуса, которая работает в градусах.

```
from math import pi, sin
def deg_sin(x):
    return sin(pi * x/180)
```

Пример 3 Функция может возвращать несколько значений как список.

Скажем, мы хотим написать функцию, которая решает систему уравнений $ax + by = e$ и $cx + dy = f$. Получается, что если существует единственное решение, тогда оно задается через $x = (de - bf)/(ad - bc)$ и $y = (af - ce)/(ad - bc)$. Нам нужно, чтобы наша функция возвращала оба x и y решения.

```
def solve(a, b, c, d, e, f):
    x = (d*e - b*f)/(a*d - b*c)
    y = (a*f - c*e)/(a*d - b*c)
    return [x, y]
xsol, ysol = solve(2, 3, 4, 1, 2, 5)
print('Решением является x = ', xsol, 'и y = ', ysol)
```

```
Решением является x = 1.3 и y = -0.2
```

Этот метод использует короткую запись для присваивания спискам, которая была упомянута в разделе 10.3

Пример 4 Инструкция **return** сама может быть использована, чтобы закончить функцию раньше.

```
def multiple_print(string, n, bad_words):
    if string in bad_words:
        return
    print(string * n)
    print()
```

Тот же самый эффект может быть достигнут с инструкцией **if**/**else**, но в некоторых случаях, использование **return** может сделать Ваш код проще и более читаемым

13.4 Аргументы по умолчанию и ключевые

Вы можете определить значения по умолчанию для аргументов. Это делает его не обязательным, и если вызывающий решит не использовать его, то оно принимает значение по умолчанию. Вот пример:

```
def multiple_print(string, n = 1):
    print(string * n)
    print()

multiple_print('Привет', 5)
multiple_print('Привет')
```

```
ПриветПриветПриветПриветПривет
Привет
```

Аргументы по умолчанию нужно указывать в конце определения функции, после всех аргументов не по умолчанию.

Аргументы ключевых слов Соответствующей идей к аргументам по умолчанию является аргументы *ключевых слов*. Скажем, у нас есть следующее определение функции:

```
def fancy_print(text, color, background, style, justify):
```

Каждый раз, когда Вы вызываете эту функцию, то должны помнить правильный порядок аргументов. К счастью, Python позволяет Вам присваивать имена аргументам при вызове функции, как показано ниже:

```
fancy_print(text = 'Hi', color = 'yellow', background = 'black', style = 'bold', justify = 'left' )
fancy_print(text = 'Hi', style = 'bold', justify = 'left', background = 'black', color = 'yellow')
```

Как мы можем видеть, порядок аргументов не имеет значения, когда мы используем ключевые слова аргументов.

Когда определяем функцию, это было бы хорошей идеей давать значения по умолчанию. Например, чаще всего, вызывающий функцию хотел бы левое выравнивание, белый фон и так далее.

Используя эти значения по умолчанию означает, что вызывающий не должен определять каждый отдельный аргумент всякий раз при вызове функции. Здесь пример:

```
def fancy_print(text, color = 'black', background = 'white', style = 'normal',
justify = 'left'):
# здесь идет код функции

fancy_print( 'Hi', style = 'bold')
fancy_print( 'Hi', color = 'yellow', background = 'black')
fancy_print( 'Hi')
```

Заметка Прежде, мы фактически видели значения по умолчанию и ключевые слова аргументов - *sep, end* и аргументы file функции **print**.

13.5 Локальные переменные

Допустим, у нас есть две функции, как внизу, каждая из которых использует переменную *i*:

```
def func1():
    for i in range(10):
        print(i)

def func2():
    i = 100
    func1 ()
    print(i)
```

Проблема, которая могла здесь возникнуть, заключается в том, что когда мы вызываем *func1*, мы могли изменить значение *i* в функции *func2*. В больших программах это был бы кошмар пытаясь не допустить повторения имен переменных в разных функциях, и к счастью мы не должны переживать об этом. Когда переменная определяется внутри функции, она становится *локальной* в той функции, то означает, что она фактически не существует снаружи той функции. Таким образом, каждая функция может определять свои собственные переменные, и не переживать, применяются ли эти имена переменных в других функциях.

Глобальные переменные С другой стороны, иногда Вы действительно захотите сделать, чтобы одна и та же переменная была доступна разнообразным функциям. Такая переменная называется *глобальной*. Вы должны быть осторожны используя глобальные переменные, особенно в больших программах, но несколько глобальных переменных, разумно примененных, хороши в маленьких программах. Вот короткий пример:

print

```
def reset():
    global time_left
    time_left = 0

def print_time():
    print(time_left)

time_left = 30
```

В этой программе у нас есть переменная *time_ left*, к которой, мы хотели бы, имели доступ различные функции. Если функция захочет изменить значение той переменной, то нам нужно сказать функции, что *time_ left* является глобальной переменной. Мы используем инструкцию global в функции, чтобы это сделать. С другой стороны, если мы хотим просто использовать значение глобальной переменной, то нам не нужна инструкция global.

Аргументы Мы заканчиваем главу немного с техническими деталями. Вы можете пропустить этот раздел на данный момент, если не хотите беспокоиться о деталях прямо сейчас. Вот две простые функции:

```
def func1(x):
    x = x + 1 def func2(L):

    L = L + [1]

a = 3
M = [1, 2, 3]
func1(a)
func2(M)
```

Когда мы вызываем *func1* с *a* и *func2* с *L* возникает вопрос: функции изменяют значения *a* и *L*? Ответ может удивить Вас. Значение *a* не изменяется, а значение *L* изменяется. Причина связана с различием в способе, которым Python обрабатывает числа и списки. Сказано, списки являются *изменяемыми* объектами, означающее, что они могут быть изменены, тогда как числа и строки *неизменяемые*, означающее, что они не могут быть изменены. Больше об этом в разделе 19.1

Если мы хотим изменить поведение примера выше так, чтобы *a* изменялось, а *L* нет, делаем следующее:

```
def func1(x):
    x = x + 1
    return x

def func2(L):
    copy = L[ : ]
    copy = copy + [1] a = 3

M = [1, 2, 3]
a = func1(a)    # Обратите внимание на изменение в этой строке
func2(M)
```

13.6 Задания

1. Напишите функцию, называемую *rectangle*, которая принимает два целых числа *m* и *n* как аргументы и выводит прямоугольник размером *m* x *n* состоящий из звездочек. Ниже показан вывод *rectangle*(2, 4)

```
* * * *
* * * *
```

2.(а) Напишите функцию, называемую *add_ excitement*, которая принимает список строк и добавляет восклицательный знак(!) в конец каждой строки списка. Программа должна изменять исходный список и ничего не возвращать.

(б) Напишите ту же самую функцию, за исключением, что она не должна изменять исходный список, а вместо этого должна возвращать новый список.

3. Напишите функцию, называемую *sum_ digits*, которая принимает целое число *num*, а возвращает сумму цифр числа *num*.

4. *Цифровой корень* числа получается следующим образом: складываем числа n, чтобы получить новое число. Складываем числа этого числа, чтобы получить другое новое число. Продолжайте до тех пор, пока не получится число состоящее только из одной цифры. Это число и будет цифровым корнем.

Например, если $n = 45893$, мы складываем цифры, чтобы получить $4 + 5 + 8 + 9 + 3 == 29$. Затем, мы складываем цифры числа 29, чтобы получить $2 + 9 = 11$. Потом, мы складываем цифры числа 11, чтобы получить $1 + 1 = 2$. Так как 2 имеет только одну цифру, следовательно 2 наш цифровой корень.

Напишите функцию, которая возвращает цифровой корень целого числа n.[Обратите внимание: существует короткая запись, где цифровой корень равняется n по модулю 9, но не применяйте это здесь.]

5. Напишите функцию, называемую *first_ diff*, которая принимает две строки и возвращает первое местоположение в котором строки различаются. Если строки одинаковые, то она должна возвращать -1.

6. Напишите функцию, называемую *binom*, которая принимает два целых числа n и k и возвращает биноминальный коэффициент $\binom{n}{k}$, который определяется $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

7. Напишите функцию, которая принимает целое число n и возвращает случайное целое число с точно n цифрами. Например, если n равняется 3, тогда 125 и 593 были бы верными возвращаемыми значениями, но 093 не было бы, потому что оно в реальности 93, которое является двухзначным числом

8. Напишите функцию, называемую *number_ of_ factors*, которая принимает целое число и возвращает, как много факторов(делителей число имеет).

9. Напишите функцию, называемую *factors*, которая принимает целое число и возвращает список его факторов.

10. Напишите функцию, называемую *closest*, которая принимает список чисел L и число n и возвращает самый большой элемент в L, который не больше чем n. Например, если $L = [1, 6, 3, 9, 11]$ и $n = 8$, тогда функция должна возвратить 6, потому что 6 самое близкое значение в L к 8, которое не больше чем 8. Не беспокойтесь если все значения в L меньше чем n.

11. Напишите функцию, называемую *matches*, которая принимает две строки как аргументы и возвращает, как много совпадений существует между строками. Совпадение происходит, где две строки имеют один и тот же символ с одним и тем же индексом. Например, 'python' и 'path'

совпадают с первым, третьим и четвертым символом, поэтому функция должна возвратить 3.

12. Вспомните, что если *s* является строкой, тогда *s.find*('a') найдет местоположение *первой a* в *s*. Проблема состоит в том, что она не находит местоположение каждой *a*. Напишите функцию, называемую *findall*, которой дается строка и одиночный символ и возвращает список всех местоположений того символа в строке. Она должна возвращать пустой список, если символ не появится в строке.

13. Напишите функцию, называемую *change_case*, которой дается строка и возвращает строку в которой каждая заглавная буква заменяется на прописную и наоборот.

14. Напишите функцию, называемую *is_sorted*, которой дается список и возвращает **True**, если список отсортирован и **False** иначе.

15. Напишите функцию, называемую *root*, которой дается число x и целое число n и возвращает $x^{1\backslash n}$. В определении функции установите n значением по умолчанию, равным 2.

16. Напишите функцию, называемую *one_away*, которая принимает две строки и возвращает **True**, если строки имеют одну и ту же длину и отличаются ровно на одну букву, типа *bike/hike* или *water/wafer*.

17.(а) Напишите функцию, называемую *primes*, которой дается число n и возвращает список первых n простых чисел. Установите значение n по умолчанию, равным 100.

(б) Измените функцию выше, чтобы имелся опциональный аргумент *start*, который позволяет списку начинаться со значения, отличным от двух. Функция должна возвращать первые n простых чисел, которые больше чем или равны *start*. Значение по умолчанию *start* должно быть 2.

18. Наша числовая система называется *десятичной*, потому что у нас имеются десять цифр: 0, 1, . . . , 9. Некоторые культуры, включая майя и кельтов, используют 20ю систему. В нашей версии этой системы, 20 цифр представлены буквами от A до T. Вот таблица представляющая несколько преобразований:

10	20	10	20	10	20	10	20
0	A	8	I	16	Q	39	BT
1	B	9	J	17	R	40	CA
2	C	10	K	18	S	41	CB
3	D	11	L	19	T	60	DA
4	E	12	M	20	BA	399	TT
5	F	13	N	21	BB	400	BAA
6	G	14	O	22	BC	401	BAB
7	H	15	P	23	BD	402	BAC

Напишите функцию называемую *base20*, которая переводит десятичные числа в двадцатичные. Одним из способов преобразования является нахождение остатка при делении числа на 20, затем разделите число на 20 и повторяйте процесс, пока число не станет равным 0. Остатки являются цифрами двадцатичной системы в обратном порядке, однако Вы должны перевести цифры в их буквенные эквиваленты.

19. Напишите функцию, называемую *verbose*, которой дается целое число меньше чем 10^{15} возвращает название числа на английском. Как пример, *verbose(123456)* должна возвратить *one hundred twenty-three thousand, four hundred fifty-six.*

20. Напишите функцию называемую *merge*, которая принимает два, уже отсортированных списка, по возможности различной длины, и соединяет их в отдельный отсортированный список.

(а) Сделайте это с использованием метода *sort*.

(б) Сделайте это без использованием метода *sort*.

21. В главе 12 способ, которым мы проверяли было ли слово *w* реальным, являлся:

```
if w in words:
```

где *words* являлся словами из списка, сгенерированным из списка слов. Это к сожалению медленно, но существует быстрый способ, который называется *бинарный поиск*. Чтобы реализовать его в функции, начните сравнивать *w* со значением, которое находится в середине *words*. Если они равны, тогда Вы закончили и функция должна возвратить **True**/ С другой стороны, если *w* находится перед значением в середине, тогда ищите в первой половине списка. Если оно находится после значения, которое в середине, тогда ищите во второй половине списка. Затем повторите процесс на соответствующей половине списка и продолжайте пока слово не будет найдено, или больше нечего искать, в котором случае, функция возвратит **False**. Оператор <and> может быть использован для алфавитного сравнения двух строк.

22. Доску для игры в крестики-нолики можно представить в виде двумерного списка размером 3 × 3, где нули обозначают пустые ячейки, единицы обозначают X, а двойки обозначают O.

(а) Напишите функцию, которой дается такой список и выбирает случайным образом место для размещения 2. Выбранное место на данный момент должно быть равно 0 и место должно быть выбрано.

(б) Напишите функцию, которой дается такой список и проверяет, выиграл ли кто-нибудь. Возвратите **True**, если есть победитель и **False** иначе.

23. Напишите функцию, которой дается список размером 9 x 9 потенциально решенного судоку и возвращает **True** если оно решено правильно и **False** если имеется ошибка. Судоку правильно решена если нет повторяющихся чисел ни в каком ряду или столбце или любом из девяти "блоков ".

Глава 14

Объектно-Ориентированное Программирование

Примерно через год или около того, после того, как я начал программировать, я решил сделать игру, чтобы играть в *Wheel of Fortune*(Колесо фортуны). Я написал программу на языке программирования BASIC и она получилась достаточно большой, пару тысяч строк. Она в основном работала, но всякий раз когда я пытался что-то исправить, мое исправление влияло на что-то в совершенно других частях программы. Затем, я бывало исправлял то и нарушал что-то еще. В конце концов моя программа заработала, но после, какое-то время я боялся даже прикасаться к ней.

Проблема в программе была в том, что каждая часть программы имела доступ к переменным из других частей программы. Изменение переменной в одной части разлаживало вещи в других. Одним из решений этого типа проблемы является *объектно- ориентированное программирование*. Одним из его главных преимуществ является *инкапсуляция*, где Вы разделяете Вашу программу на части и каждая часть функционирует внутри, независимо от других. Части взаимодействуют друг с другом, и им не нужно точно знать, как другие части выполняют свои задачи. Это требует некоторого планирования и время установки, прежде чем Вы запустите Вашу программу, и поэтому это не всегда подходит для коротких программ, подобных многим, которые мы написали не так давно.

Мы просто рассмотрим основы объектно-ориентированного программирования здесь. Объектно-ориентированное программирование широко применяется в разработке програмного обеспечения и я бы рекомендовал подобрать другую книгу по программированию или разработке програмного обеспечения, чтобы узнать больше о разработке программ в объектно-ориентированном подходе.

14.1 Python является объектно-ориентированным

Python является объектно-ориентированным языком программирования и мы уже в действительности используем объектно-ориентированные идеи. Ключевое понятие - это понятие *объект*. Объект состоит из двух вещей: данных и функций(называемых *методами*), которые работают с теми данными. Как пример, строки в Python - это объекты. Данными строкового объекта являются реальные символы, которые составляют ту строку. Методы - это вещи типа *lower, replace* и *split*. В Python все объекты. То включает не только строки и списки, но также целые числа, числа с плавающей точкой и даже сами функции.

14.2 Создание своих собственных классов

Класс является шаблоном для объектов. Он содержит код для всех методов объекта.

Простой пример Здесь простой пример, который демонстрирует как выглядит класс. Он не делает ничего интересного.

```
class Example:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add(self):
        return self.a + self.b

e = Example(8, 6)
print(e.add())
```

- Чтобы создать класс мы используем инструкцию class. Имя класса обычно начинается с заглавной буквы.
- Большинство классов будут иметь метод, который называется __init__. Подчеркивания указывают, что это специальный тип метода. Он называется *конструктор*, и он автоматически вызывается когда кто-то создает новый объект из Вашего класса. Конструктор обычно используется, чтобы установить переменные класса. В программе выше, конструктор принимает два значения *a* и *b*, и присоединяет переменные класса *a* и *b* к этим значениям.
- Первый аргумент каждого метода в Вашем классе - это специальная переменная, называемая self. Предназначение self - это, чтобы различать переменные и методы класса от других переменных и функций в программе.
- Чтобы создать новый объект из класса, Вы вызываете имя класса со значением, которое Вы хотите послать в конструктор. Обычно Вы будете хотеть присвоить его к имени переменной. Это то, что делает строка *e = Example*(8, 6)
- Чтобы использовать методы объекта, применяйте оператор *dot*(точка), как в *e.addmod*().

Более практичный пример Здесь класс, называемый *Analyzer*, который выполняет некоторый простой анализ строки. Существуют методы, которые возвращают - сколько слов в строке, сколько их данной длины и сколько начинаются с данной буквы.

```
from string import punctuation
class Analyzer:
    def __init__(self, s):
        for c in punctuation:
            s = s.replace(c, '')
        s = s.lower()
        self.words = s.split()

    def number_of_words(self):
        return len(self.words)

    def starts_with(self, s):
        return len([w for w in self.words if w[:len(s)] == s])

    def number_with_length(self, s):
        return len([w for w in self.words if len(w) == n])
```

```
s = 'This is a test of the class.'
analyzer = Analyzer(s)
print(analyzer.words)
print('Количество слов:', analyzer.number_of_words())
print('Количество слов начинающихся с "t":', analyzer.starts_with('t'))
print('Количество слов из двух букв:', analyzer.number_with_length(2))
```

```
[ 'this',  'is',  'a',  'test',  'of',  'the',  'class']
Количество слов: 7
Количество слов начинающихся с 't': 3
Количество слов из двух букв: 2
```

Несколько замечаний об этой программе:

- Одна причина, почему мы бы заключили этот код в класс, заключается в том, что мы могли потом использовать его в множестве разных программ. Это также хорошо просто для организации вещей. Если все, что делает наша программа - это просто анализирует некоторые строки, тогда нет большого смысла в написании класса, но если это должно быть частью большой программы, тогда использование класса предоставляет хороший способ отделить код *Analyzer* от другого кода. Это также означает, что если бы мы изменили внутреннее содержание класса *Analyzer*, остальная часть программы не изменилась бы до тех пор пока интерфейс, способ остальной части программы взаимодействовать с классом, не измениться. Также, класс *Analyzer* может быть импортирован как-есть в другие программы.

- Следующая строка имеет доступ к переменной класса:

  ```
  print(analyzer.words)
  ```

  Вы также можете изменять переменные класса. Это не всегда хорошая вещь. В некоторых случаях это удобно, но Вы должны быть осторожны с этим. Беспорядочное использование переменных класса идет против идеи инкапсуляции и может привести к ошибкам программирования, которые тяжело исправить. Некоторые другие объектно- ориентированнные языки программирования имеют понятие публичных и частных переменных, публичные переменные - это те, к которым любой может иметь доступ и возможность изменить, а частные доступны только методам внутри класса. В Python все переменные публичные и ответственность за них несет программист. Существует соглашение, по которому Вы присваиваете имена этим переменным, чтобы они были частными, с начальным подчеркиванием, типа _var1. Это служит для того, чтобы дать знать другим, что эти переменные являются внутренними для класса и не должны изменяться.

14.3 Наследование

В объектно-ориентированном программировании имеется понятие, называемое *наследование*, это когда Вы можете создать класс, который строится на основе другого класса. Когда Вы делаете это, новый класс получает все переменные и методы из класса от которого он наследуется(называется *базовый класс*). Он может затем определять дополнительные переменные и методы, которые не существуют в базовом классе и он также может *перегружать* некоторые из методов базового класса. То есть, он может переписать их подгоняя под собственные нужды. Вот простой пример:

```
class Parent :
    def __init__(self, a):
        self.a = a
    def method1(self):
        return self.a * 2
    def method1(self):
        return self.a + '!!!'

class Child(Parent):
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def method1(self):
        return self.a * 7
    def method3(self):
        return self.a + self.b

c = Child('привет', 'пока' )

print('Базовый method 1: ', p.method1())
print('Базовый method 2: ', p.method2())
print()
print('Наследуемый method 1:', c.method1())
print('Наследуемый method 2:', c.method2())
print('Наследуемый method 3:', c.method3())
```

```
Базовый method 1: приветпривет
Базовый method 2: привет !!!

Наследуемый method 1: приветприветприветприветприветприветпривет
Наследуемый method 2: привет!!!
Наследуемый method 3: приветпока
```

Мы видим в примере выше, что наследуемый класс перегрузил базовый *method1*, выражая это, как повторение строки семь раз.Наследуемый класс унаследовал базовый *method2*, поэтому он может использовать его без определения. Наследуемый класс также добавляет некоторые возможности в базовый класс, а именно новую переменную *b* и новый метод *method3*.

Замечание о синтаксисе: при наследовании от класса, Вы указываете базовый класс в круглых скобках в инструкции class.

Если наследуемый класс добавляет некоторые новые переменные, он может вызывать конструктор базового класса, как продемонстрировано ниже. Другое применение заключается в том, что наследуемый класс просто хочет дополнить один из методов базового класса. В примере ниже, метод наследуемого класса *print_var* вызывает метод базового класса и добавляет дополнительную строку.

```
class Parent:
    def __init__(self, a):
        self.a = a
```

```
    def print_var(self):
        print('Значение переменных этого класса равно :')
        print (self.a)

class Child(Parent):
    def __init__(self, a, b):
        Parent.__init__(self, a)
        self.b = b

    def print_var(self):
        Parent.print_var(self)
        print(self.b)
```

Обратите внимание Вы можете также наследовать из Python встроенные типы, как строки (**str**) и списки (**list**), а также классы, определенныев различных модулях, которые идут с Python.

Обратите внимание Ваш код может наследоваться более чем от одного класса за раз, хотя это может быть немного сложно.

14.4 Пример карточной игры

В этом разделе мы покажем, как разрабатывать программу с классами. Мы создадим простую hi-low карточную игру, в которой пользователю выдается карта и он должен сказать будет ли следующая карта выше или ниже чем эта. Эта игра может быть легко сделана без классов, но мы создадим классы, чтобы представить карту и колоду карт, и эти классы могут быть применены в других карточных играх.

Мы начнем с класса для игральной карты. Информация ассоциированная с картой состоит из её значения (от 2 до 14) и её масти. У класса *Card*, ниже, имеется только один метод __str__. Это специальный метод, который среди других вещей, говорит функции **print**, как выводить объект *Card*.

```
class Card:
    def __init__(self, value, suit):
        self.value = value
        self.suit = suit

    def __str__(self):
        names = ['Валет', 'Дама', 'Король', 'Туз']
        if self.value <= 10:
            return ' { } of { }'. format(self.value, self.suit)
        else:
            return ' { } of { }'. format(names[self.value - 11], self.suit)
```

Дальше у нас имеется класс представляющий группу карт. Его данные состоят из списка объектов *Card*. Он имеет ряд методов: *nextCard*, который удаляет первую карту из списка и возвращает её; *hasCard*, которая возвращает **True** или **False**, в зависимости от того, остались какие-нибудь карты в списке; *size*, который возвращает количество карт в списке; и *shuffle*, который перемешивает список.

```
import random
class Card_group:
    def __init__(self, cards = [ ]):
        self.cards = cards

    def nextCard(self):
        return self.cards.pop(0)

    def hasCard(self):
        return len (self.cards) > 0

    def size(self):
        return len (self.cards)

    def shuffle(self):
        random.shuffle(self.cards)
```

У нас есть ещё один класс *Standart_ deck*, который наследуется из *Card_ group*. Идея здесь заключается в том, что *Card_ group* представляет произвольную группу карт, а *Standart_ deck* представляет специальную группу карт, а именно стандартную колоду из 52 карт, которая применяется в большинстве карточных игр.

```
class Standart_deck(Card_group):
    def __init__(self):
        self.cards = [ ]
        for s in [ 'Черви', 'Бубы', 'Трефы', 'Пики' ]:
            for v in range(2, 15):
                self.cards.append(Card(v, s))
```

Предположим, мы только что создали отдельный класс, который представляет стандартную колоду со всеми обычными операциями, типа перемещивания. Если бы мы захотели создать новый класс для игры Pinochle или какую-либо игры, которая не использует стандартную колоду, тогда мы должны бы были скопировать и вставить код стандартной колоды и изменить множество вещей. Делая вещи более обобщенно, как мы сделали здесь, каждый раз когда нам потребуется новый тип колоды, мы можем выстроить(унаследовать) из того что есть в *Card_ group*. Например, класс для Pinochle колоды выглядел бы как:

```
class Pinochle_deck(Card_group):
    def __init__(self):
        self.cards = [ ]
        for s in [ 'Черви', 'Бубы', 'Трефы', 'Пики' ] * 2:
            for v in range(9, 15):
                self.cards.append(Card(v, s))
```

Колода для Pinochle имеет только девятки, десятки, вальтов, дам, королей и тузов. Существуют две копии каждой карты, каждой масти.

Вот hi-low программа, которая использует классы, разработанные нами здесь. Один из способов представить, что мы сделали с классами, заключается в том , что мы создали миниатюрный карточный язык программирования, на котором мы можем думать, как работает карточная игра

и не переживать о том, как именно карты перемешиваются или раздаются или что-то еще, так как это вложено в классы. Для hi-low игры мы возьмем новую колоду карт, перемешаем её, а затем раздадим карты, одну за раз. Когда у нас закончатся карты мы возьмем новую колоду и перемешаем её. Хорошая особенность этой игры заключается в том, что она раздает все 52 карты колоды, поэтому игрок может задействовать свою память, чтобы помочь самому себе играть в игру.

```
deck = Standart_deck()
deck.shuffle()

new_card = deck.nextCard()
print('\ n', new_card)
choice = input('Выше (в) или ниже (н) :')
streak = 0

while  (choice == 'в'  or  choice == 'н'):
    if not deck.hasCard():
          deck = Standart_deck()
          deck.shuffle()

    old_card = new_card
    new_card = deck.nextCard()

    if  (choice.lower() == 'в'  and  new_card.value>old_card.value  or \
        choice.lower() == 'н'  and  new_card.value>old_card.value):
        streak = streak + 1
        print('Правильно! То есть', streak, 'подряд!')
    elif  (choice.lower() == 'в'  and  new_card.value<old_card.value  or \
        choice.lower() == 'н'  and  new_card.value>old_card.value):
        streak = 0
        print('Wrong.')
    else:
        print('Push.')
    print('\ n', new_card)
    choice = input('Выше (в) или ниже (н) :')
```

```
Король Треф
Выше (в) или ниже (н): н
Правильно!  То есть 1 подряд!

2 Пики
Выше (в) или ниже (н): в
Правильно!  То есть 2 подряд!
```

14.5 Пример игры Крестики-нолики

В этом разделе мы создадим объектно-ориентированную игру "Крестики-нолики". Мы используем класс, чтобы запаковать логику игры. Класс содержит две переменные, целое число представляющее текущего игрока и список размером 3 x 3 представляющий доску. Переменная представляющая доску состоит из нулей, единиц и двоек. Нули представляют пустую клетку, в то время как единицы и двойки представляют ячейки отмеченные игроками 1 и 2 соответственно. Имеются четыре метода:

- *get_open_spots* — возвращает список мест на доске, которые еще не были отмечены игроками
- *is_valid_move* — принимает ряд и колонку представляющие потенциальный ход и возвращает **True** если движение допустимо и **False** иначе
- *make_move* — принимает ряд и колонку представляющие потенциальный ход, вызывает *is_valid_move*, чтобы проверить соответствует ли ход и если да, устанавливает массив доски соответственно и меняет игрока.
- *check_for_winner* — сканирует список, который представляет доску и возвращает 1 если игрок 1 выиграл, 2 если игрок 2 выиграл, 0 если не осталось ходов и нет победителей, и -1 если игра должна продолжаться

Здесь код для класса:

```
class tic_tac_toe :
    def __init__(self):
        self.B = [ [ 0, 0, 0],
                   [ 0, 0, 0],
                   [ 0, 0, 0]]
        self.player = 1

    def get_open_spots(self):
        return [ [r, c] for r in range(3) for c in range(3)
                 if self.B [r] [c] == 0]

    def is_valid_move(self, r, c):
        if 0<=r<=2 and 0<=c<=2 and self.B [r] [c] == 0]:
            return True
        return False

    def make_move( self, r, c):
        if self,is_valid_move(r, c):
            self.B[r] [c] = self.player
            self.player = (self.player + 2)%2 + 1

    def check_for_winner(self):
        for c in range(3):
            if self.B[0] [c] == self.B[1] [c] == self.B[2] [c] != 0:
                return self.B[0] [c]
        for r in range(3):
            if self.B[r] [0] == self.B[r] [1] == self.B[r] [2] != 0:
                return self.B[r] [0]
        if self.B[0] [0] == self.B[1] [1] == self.B[2] [2] !=0:
            return self.B[0] [0]
        if self.B[2] [0] == self.B[1] [1] == self.B[0] [2] !=0:
            return self.B[2] [0]
        if self.get_open_spots() == [ ]:
            return 0
        return -1
```

Этот класс содержит логику игры. В классе нет ничего, что определяет интерфейс пользователя. Ниже у нас есть текстовой интерфейс использующий инструкции print и input. Если мы решим применить графический интерфейс, мы можем использовать класс *Tic_tac_toe* без изменения

что-либо в нем. Обратите внимание, что метод *get_open_spots* не используется этой программой. Он полезен, однако, если Вы захотите реализовать компьютер, как игрока. Компьютер, в качестве простого игрока, вызывал бы этот метод и применяя метод *random.choice*, чтобы выбрать случайный элемент из возвращаемого списка ячеек.

```
def print_board():
    chars = [ '–', 'X', 'O']
    for  r  in  range(3):
            for  c  in  range(3):
                    print(chars[ game.B[r] [c]], end = '  ')
            print()

game = tic_tac_toe()
while  game.check_for_winner() == -1:
            print_board()
            r, c = eval(input('Введите ячейку, игрок' + str(game.player) + ': '))
            game.make_move(r, c)

print_board()
x = game.check_for_winner()
if  x == 0:
            print('Это ничья ')
else:
            print('Игрок',  x,  'выиграл!')
```

```
–  –  –
–  –  –
–  –  –
Введите ячейку,  игрок 1: 1, 1
–  –  –
–  X  –
–  –  –
Введите ячейку,  игрок 2: 0, 2
–  –  O
–  X  –
–  –  –
Введите ячейку,  игрок 1:
```

14.6 Дальнейшие темы

- **Специальные методы** — Мы уже увидели два специальных метода, конструктор *__init__* и метод *__str__*, который определяет как Ваши объекты будут выглядить при выводе. Существует много других. Например, есть *__add__*, который позволяет Вашему объекту применять оператор *+*. Существуют специальные методы для всех операторов Python. Также имеется метод, называемый *__len__*, который позволяет Вашему объекту работать со встроенной функцией **len**. Ещё есть специальный метод *__getitem__*, который позволяет Вашей программе работать с квадратными скобками списка и строки [] .

- **Копирование объектов** — Если Вы хотите сделать копию объекта *x*, это недостаточно сделать следующее:

```
x_copy = x
```

Причина рассматривается в разделе 19.1. Вместо этого делайте следующее:

```
from copy import copy
x_copy = copy(x)
```

- **Вставка Вашего кода в различные файлы** — Если Вы хотите многократно использовать класс в нескольких программах, то Вам не нужно копировать и вставлять код в каждый. Вы можете сохранить его в файле и используя инструкцию **import** импортировать код в Ваши программы. Файлу нужно будет находиться где-то, чтобы Ваша программа могла найти его, как в той же самой директории.

```
from analyzer import Analyzer
```

14.7 Задания

1. Напишите класс, который называется *Investment* с полями называемыми *principal* и *interest* . Конструктор должен установить значения для этих полей. В нем должен быть метод называемый *value_after*, который возвращает значение инвестиции через n лет. Формула для этого $p(1+i)^n$, где p - основная сумма, а i - процентная ставка. Класс должен также использовать специальный метод *__str__* так, чтобы выведение объекта приводило к результату вроде приведенного внизу:

Основная сумма - $1000.00, Процентная ставка - 5.12%

2. Напишите класс называемый *Product*. Класс должен иметь поля называемые - *name, amount* и *price*, содержащие наименование товара, число предметов того товара на складе и обычная цена товара. В классе должен быть метод *get_price*, который получает число предметов подлежащих покупке и возвращает стоимость покупки того количества предметов, где обычная цена берется для заказов меньше чем 10 предметов, скидка 10% применяется для заказов от 10 до 99 предметов, и скидка 20% применяется для заказов 100 или больше предметов. Также должен быть метод называемый *make_purchase*, который получает количество предметов подлежащих покупке и уменьшающий *amount* на то значение.

3. Напишите класс называемый *Password_manager*. Класс должен иметь список называемый *old_passwo* который содержит все предыдущие пароли пользователя. Последний элемент списка является текущим паролем пользователя. Там должен быть метод называемый *set_password*, который устанавливает пароль пользователя. Метод *set_password* должен изменять пароль, только если введенный пароль отличается от всех предыдущих паролей пользователя. Наконец, создайте метод называемый *is_correct*, который получает строку и возвращает булевы значения **True** или **False** в зависимости от того, является ли строка равной текущему паролю или нет.

4. Напишите класс называемый *Time*, у которого единственное поле является временем в секундах. Он должен иметь метод называемый *convert_to_minutes*, который возвращает строку минут и секунд отформатированную как в следующем примере: если секунды равняются 230, метод должен возвратить '5:50'. Он также должен иметь метод называемый *convert_to_hours*, который возвращает строку часов, минут и секунд, отформатированных аналогично предыдущему методу.

5. Напишите класс называемый *Wordplay*. Он должен иметь поле, которое содержит список слов. Пользователь класса должен передать список слов, который они хотят использовать в классе. В классе должны быть следующие методы:

- *words_with_length(length)* — возвращает список всех слов длиной *length*

- *starts_with(s)* — возвращает список всех слов, которые начинаются с *s*
- *ends_with(s)* — возвращает список всех слов, которые заканчиваются на *s*
- *palindromes()* — возвращает список всех палиндромов в списке
- *only(L)* — возвращает список всех слов, которые содержат только эти буквы в *L*
- *avoids(L)* — возвращает список всех слов, которые не содержат ни одну букву в *L*

6. Напишите класс называемый *Converter*. Пользователь передаст длину и единицу измерения при объявлении объекта из класса – например, *c* = *Converter*(9, 'дюймы'). Возможные единицы измерения - дюймы, футы, ярды, мили, километры, сантиметры и миллиметры. Для каждой из этих размерностей в классе должен быть метод, который возвращает длину переведенную в эти размерности. Например, используя объект *Converter* созданный выше, пользователь может вызвать *c.feet* и должен получить 0.75 как результат.

7. Используя класс *Standart_deck* из этого раздела, создайте упрощенную версию игры *War*. В этой игре имеются два игрока. У каждого на старте половина колоды. Каждый игрок сдает верхнюю карту из своих карт и всякий, у кого выше карта выигрывает карту другого игрока и добавляет их на дно своей колоды. В случае ничьей, две карты удаляются из игры(это отличает от настоящей игры, но упрощает программу). Игра заканчивается, когда у одного игрока заканчиваются карты.

8. Напишите класс, который наследуется из *Card_group* этого раздела. Класс должен представлять колоду карт, которая содержит только черви и пики, только с картами от 2 до 10 каждой масти. Добавьте в класс метод называемый *next2*, который возвращает две верхние карты из колоды.

9. Напишите класс называемый *Rock_paper_scissors*, который реализует логику игры Rock-paper-scissors. В этой игре пользователь играет против компьютера определенное число раундов. Ваш класс должен иметь поля - сколько раундов будет, номер текущего раунда, и число побед, которые каждый игрок имеет. В классе должны быть методы - для получения выбора компьютера, определения победителя раунда и проверки того, есть ли у кого-то один вариант (всей) игры.

10.(а) Напишите класс называемый *Connect4*, который реализует логику игры 'Четыре в ряд'(Connect4). Используйте класс *Tic_tac_toe* из этого раздела, как начальную точку.

(б) Используйте класс *Connect4*, чтобы создать простую текстовую версию игры.

11. Напишите класс называемый *Poker_hand*, имеющий поле, которое является списком объектов *Card*. В нем должны быть следующие, не требующие объяснения, методы:

has_royal_flush(флеш-рояль,5 старших карт одной масти, начинающихся с туза), has_straight_flush(стрит-флеш,5 карт одной масти, следующих друг за другом по достоинству), has_four_of_a_kind(каре, 4 карты одного достоинства и одной произвольной), has_full_house(полный дом, 3 карты и 2 карты одного достоинства), has_flush(флеш - масть, 5 карт одной масти), has_straight(стрит - порядок, 5 карт по порядку любых мастей),
has_three_of_a_kind(3 карты одного достоинства), has_two_pair(две пары, 2 пары карт одного достоинства), has_pair(пара, 2 карты одного достоинства)

В классе также должен быть метод называемый *best*, который возвращает строку, указывающую, какая лучшая комбинация может быть составлена из этих карт.

Часть II

Графика

Глава 15

GUI программирование с Tkinter

До сих пор, единственным способом, которым наши программы были способны взаимодействовать с пользователем является ввод с клавиатуры посредством инструкции **input**. Но большинство реальных программ используют окна, кнопки, полосы прокрутки и другие различные вещи. Эти *виджеты*(небольшие графические приложения) являются частью того, что называется *Graphical User Interface*(Графический пользовательский интерфейс) или GUI. Этот раздел о GUI программировании в Python с Tkinter.

Все виджеты, которые мы будем рассматривать имеют намного больше возможностей, чем мы могли бы охватить здесь. Отличной ссылкой является Introduction to Tkinter [2]

15.1 Основы

Почти каждая GUI программа, которую мы будем писать, содержит следующие три строки:

```
from tkinter import *
root = Tk() mainloop()
```

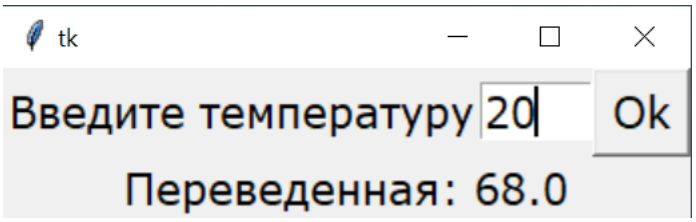
Первая строка импортирует весь GUI материал из модуля *tkinter*. Вторая строка создает окно на экране, которое мы назвали *root*. Третья строка помещает программу в то, что по сути является длительным циклом while, которое называется *event loop*(циклом событий). Этот цикл продолжается, ожидая нажатия клавиш, нажатия кнопок и так далее, и программа выходит из цикла когда пользователь закрывает окно.

Вот работающая GUI программа, которая переводит температуры из Фаренгейта в Цельсия

```
from tkinter import *
def calculate():
    temp = int (entry.get())
    temp = 9/5 * temp + 32
    output_label.configure(text = 'Переведенная:  { : . 1f }' . format(temp))
    entry.delete(0 , END)
root = Tk()
message_label = Label(text = 'Введите температуру',
                      font = ('Verdana', 16))
output_label = Label(font = ('Verdana', 16))
entry = Entry(font = ('Verdana', 16), width = 4)
calc_button = Button(text = 'Ok', font = ('Verdana', 16),
                     command = calculate )
```

```
message_label.grid(row = 0, column = 0)
entry.grid(row = 0, column = 1)
calc_button.grid(row = 0, column = 2)
output_label.grid(row = 1, column = 0, columnspan = 3)

mainloop()
```

Вот как программа выглядит:



Сейчас, мы исследуем компоненты программы отдельно.

15.2 Метки

Метка(текстовой виджет) - это место для Вашей программы, чтобы разместить какой-либо текст на экране. Следующий код создает метку и располагает ее на экране.

```
hello_label = Label(text = 'привет')
hello_label.grid(row=0, column=0)
```

Мы вызываем *Label*, чтобы создать новую метку. Заглавная *L* необходима. Именем нашей метки является *hello_label*. После создания используйте метод *grid*, чтобы разместить метку на экране. Мы объясним *grid* в следующем разделе.

Параметры Имеется ряд параметров, которые Вы можете изменить, включая размер шрифта и цвет. Вот несколько примеров:

```
hello_label = Label(text='привет', font=('Verdana', 24, 'bold'),
                    bg='blue', fg='white')
```

Обратите внимание на использование аргументов ключевых слов. Здесь несколько общих параметров:

- *font* — Базовая структура - *font*= (*font name, font size, style*). Вы можете опустить размер шрифта(font size) или стиль(style). Вариантами для стиля являются 'bold', 'italic', 'underline', 'overstrike', 'roman', 'normal'(который по умолчанию). Вы можете комбинировать различные стили, как: 'bold italic'.
- *fg* и *bg* — Они обозначают *foreground* и *background*. Общераспространенные имена цветов могут быть использованы, как 'blue', 'green' и так далее. Раздел **??** описывает, как получить по сути любой цвет.
- *width* — Это определяет, сколько символов в длину метка должен быть. Если Вы пропустите его, Tkinter будет основывать длину из текста, который Вы положите в метку. Это может привести к непредсказуемым результатам, поэтому хорошо определить раньше времени желаемую длину Вашей метки и установить *width* соответственно.

- *height* — Это определяет, сколько строк в высоту метка должна быть. Вы можете использовать это многостроковых меток. Используйте символ новой строки в тексте, чтобы метка охватывала разные строки. Например, *text*='hi\ nthere'.

Существует более дюжины параметров. Вышеупомянутое *Introduction to Tkinter* имеет отличный список других и что они делают.

Изменение свойств метки Позднее в Вашей программе, после того как создали метку, Вы можете захотеть изменить что-то в ней. Вот два примера, в которых изменяются свойства метки называемой *label*:

```
label.configure(text='Пока')
label.configure(bg='white', fg='black')
```

Устанавливая *text* к чему-либо используя метод *configure*, это похоже на GUI эквивалент инструкции **print**. Однако, в вызовах *configure*, мы не можем использовать запятые, чтобы разделить вещи при выводе. Вместо этого, нам нужно применять строковое форматирование. Вот инструкция **print** и его эквивалент используя метод *configure*.

```
print('a = ', a, 'and b = ', b)
label.configure(text= 'a =  , { } ,  and b  { }  '. format(a, b))
```

Метод *configure* работает с большинством других виджетов, которые мы увидим.

15.3 grid

Метод *grid* используется для размещения объектов на экране. Он отображает экран как прямоугольную сетку строк и колонок. Первые несколько строк и колонок показаны ниже.

(row=0, column=0)	(row=0, column=1)	(row=0, column=2)
(row=1, column=0)	(row=1, column=1)	(row=1, column=2)
(row=2, column=0)	(row=2, column=1)	(row=2, column=2)

Охватывание несколько строк или колонок. Имеются опциональные аргументы, *rowspan* и *columnspan*, которые позволяют виджету занимать больше чем одну строку или колонку. Вот пример нескольких инструкций *grid* отвечающих за то, как будет выглядеть компоновка:

```
label1.grid(row=0, column=0)
label2.grid(row=0, column=1)
label3.grid(row=1, column=0, columnspan=2)
label4.grid(row=1, column=2)
label5.grid(row=2, column=2)
```

<table>
<tr><td>label1</td><td>label2</td><td></td></tr>
<tr><td colspan="2">label3</td><td>label4</td></tr>
<tr><td></td><td></td><td>label5</td></tr>
</table>

Интервал Чтобы добавить дополнительное пространство между виджетами, существуют опциональные аргументы *padx* и *pady*.

Важное замечание Всегда, когда Вы создаете виджет, чтобы разместить его на экране Вам нужно использовать *grid*(или его собрата *pack*, о котором мы будем говорить позднее). Иначе он не будет видимым.

15.4 Поля ввода

Поля ввода являются способом для Вашего GUI получить текстовой ввод. Следующий пример создает простое поле ввода и размещает его на экране.

```
entry = Entry()
entry.grid(row=0, column=0)
```

Большинство тех же самых опций, которые работают с ярлыками, работают с полями ввода(и большинство других виджетов, о которых мы будем говорить). Опция *width* особенно полезная потому, что поле ввода часто будет шире чем Вам нужно.

- **Получение текста** Чтобы получить текст из поля ввода используйте метод *get*. Это возвратит строку. Если Вам нужны числовые данные, используйте **eval**(или **int** или **float**) со строкой. Вот простой пример, в котором получаем текст из поля ввода называемого **entry**.

```
string_value = entry.get()
num_value = eval(entry.get())
```

- **Удаление текста** Чтобы очистить поле ввода используйте следующее:

```
entry.delete(0, END)
```

- **Вставка текста** Чтобы вставить текст в поле ввода используйте следующее:

```
entry.insert(0, 'привет')
```

15.5 Кнопки

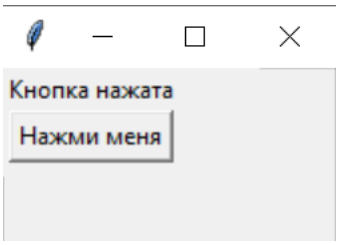
Следующий пример создает простую кнопку:

```
ok_button = Button(text='Ok')
```

Чтобы заставить кнопку что-то сделать, когда на ней щелкают мышкой, применяйте аргумент *command*. Ему присваивается имя функции, называемой *callback function*(функция обратного вызова). Когда на кнопку нажимают, функция обратного вызова вызывается. Вот пример:

```
from tkinter import *

def callback():
    label.configure(text='Кнопка нажата')

root = Tk()
label = Label(text='Не нажата')
button = Button(text='Нажми меня', command=callback)

label.grid(row=0, column=0)
button.grid(row=1, column=0)

mainloop()
```

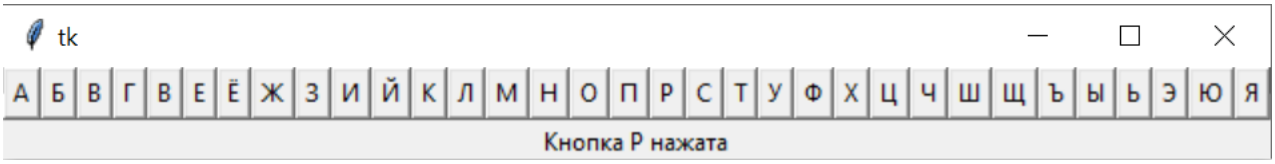
Когда программа запустится, метка скажет *Нажми меня*. Когда кнопка нажимается, вызывается функция обратного вызова, которая изменяет метку, чтобы сказать *Кнопка нажата*.



Трюк с lambda Иногда нам захочется передать информацию в функцию обратного вызова, как если,когда у нас есть несколько кнопок, которые используют ту же самую функцию обратного вызова и мы хотим дать функции информацию о том какая кнопка нажимается. Здесь пример, в котором мы создаем 33 кнопки, по одной для каждой буквы алфавита. Вместо того, чтобы использовать 33 отдельных инструкций *Button* и 33 разных функций, мы применим список и одну функцию.

```
from tkinter import *
alphabet = 'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ'

def callback(x):
    label.configure(text='Кнопка { } нажата'. format(alphabet[x]))

root = Tk()

label = Label()
label.grid(row=1, column=0, columnspan=33)

buttons = [0]*33 # создает список содержащий 33 кнопки
for i in range(26):
    buttons[i] = Button(text=alphabet[i],
                        command = lambda x=i: callback(x))
    buttons[i].grid(row=0, column=i)

mainloop()
```

Мы обратим внимание на несколько вещей в этой программе. Сначала мы устанавливаем *buttons=[0]* ⋆ *33*. Это создает список с 33 объектами в нем. Нам не нужно реально переживать чем эти объекты являются, потому что они будут заменены кнопками. Альтернативным способом для создания списка было бы установить *buttons=[ ]* и использовать метод *append*.

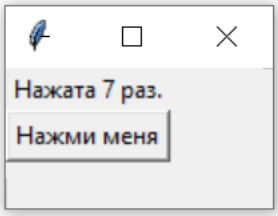


Мы используем только одну функцию обратного вызова и она имеет один аргумент, который отображает, какая кнопка была нажата. Что касается трюка с lambda, не вдаваясь в детали, *command=callback(i)* не работает, и поэтому мы прибегаем к приему lambda.

15.6 Глобальные переменные

Скажем, мы хотим отследить сколько раз кнопка нажимается. Легким способом сделать это, является использование глобальной переменной, как показано ниже:

```
from tkinter import *

def callback():
    global num_clicks
    num_clicks = num_clicks + 1
    label.configure(text='Нажата { } раз ' . format(num_clicks))

num_clicks = 0
root = Tk()

label = Label(text='Не нажата')
button = Button(text='Нажми меня', command = callback)

label.grid(row=0, column=0)
button.grid(row=1, column=0)

mainloop()
```



Мы будем использовать несколько глобальных переменных в наших GUI программах. В использовании глобальных переменных нет необходимости, особенно в длинных программах может привести к трудностям в нахождении ошибок, что делает программы трудными в обслуживании, но в коротких программах, которые мы будем писать, у нас должно быть хорошо. Объектно-ориентированное программирование предоставляет альтернативу глобальным переменным.

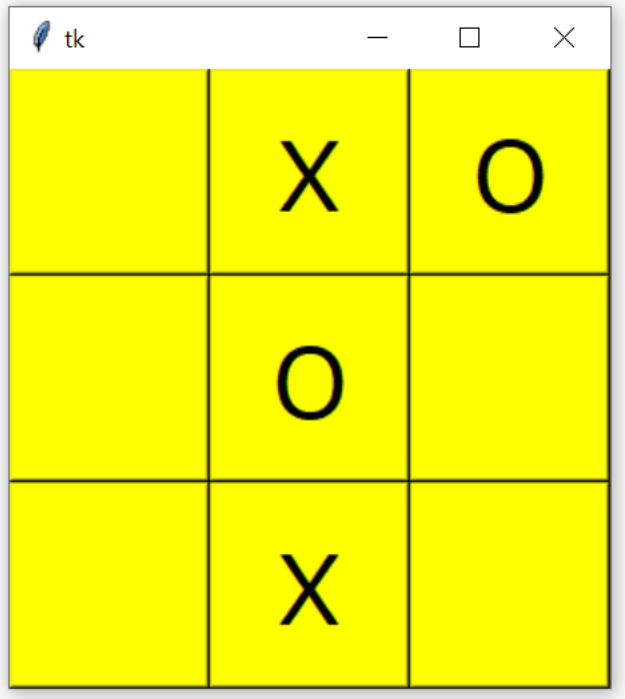
15.7 Крестики-нолики

Применяя Tkinter, мы можем сделать работающую программу 'крестики-нолики' только из 20 строк:

```
from tkinter import *

def callback(r, c):
    global player
    if player == 'X':
        b[r][c].configure(text = 'X')
        player = 'O'
    else:
        b[r][c].configure(text = 'X')
```

```
            player = 'O'

root = Tk()

b = [ [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0], ]

for i in range(3):
    for j in range(3):
            b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                             command = lambda r=i, c=j: callback(r,c))
            b[i][j].grid(row = i, column = j)

player = 'X'

mainloop()
```

Программа работает, хотя она, на самом деле, имеет несколько проблем, таких как позволяет Вам изменять клетку, которая уже содержит что-то. Мы исправим это скоро. Сначала давайте взглянем как программа делает то, что она делает. Начнем с конца, у нас имеется переменная *player*, которая отслеживает чьим ходом она является. Перед этим мы создали доску, которая состоит из девяти кнопок, хранящиеся в двумерном списке. Мы применили трюк lambda, чтобы передать значения строки и колонки нажатой кнопки в функцию обратного вызова. В функции обратного вызова мы пишем X или O в кнопку, которая была нажата и изменяем значение глобальной переменной *player*.



Устранение проблем Чтобы устранить проблему в способности изменять клетку, которая уже содержит что-то, нам нужно иметь способ узнать, какие клетки содержат X, какие O, а какие пустые. Одним из способом является применение метода *Button*, чтобы спросить кнопку, какой текст она содержит. Другим способом, который мы применим здесь, является создание нового двумерного списка, который мы назовем *states*, чтобы отслеживать объекты. Вот код.

```
from tkinter import *

def callback(r, c):
    global player
    if player == 'X' and states[r][c] == 0:
        b[r][c].configure(text = 'X')
        states[r][c] = 'X'
        player = 'O'
    if player == 'O' and states[r][c] == 0:
        b[r][c].configure(text = 'O')
        states[r][c] = 'O'          player = 'X'

root = Tk()

states = [ [0, 0, 0],
           [0, 0, 0],
           [0, 0, 0] ]

b = [ [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0] ]

for i in range(3):
    for j in range(3):
        b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                         command = lambda r=i, c=j: callback(r,c))
        b[i][j].grid(row = i, column = j)

player = 'X'

mainloop()
```

Мы не добавили много в программу. Многое из нового решения происходит в функции обратного вызова. Каждый раз, когда кто-то нажимает на клетку, мы сначала проверяем, является ли она пустой (соответствующий индекс в *states* равняется 0), и если она пустая, то мы отображаем на экране X или O и записываем новое значение в *states*. Многие игры имеют переменную, подобную *states*, которая отслеживает что содержится на доске.

Проверка наличия победителя У нас появится победитель, когда имеется три X или O в строке, а также по вертикали, горизонтали или диагонали. Чтобы проверить, имеются ли три подряд на верхней строке, мы можем использовать следующую инструкцию if:

```
if states[0][0] == states[0][1] == states[0][2] !=0:
    stop_game=True
    b[0][0].configure(bg='grey')
    b[0][1].configure(bg='grey')
    b[0][2].configure(bg='grey')
```

Эта инструкция проверяет, содержит ли каждая из клеток одну и ту же ненулевую запись. Здесь мы применяем короткую запись из раздела 10.3 в инструкции if. Существуют более более подробные инструкции if, которые бы работали. Если мы, все же, найдем победителя,то выделим выигрышные клетки, а затем установим переменную *stop_game* равную True. Эта переменная

будет использоваться в функции обратного вызова. Всякий раз, когда переменная равна **True**, мы не должны позволять происходить какому-либо движению.

Затем, чтобы проверить три подряд в средней строке, измените первую координату с 0 на 1 во всех трех ссылках, а чтобы проверить имеются ли три подряд в нижней строке измените 0 на 2. Так как у нас будут три очень похожие инструкции if, которые отличаются только в одной позиции, то может быть использован цикл for, чтобы сделать код короче:

```
for i in range(3):
    if states[i][0] == states[i][1] == states[i][2] !=0:
            b[i][0].configure(bg='grey')
            b[i][1].configure(bg='grey')
            b[i][2].configure(bg='grey')
            stop_game = True
```

Далее, проверка победителя по вертикали почти та же самая, за исключением того, что мы изменяем вторую координату вместо первой. Наконец, у нас имеются две дополнительные инструкции if, чтобы разобраться с диагоналями. Полная программа имеется в конце этого раздела. Мы также добавили несколько параметров цвета к инструкциям *configure*, чтобы игра выглядела более красивой.

Дальнейшие улучшения Дальше было бы легко добавить кнопку перезапуска. Функция обратного вызова для этого должна вернуть *stop_game* значение **False**, она должна обнулить список *states*, а также установить все кнопки к *text* ='' и *bg*='yellow'.

Добавить компьютер в качестве игрока, также, было бы не трудно, если Вы не возражаете против простого компьютерного игрока, который ходит случайным образом. Это заняло бы, примерно, 10 строчек кода. Сделать разумного компьютерного игрока, также, не трудно. Такой игрок должен стремиться к двум подряд O или X, чтобы попытаться выиграть или заблокировать, а кроме того избегать попадания в безнадежную ситуацию.

```
from tkinter import *

def callback(r, c):
    global player

    if player == 'X' and states[x][y] == 0 and stop_game == False:
            b[r][c].configure(text='X', fg = 'blue', bg = 'white')
            states[r][c] = 'X'
            player = 'O'

    if player == 'O' and states[x][y] == 0 and stop_game == False:
            b[r][c].configure(text='O', fg = 'orange', bg = 'black')
            states[r][c] = 'O'
            player = 'X'
    check_for_winner()

def check_for_winner():
    global stop_game

    for i in range(3):
            if states[i][0] == states[i][1] == states[i][2] !=0:
```

```
                b[i][0].configure(bg='grey')
                b[i][1].configure(bg='grey')
                b[i][2].configure(bg='grey')
                stop_game = True

    for i in range(3):
        if states[0][i] == states[1][i] == states[2][1] !=0:
                b[0][i].configure(bg='grey')
                b[1][i].configure(bg='grey')
                b[2][i].configure(bg='grey')
                stop_game = True

     states[0][0] == states[1][1] == states[2][2] !=0:
            b[0][0].configure(bg='grey')
            b[1][1].configure(bg='grey')
            b[2][2].configure(bg='grey')
            stop_game = True

     states[2][0] == states[1][1] == states[0][2] !=0:
            b[2][0].configure(bg='grey')
            b[1][1].configure(bg='grey')
            b[0][2].configure(bg='grey')
            stop_game = True

root = Tk()

b = [ [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0] ]

states = [ [0, 0, 0],
           [0, 0, 0],
           [0, 0, 0] ]

for  i  in  range(3):
    for  j  in  range(3):
        b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                    command = lambda r=i, c=j: callback(r,c))
        b[i][j].grid(row = i, column = j)

player = 'X'
stop_game = False

mainloop()
```

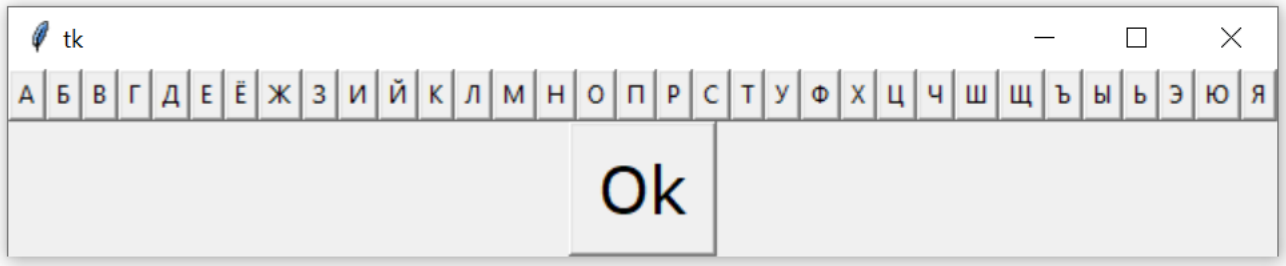
Глава 16

GUI программирование II

В этом разделе мы рассмотрим больше базовых GUI идей.

16.1 Контейнеры(Frames)

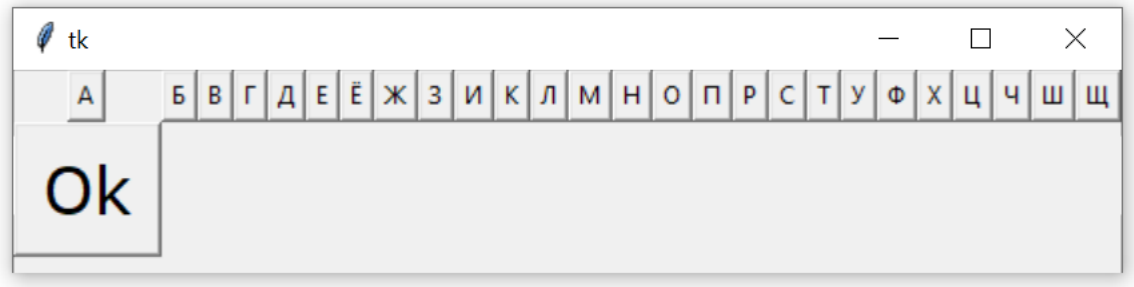
Допустим, мы хотим 33 маленьких кнопок вверху экрана и большую кнопку Ok под ними, как внизу:



Мы попробуем следующий код:

```
from tkinter import *

root = Tk()

alphabet = 'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ'
buttons = [0]*33
for i in range(33):
    buttons[i] = Button(text = alphabet[i])
    buttons[i].grid(row=0, column=i)

ok_button = Button(text='Ok', font=('Verdana', 24))
ok_button.grid(row=1, column=0)

mainloop()
```

Но вместо этого мы получим следующий, к сожалению, результат:



Проблема существует в столбце 0. В нем имеется два виджета, кнопка с буквой А и кнопка Ок, и Tkinter будет делать тот столбец достаточно большим, чтобы разместить большой виджет, кнопку Ок. Одно из решений этой проблемы показано ниже:

```
ok_button.grid(row=1, column=0, columnspan=33)
```

Другое решение этой проблемы является использование то, что называется *frame(контейнер)*. Работа *frame* заключается в удержании других виджетов и особенно в объединении их в один большой виджет. В этом случае, создадим *frame*, чтобы сгруппировать все кнопки букв в один большой виджет. Код показан ниже:

```
from tkinter import *

alphabet = 'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ'
root = Tk()

button_frame = Frame()
buttons = [0]*33
for i in range(33):
    buttons[i] = Button(button_frame, text = alphabet[i])
    buttons[i].grid(row=0, column=i)

ok_button = Button(text='Ok', font=('Verdana', 24))

button_frame.grid(row=0, column=0)
ok_button.grid(row=1, column=0)

mainloop()
```

Чтобы создать фрейм, мы используем *Frame()* и даем ему имя. Далее, для любых виджетов, которые мы хотим включить во фрейм, мы вставляем имя фрейма, как первый аргумент в объявлении виджета. Мы до сих пор должны компоновать виджеты, но сейчас строки и столбцы будут относится к фрейму. Наконец, нам нужно скомпоновать сам фрейм.

16.2 Цвета

Tkinter определяет множество распространенных названий цветов, как 'yellow' и 'red'. Он также предоставляет способ получения доступа к миллионам других цветов. Сначала нам нужно понять как цвета отображаются на экране.

Каждый цвет раскладывается на три компонента - красный, зеленый и синий. Каждый компонент может иметь значение от 0 до 255, с 255 образуется полный объем того цвета. Равные части красного и зеленого создают оттенки желтого, равные части красного и синего создают оттенки

пурпурного, а равные части синего и зеленого создают оттенки бирюзового. Равные части всех трех создают оттенки серого. Черный - это когда все три компонента имеют значение 0, а белый, когда все три компонента имеют значение 255. Изменяя значения компонентов можно произвести до $256^3 \approx 16$ миллионов цветов. Существует ряд ресурсов в интернете, которые позволяют Вам изменять количество компонентов и видеть, какой цвет получается.

Использовать цвета в Tkinter легко, но с одним нюансом - значения компонентов даются в шестнадцатеричной системе счисления. Шестнадцатеричная система счисления основана на системе с 16 числами, где буквы A-F используются для представления чисел от 10 до 15. Она широко применялась на раннем этапе компьютеризации, и она до сих пор используется повсеместно. Вот таблица сравнивающая две числовые системы:

0	0	8	8	16	10	80	50
1	1	9	9	17	11	100	64
2	2	10	A	18	12	128	80
3	3	11	B	31	1F	160	A0
4	4	12	C	32	20	200	C8
5	5	13	D	33	21	254	FE
6	6	14	E	48	30	255	FF
7	7	15	F	64	40	256	100

Из-за того, что значения компонентов цвета идут от 0 до 255, в шестнадцатиричной системе они будут идти от 0 до FF, и таким образом описываются двумя шестнадцатиричными цифрами. Обычный цвет в Tkinter определен, как: '#A202FF'. Наименование цвета предваряется знаком решетки. Далее первые две цифры - это красный компонент(в этом случае A2, что равняется 162 в десятичной системе). Следующие две цифры определяют зеленый компонент(здесь 02, в десятичной 2), и последние две цифры отвечают за синий компонент(здесь FF, в десятичной 255). Этот цвет получается голубовато-фиолетовым. Вот пример его использования:

```
label = Label(text='Привет', bg='#A202FF')
```

Если Вы не захотите заморачиваться с шестнадцатиричными числами, то можете использовать следующую функцию, которая переводит проценты в шестнадцатиричную строку, которую Tkinter применяет.

```
def color_convert(r, g, b):
    return '# {  : 02x } {  : 02x} {  : 02x} '.format(int(r*2.55), int(g*2.55), int(b*2.55))
```

Вот пример того, как создать фоновой цвет, который имеет 100% красного компонента, 85% зеленого и 80% синего.

```
label = Label(text = 'Привет', bg=color_convert(100, 85, 80))
```

16.3 Изображения

Метки и кнопки (и другие виджеты) могут отображать изображения вместо текста.

Чтобы воспользоваться, изображение требует немного подготовительной работы. Сначала нам нужно создать объект *PhotoImage* и дать ему имя. Вот пример:

```
cheetah_image = PhotoImage(file='cheetah.gif')
```

Вот некоторые примеры вложения изображения в виджеты:

```
label = Label(image=cheetah_image)
button = Button(image=cheetah_image, command=cheetah_callback())
```

Вы можете применить метод *configure*, чтобы установить или изменить изображение:

```
label.configure(image=cheetah_image)
```

Типы файлов Одним из досадных ограничений Tkinter является то, что единственным распространенным файловым типом изображения, которое он может использовать является GIF. Если Вы захотите применять другие типы файлов, одним из решений является применение Python Imaging Library, которую мы будем рассматривать в разделе 18.2

16.4 Холсты

Холст - это виджет, на котором Вы можете рисовать объекты типа линий, кругов, прямоугольников. Вы также можете наносить текст, изображения и другие виджеты на него. Это очень универсальный виджет, однако здесь мы будем описывать основы.

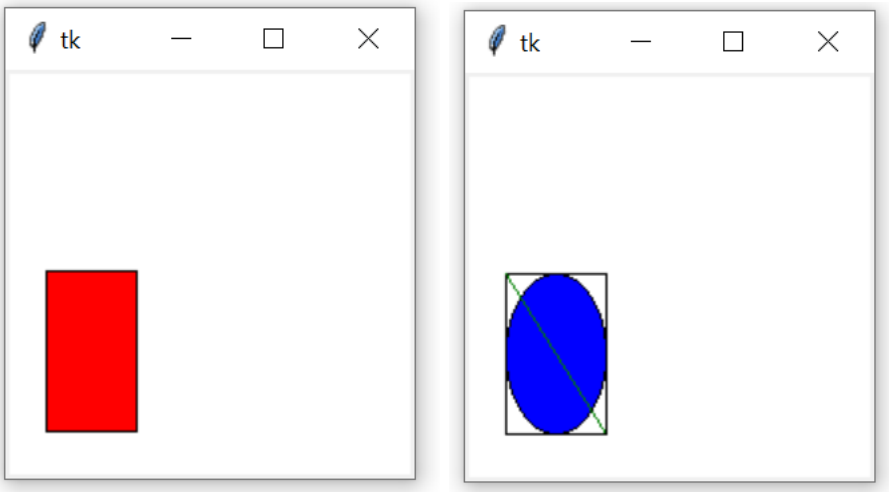
Создание холстов Следующая строка создает холст с белым фоном, размером 200 x 200 пикселей.

```
canvas = Canvas(width=200, height=200, bg='white')
```

Прямоугольники Следующий код рисует красный прямоугольник на холсте:

```
canvas.create_rectangle(20, 100, 65, 180, fill='red')
```

Посмотрите на изображение внизу слева. Первые четыре аргумента определяют координаты, где расположится прямоугольник на холсте. Верхний левый угол холста - начало(0, 0). Верхний левый прямоугольника равен (20, 100), а нижний правый равен (30, 150). Если Вы исключите *fill*='red', результатом был бы прямоугольник с черным контуром.



Овалы и линии Рисование овалов и линий подобно. Изображение выше справа создано следующем кодом:

```
canvas.create_rectangle(20, 100, 70, 180)
canvas.create_oval(20, 100, 70, 180, fill='blue')
canvas.create_line(20, 100, 70, 180, fill='green')
```

Прямоугольник здесь для того, чтобы показать, что линии и овалы работают подобно прямоугольникам. Первые две координаты являются левым верхним углом, а вторые две правым нижним.

Чтобы получить круг с радиусом r и центром x, y, мы можем создать следующую функцию:

```
def create_circle(x, y, r):
    canvas.create_oval(x-r, y-r, x+r, y+r)
```

Изображения Мы можем добавить изображения на холст. Вот пример:

```
cheetah_image = PhotoImage(file='cheetah.gif')
canvas.create_image(50,50, image=cheetah_image)
```

Две координаты- это, где должен быть центр изображения.

Обозначение объектов, их изменение, перемещение и удаление Мы можем давать имена объектам, когда помещаем их на холст. Мы можем, затем, использовать имена, чтобы ссылаться на объект в случае, когда хотим переместить его или удалить с холста. Вот пример, где мы создаем прямоугольник, изменяем его, перемещаем, а затем удаляем его:

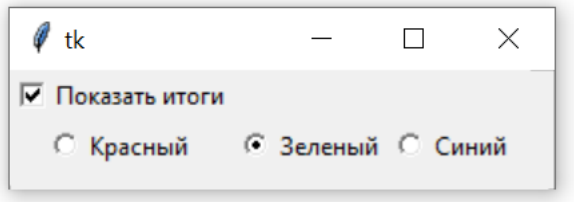
```
rect = canvas.create_rectangle(0, 0, 20, 20)
canvas.itemconfigure(rect, fill='red')
canvas.coords(rect,40,40,60,60) canvas.delete(rect)
```

Метод *coords* используется, чтобы перемещать или изменять размеры объекта, а метод *delete* применяется для его удаления. Если Вы хотите удалить все с холста, используйте следующее:

```
canvas.delete(ALL)
```

16.5 Флажки и переключатели

На изображении внизу, верхняя строка показывает флажок, а нижняя строка переключатель.



Переключатели Код для флажка сверху:

```
show_totals = IntVar()
check = Checkbutton(text='Показать итоги', var=show_totals)
```

Одна вещь, на которую нужно обратить внимание здесь - это то, что мы должны привязать флажок к переменной и она не может быть просто какой-либо переменной, она должна быть специальным типом Tkinter переменной, называемой *IntVar*. Эта переменная *show_totals*,будет равна 0, когда флажок не отмечен, и 1, когда отмечен. Чтобы получить доступ к значению переменной, Вам нужно использовать его метод *get*, как:

```
show_totals.get()
```

Вы можете также установить значение переменной используя его метод *set*. Это будет автоматически отмечать или не отмечать флажок на экране. Например, если Вы захотите, чтобы флажок сверху отмечался при запуске программы, сделайте следующее:

```
show_totals = IntVar()
show_totals.set(1)
check = Checkbutton(text='Показать итоги', var=show_totals)
```

Переключатели Переключатели работают аналогично. Код для переключателя, который показан в начале раздела следующий:

```
color = IntVar()
redbutton = Radiobutton(text='Красный', var=color, value=1)
greenbutton = Radiobutton(text='Зеленый', var=color, value=2)
bluebutton = Radiobutton(text='Синий', var=color, value=3)
```

Значение *IntVar* объекта *color* будет 1, 2 или 3, в зависимости от того, выбрана ли левый, средний или правый переключатель. Эти значения контролируются опцией *value*, определяемой когда мы создаем переключатели.

Команды И флажки и переключатели имеют опцию *command*, в которой мы можем установить функцию обратного вызова, чтобы запустить что-либо, когда кнопка отмечена или нет.

16.6 Text виджет

Виджет *text* - это большой, более мощный вариант виджета *Entry*. Вот пример созданного виджета:

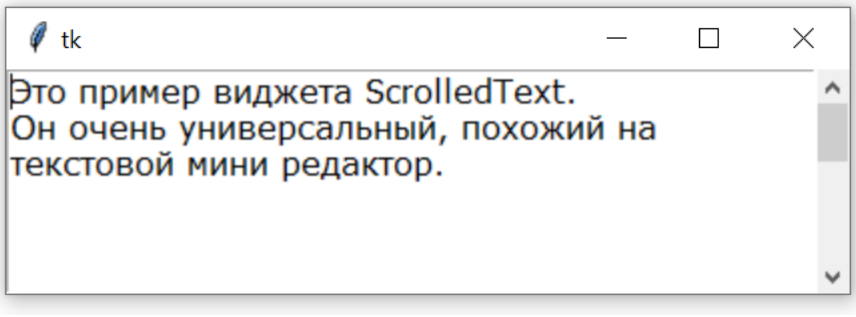
```
textbox = Text(font=('Verdana', 16), height=6, width=40)
```

Виджет будет 40 символов в ширину и 6 строк в высоту. Вы можете вводить еще дальше шестой строки; виджет будет только отображать лишь шесть строк сразу и Вы можете использовать кнопки со стрелками для прокрутки.

Если Вы захотите полосу прокрутки привязанную к текстовому полю, то можете использовать виджет *ScrolledText*. За исключением полосы прокрутки, *ScrolledText* работает более или менее так же как *Text*. Пример того, как это выглядит показан ниже. Чтобы использовать виджет *ScrolledText*, Вам будет нужен следующий импорт:

```
from tkinter.scrolledtext import ScrolledText
```

```
from tkinter import *
from tkinter.scrolledtext import ScrolledText

root = Tk()

scrolledtext = ScrolledText(font=('Verdana', 16), height=6, width=40)
scrolledtext.grid()
mainloop()
```



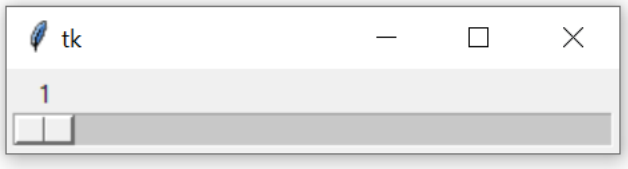
Ниже представлены несколько обычных команд:

Инструкция	Описание
textbox.get(1.0, END)	возвращает содержание текстового поля
textbox.delete(1.0, END)	удаляет все в текстовом поле
textbox.insert(END, 'Привет')	вставляет текст в конец текстового поля

Одной хорошей опцией, когда объявляется виджет *Text*, является *undo*=**True**,которая позволяет Ctrl+Z и Ctrl+Y отменять и повторять изменения. Существует множество других вещей, которые Вы можете делать с виджетом *Text*. Он почти похож на миниатюрный текстовой процессор.

16.7 Scale виджет

Scale - это виджет, который Вы можете двигать туда-сюда, чтобы выбрать различные значения. Пример показан ниже сопровождаемый кодом, который его создает.




```
scale = Scale(from_=1, to_=100, length=300, orient='horizontal')
```

Ниже следуют некоторые полезные опции виджета *Scale*:

Опция	Описание
from_	минимальное значение, возможное перемещением шкалы
to_	максимальное значение, возможное перемещением шкалы
length	длина шкалы в пикселях
label	определяет ярлык для шкалы
showvalue	отказывается от числа, которое отображается над шкалой
tickinterval=1	отображает галочки над каждом значении(1 может быть изменена)

Существуют несколько способов для Вашей программы взаимодействовать со шкалой. Одним из способов является связывание ее с *IntVar*, точно также как с флажками и переключателями, используя опцию *variable*. Другой способ заключается в использовании методов виджета *Scale get* и *set*. Третий способ - это использовать опцию *command*, которая работает точно также, как и с кнопками.

16.8 GUI события

Часто нам понадобится, чтобы наша программа делала что-то, когда пользователь нажимает определенную кнопку, перетаскивает что-то на холсте, использует колесо мыши и так далее. Эти вещи называются *events*(событиями).

Простой пример Первую GUI программу, которую мы видели в разделе 15.1, была простым температурным преобразователем. Всегда, когда мы хотели конвертировать температуру, мы обычно вводили её в поле ввода и щелкали на кнопке Вычислить. Было бы хорошо, если пользователь мог бы просто нажимать кнопку Enter, после того как он ввел температуру вместо щелкания по кнопке Вычислить. Мы можем достигнуть этого добавлением одной строки в программу:

```
entry.bind('<Return>', lambda dummy=0:calculate())
```

Эта строка должна следовать сразу, после того как Вы объявили поле ввода. Что она делает, то что принимает событие, когда кнопка Enter(Return) нажимается и привязывает это к функции *calculate*.

Как то так.Функция, которую Вы привязываете к событию, предполагается быть способной получить копию объекта *Event*, но функция для вычисления, которую мы ранее написали, не принимает аргументов. Вместо того, чтобы переписывать функцию, строка выше использует прием с lambda, по сути отбрасывая объект *Event*.

Распространенные события Ниже список некоторых распространенных событий:

Событие	Описание
<Button-1>	Нажатие на левую кнопку мыши
<Double-Button-1>	Двойное нажатие на левую кнопку мыши
<Button-Release-1	Левая кнопка мыши отпускается
<B1-Motion>	Нажатие и перетаскивание левой кнопкой мыши
<MouseWheel>	Передвижение колеса мыши
<Motion>	Перемещение мыши
<Enter>	Мышь находится сейчас над областью виджета
<Leave>	Мышь сейчас покинула область виджета
<Key>	Нажатие клавиши
<key name>	*Имя кнопки* нажатие клавиши

Для всех примеров с кнопкой мыши, число 1 может быть заменено другими числами. Кнопка 2 - это средняя кнопка, а кнопка 3 - правая кнопка.

Самыми полезными аттрибутами в объекте *Event* являются следующие:

Аттрибут	Описание
keysym	Имя клавиши, которая была нажата
x, y	Координаты указателя мыши
delta	Значение колеса мыши

События нажатий на клавиши Для событий нажатия на клавиши, Вы можете либо иметь определенные обратные вызовы для различных клавиш либо перехватывать все нажатия клавиш и обрабатывать их в одном и том же обратном вызове. Ниже следует пример последнего:

```
from tkinter import *

def callback(event):
    print (event.keysym)

root = Tk()
root.bind('<Key>', callback)
mainloop()
```

Программа выше выводит значения клавиш, которые были нажаты. Вы можете использовать эти значения в инструкциях if для обработки нескольких различных нажатий клавиш в функции обратного вызова, как показано ниже:

```
if event.keysym == 'percent':
    # percent(shift+5) был нажат, делать что-нибудь с этим...
elif event.keysym == 'a':
    # прописная была нажата, делать что-нибудь с этим...
```

Используйте единый метод обратного вызова, если Вы перехватываете множество нажатий клавиш и делаете что-то подобное со всеми ними. С другой стороны, если Вы просто хотите перехватить пару определенных нажатий клавиш или если определенные клавиши имеют очень длинный и специфический обратный вызов, Вы можете перехватить нажатия клавиш отдельно, как внизу:

```
from tkinter import *

def callback1(event):
    print('Ва нажали клавишу Enter.')
def callback2(event):
    print('Ва нажали клавишу стрелка вверх.')

root = Tk()
root.bind('', callback1)
root.bind('', callback2)

mainloop()
```

Названия клавиш являются те же самыми, что и названия содержащиеся в аттрибуте keysym. Вы можете использовать предыдущую программу в этом разделе, чтобы найти названия всех клавиш. Ниже следуют наименования для ряда распространенных клавиш:

Tkinter название	Обычное название
<Return>	Клавиша Enter
<Tab>	Клавиша Tab
<Space>	Пробел
<F1>,. . . ,<F12>	F1,. . . ,F12
<Next>,<Prior>	Page up, Page down
<Up>,<Down>,<Left>,<Right>	Клавиши со стрелками
<Home>,<End>	Home, End
<Insert>,<Delete>	Insert, Delete
<Caps_Lock>,<Num_Lock>	Caps lock, Number lock
<Control_L>,<Control_R>	Левая и правая клавиши Control
<Alt_L>,<Alt_R>	Левая и правая клавиши Alt
<Shift_L>,<Shift_R>	Левая и правая клавиши Shift

Большинство доступных клавиш для печати могут быть взяты с их именами, как представлено внизу:

```
root.bind('a', callback)
root.bind('A', callback)
root.bind('-', callback)
```

Исключениями являются пробел (<Space>) и знак меньше чем (<Less). Вы также можете перехватывать комбинации клавиш таких как, <Shift-F5>, <Control-Next>, <Alt-2> или <Control-Shift-F1>.

Обратите внимание Все эти примеры связызывают нажатия клавиш к *root*, которое является нашим именем для основного окна. Вы можете также привязать нажатия клавиш к определенным виджитам. Например, если Вам потребуется только клавиша левая стрелка, чтобы работать на Холсте, называемом *canvas*, то могли использовать следующее:

```
canvas.bind(<Left>, callback)
```

Одна хитрость здесь, однако, заключается в том, что холст не будет распознавать нажатия клавиш до тех пор, пока он не будет в фокусе GUI. Это может быть сделано, как показано ниже:

```
canvas.focus_set()
```

16.9 Примеры событий

Пример 1 Ниже представлен пример, в котором пользователь может двигать прямоугольник клавишами - левой или правой стрелкой.

```
from tkinter import *//
def callback(event):
    global move
    if event.keysym=='Вправо':
```

```
        move += 1
    elif event.keysym=='Влево':
        move -= 1
    canvas.coords(rect, 50+move, 50, 100+move, 100)

root = Tk() root.bind('<Key>', callback)
canvas = Canvas(width=200, height=200)
canvas.grid(row=0, column=0)
rect = canvas.create_rectangle(50, 50, 100, 100, fill='blue')
move = 0

mainloop()
```

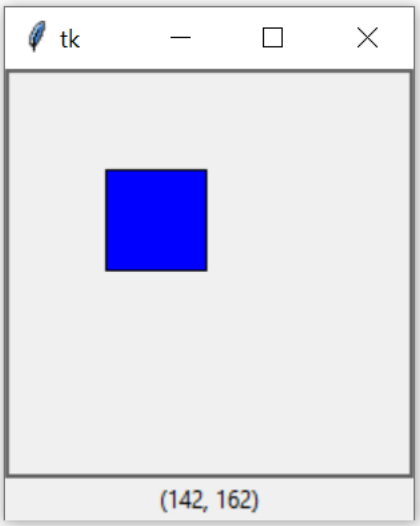
Пример 2 Ниже следует пример программы, которая представляет события от нажатия мыши. Программа запускается с рисования прямоугольника на экране. Пользователь может сделать следующее:

- Перемещать прямоугольник мышкой(<B1_Motion>).
- Изменять размер прямоугольника колесом мышки(<MouseWheel>).
- Всякий раз, когда пользователь нажимает левую кнопку, прямоугольник будет изменять цвета(<Button-1>).
- Всегда, когда мышка перемещается, текущие координаты мышки отображаются в метке(<Motion>

Далее представлен код программы:

```
from tkinter import *

def mouse_motion_event(event):
    label.configure(text='( { } , { } ) ' . format(event.x, event.y) )

def wheel_event(event):
    global x1, x2, y1, y2
    if event.delta>0:
        diff = 1
    elif event.delta<0:
        diff = -1
    x1 += diff
    x2 -= diff
    y1 += diff
    y2 -= diff
    canvas.coords(rect, x1, y1, x2, y2)

def b1_event(event):
    global color
    if not b1_drag:
        color = 'Red' if color == 'Blue' else 'Blue'
        canvas.itemconfigure(rect, fill=color)

def b1_motion_event(event):
```

```
    global b1_drag, x1, x2, y1, y2, mouse_x, mouse_y
    x = event.x
    y = event.y
    if not b1_drag:
        mouse_x = x
        mouse_y = y
        b1_drag = True
        return
    x1 += (x-mouse_x)
    x2 += (x-mouse_x)
    y1 += (y-mouse_y)
    y2 += (y-mouse_y)
    canvas.coords(rect, x1,y1,x2,y2)
    mouse_x = x
    mouse_y = y

def b1_release_event(event):
    global b1_drag
    b1_drag = False

root = Tk()

label = Label()

canvas = Canvas(width=200, height=200)
canvas.bind('<Motion>', mouse_motion_event)
canvas.bind('<ButtunPress-1>', b1_event)
canvas.bind('<B1-Motion>', b1_motion_event)
canvas.bind('<ButtonRelease-1>', b1_release_event)
canvas.bind('<MouseWheel>', wheel_event)
canvas.focus_set()

canvas.grid(row=0, column=0)
label.grid(row=1, column=0)

mouse_x = 0
mouse_y = 0
b1_drag = False

x1 = y1 = 50
x2 = y2 = 100
color = 'blue'
rect = canvas.create_rectangle(x1,y1,x2,y2,fill=color)

mainloop()
```



Далее следует несколько замечаний о том, как программа работает:

1. Во первых, всякий раз, когда мышь перемещается в фокусе холста, вызывается функция *mouse_ motion_ event*. Эта функция выводит текущие координаты, которые содержатся в аттрибутах *x* и *y* объекта *Event*.

2. Функция *wheel_ event* вызывается всякий раз, когда пользователь использует колесо мыши(прокручивание). Аттрибут *delta* объекта *Event* содержит информацию о том, как быстро и в каком направлении колесо двигается. Мы просто растягиваем или сжимаем прямоугольник на основании - двигается колесо вперед или назад.

3.Функция *b1_ event* вызывается всегда, когда пользователь нажимает левую кнопку мыши. Функция изменяет цвет прямоугольника всякий раз, когда на прямоугольник щелкают(кликают). Здесь имеется глобальная переменная, называемая *b1_ drag*, которая имеет важное значение. Она устанавливается **True** всякий раз, когда пользователь тянет прямоугольник. Когда происходит перемещение, левая кнопка мыши нажата и функция *b1_ event* постоянно вызывается. Мы не хотим продолжат изменять цвет прямоугольника в этом случае, отсюда и оператор if.

4. Перемещение выполняется, в основном, в функции *b1_ motion_ event*, которая вызывается всегда, когда нажата левая кнопка мыши и мышь передвигается. Она использует глобальную переменную, которая отслеживает какое положение мыши было последний раз, когда функция вызывалась, а затем передвигает прямоугольник в соответствии между новым и старым положением. Когда перетаскивание завершится, левая кнопка мыши будет отпущена. Когда это происходит, то вызывается функция *b1_ release_ event* и мы устанавливаем глобальную переменную *b1_ drag* соответствующим образом.

5. Метод *focus_ set* необходим, потому что холст не будет распознаваться событиями при вращении колеса мыши, пока холст не будет в фокусе.

6. Одна проблема в этой программе заключается в том, что пользователь может изменять прямоугольник кликая в любом месте холста, но не на самом прямоугольнике. Если мы захотим, чтобы изменения происходили когда мышь над прямоугольником, то можем непосредственно привязать прямоугольник, вместо всего холста, как показано внизу:

```
canvas.tag_bind(rect, '<B1-Motion>', b1_motion_event)
```

7.Наконец, использование здесь глобальных переменных приводит к небольшому беспорядку. Если это было бы частью большого проекта, то могло иметь смысл объединить все это в класс.

Глава 17

GUI программирование III

Эта глава содержит немного больше всяких мелочей GUI.

17.1 Панель заголовка

Окно GUI, которое Tkinter создает, указано Tk по умолчанию. Ниже показано как его изменить:

```
root.title('Ваш заголовок')
```

17.2 Ограничение объектов

Иногда Вам потребуется деактивировать кнопку так, чтобы она не могла быть нажата. Кнопки имеют аттрибут *state*, который позволяет Вам деактивировать виджет. Используйте *state = DISABLED*, чтобы деактивировать и *state = NORMAL*, чтобы задействовать кнопку:

```
button = Button(text='Привет', state=DISABLED, command=function)
button.configure(state=NORMAL)
```

Вы можете использовать аттрибут *state*, чтобы деактивировать, также, много других типов виджетов.

17.3 Контроль состояния виджетов

Иногда, Вам потребуется знать что-то про виджет, типа какой точно текст содержится в нем или какой цвет фона. Метод *cget* применяется для этого. Например, нижеследующее принимает текст метки, которая называется *label*:

```
label.cget('text')
```

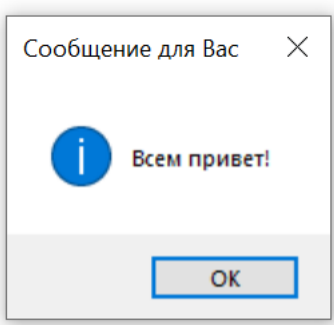
Метод может быть использован с кнопками, холстами и так далее, и он может быть применен к любому из его свойств, как *bg*, *fg*, *state* и так далее. Как сокращение, Tkinter перегружает операторы [ ], так что *label*['text'] выполняет то же самое, что и пример выше.

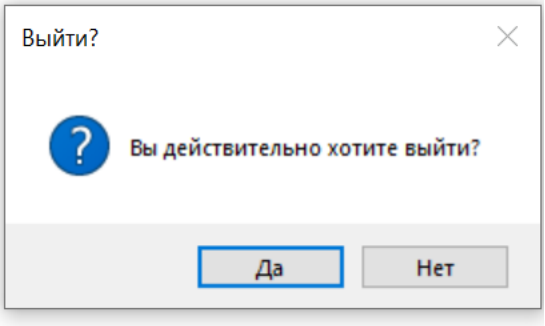
17.4 Диалоговые окна

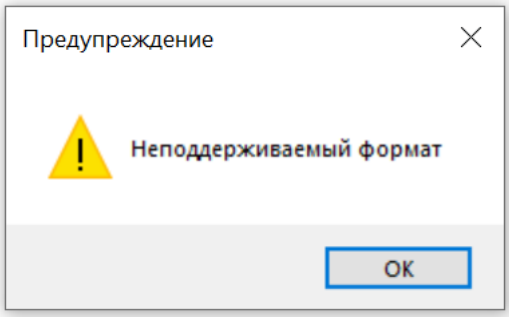
Диалоговые окна - это окна, которые всплывают, чтобы задать Вам вопрос или спросить что-то, а потом исчезают. Чтобы применять их, нам понадобится оператор import:

```
from tkinter.messagebox import *
```

Существует множество различных типов диалоговых окон. Для каждого из них Вы можете определить сообщение, которое пользователь увидит, а также заголовок диалогового окна. Ниже представлены три типа диалоговых окон с нижеследующим кодом, который их создает:








```
showinfo(title='Сообщение для Вас', message='Всем привет!')
askquestion(title='Выйти?', message='Вы действительно хотите выйти?')
showwarning(title='Предупреждение', message='Неподдерживаемый формат')
```

Ниже представлен список всех типов диалоговых окон. Каждое отображает сообщение по-своему.

Диалоговое окно	Особые свойства
showinfo	*Ok* кнопка
askokcancel	Кнопки *Ок* и *Cancel*
askquestion	Кнопки *Yes* и *No*
askretrycancel	Кнопки *Retry* и *Cancel*
askyesnocancel	Кнопки *Yes*, *No* и *Cancel*
showerror	Значок ошибки и кнопка *Ok*
showwarning	Значок предупреждения и кнопка *Ok*

Каждая из этих функций возвращает значение, указывающее на что нажал пользователь. Смотри следующий раздел для простого примера использования возвращаемого значения. Ниже представлена таблица возвращаемых значений:

Функция	Возвращаемое значение(основанное на что пользователь нажимает)
showinfo	Всегда возвращает 'ok'
askokcancel	*Ок*— **True** *Cancel* или окно закрылось— **False**
askquestion	Кнопки *Yes* — 'yes' *No* — 'no'
askretrycancel	*Retry* — **True** *Cancel* или окно закрылось — **False**
askyesnocancel	*Yes* — **True** *No* — **False** что-нибудь еще — **None**
showerror	Всегда возвращает 'ok'
showwarning	Всегда возвращает 'ok'

17.5 Удаление объектов

Чтобы удалить виджет, используйте его метод *destroy*. Например, чтобы удалить кнопку, называемую *button*, делайте следующее:

```
button.destroy()
```

Чтобы удалить GUI окно полностью, используйте следующее:

```
root.destroy()
```

Предотвращение закрытия окна Когда Ваш пользователь пытается закрыть главное окно, Вам может понадобится что-то сделать, например спросить его, действительно ли он хочет выйти. Внизу представлен способ сделать это:

```
from tkinter import *
from tkinter.messgebox import askquestion

def quitter_function():
    answer = askquestion(title='Выход?', message='Действительно выйти?')
    if answer == 'да':
        root.destroy()

root = Tk()
root.protocol('WM_DELETE_WINDOW', quitter_function)

mainloop()
```

Ключевой является следующая строка, которая приводит к вызову *quitter_function* всякий раз, когда пользователь пытается закрыть окно.

```
root.protocol('WM_DELETE_WINDOW', quitter_function)
```

17.6 Обновление

Tkinter обновляет экран время от времени, но иногда это происходит недостаточно часто. Например, в функции запускаемой нажатием кнопки, Tkinter не будет обновлять экран пока функция не выполнится. Если в той функции Вы захотите изменить что-то на экране, сделайте короткую паузу, а затем измените что-то еще, Вам нужно будет сообщить Tkinter, чтобы он обновил экран перед паузой. Чтобы сделать то, просто используйте это:

```
root.update()
```

Если Вам нужно обновить только определенный виджет и ничего больше, Вы можете применить метод *update* на тот виджет. Например,

```
canvas.update()
```

Связанная с этим вещь, которая иногда бывает полезна, заключается в том, чтобы что-то произошло по истечении запланированного интервала времени. Например, у Вас может быть таймер в программе. Для этого, Вы можете использовать метод *after*. Его первый аргумент является

временем в миллисекундах для задержки перед обновлением, а второй аргумент - это функция, которая вызывается когда время истекает. Ниже следует пример, который реализует таймер:

```
from time import time
from tkinter import *

def update_timer():
    time_left = int(90 - (time()-start))
    minutes = time_left // 60
    seconds = time_left % 60
    time_label.configure(text=' { }  : {  :02d }  '.format(minutes, seconds))
    root.after(100, update_timer)

root = Tk()
time_label = Label()
time_label.grid(row=0, column=0)

start = time()
update_timer()

mainloop()
```

Этот пример использует модуль *time*, который рассматривается в разделе 20.2.

17.7 Диалоги

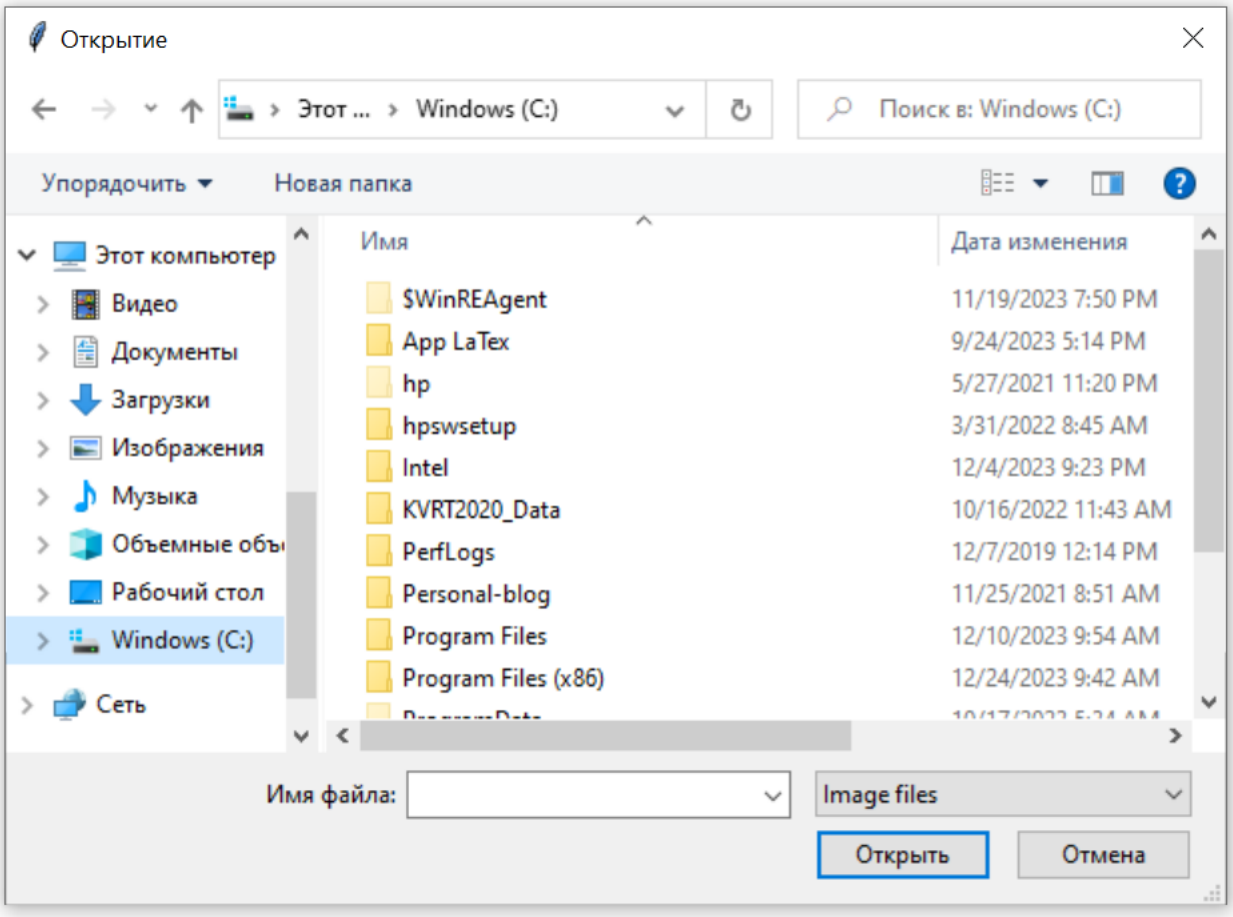
Многие программы имеют диалоговые окна, которые позволяют пользователю выбирать файл, чтобы открыть или сохранить его. Чтобы применить их в Tkinter, нам понадобится следующая инструкция импортирования:

```
from tkinter.filedialog import *
```

Диалоговые окна Tkinter обычно выглядят аналогично тем, которые встроены в операционную систему.

Ниже представлены самые полезные диалоговые окна:

Диалоговое окно	Описание
askopenfilename	Открывает обычное окно выбора файла
askopenfilenames	Аналогично предыдущему, но с выбором больше одного файла
asksaveasfilename	Открывает обычное окно для сохранения файла
askdirectory	Открывает окно выбора папки



Возвращаемыми значениями *askopenfilename* и *asksaveasfilename* являются имена выбранных файлов. Возвращаемые значения отсутствуют, если пользователь не выбрал наименование. Возвращаемое значение *askopenfilenames* - это список файлов, который будет пустым если файлы не выбраны. Функция *askdirectory* возвращает имя выбранной папки.

Существуют некоторые опции, которые Вы можете передать в эти функции. Вы можете установить значение *initialdir* равное наименованию папки, с которой Вам нужно, чтобы запускалось диалоговое окно. Вы также можете определить типы файлов. Далее пример:

```
filename = askopenfilename(initialdir='c:\\ python31\\'
                           filetypes =[('Image files', '.jpg .png .gif'),
                                       ('All files', ' * ')])
```

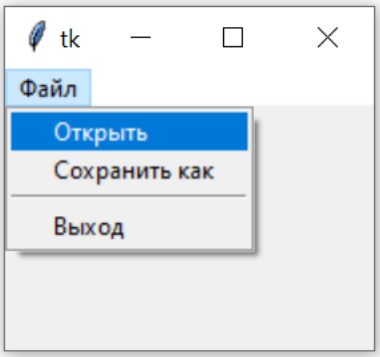
Простой пример Ниже представлен пример, который открывает файловое диалоговое окно, позволяющее Вам выбрать текстовой файл. Затем программа отображает содержимое файла в текстовом окне.

```
from tkinter import *
from tkinter.filedialog import *
from tkinter.scrolledtext import ScrolledText

root = Tk()
textbox = ScrolledText()
textbox.grid()
filename = askopenfilename(initialdir='c:\\ python31\\'
                           filetypes =[('Image files', '.jpg .png .gif'),
                                       ('All files', ' * ')])
```

```
s = open(filename).read()
textbox.insert(1.0, s)

mainloop()
```

17.8 Панели меню

Мы можем создать панель меню, как например внизу, сверху окна.



Далее пример, который использует некоторые диалоговые окна из предыдущего раздела:

```
from tkinter import *
from tkinter.filedialog import *

def open_callback():
    filename = askopenfilename()
    # сюда добавить код, который что-то делает с именем файла

def saveas_callback():
    filename = asksaveasfilename()
    # сюда добавить код, который что-то делает с именем файла

root = Tk()
menu = Menu()
root.config(menu=menu)
file_menu = Menu(menu, tearoff=0)
file_menu.add_command(label='Открыть', command=open_callback)
file_menu.add_command(label='Сохранить как', command=saveas_callback)
file_menu.add_separator()
file_menu.add_command(label='Выход', command=root.destroy)
menu.add_cascade(label='Файл', menu=file_menu)

mainloop()
```

17.9 Новые окна

Создать новое окно очень просто. Используйте функцию *Toplevel*:

```
window = Toplevel()
```

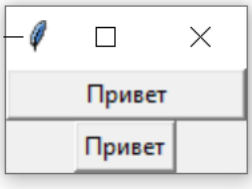
Вы можете добавить виджеты к новому окну. Первым аргументом, когда Вы создаете виджет, должно быть имя окна, как ниже:

```
new_window = Toplevel()
label = Label(new_window, text = 'Привет')
label.grid(row=0, column=0)
```

17.10 Метод pack

Существует альтернатива методу *grid*, называемый *pack*. Он не такой универсальный как *grid*, но имеются некоторые места где он полезен. Метод использует аргумент, называемый *side*, который позволяет Вам определить четыре расположения для Ваших виджетов: *TOP*, *BOTTOM*, *LEFT* и *RIGHT*. Имеются два полезных опциональных аргумента - *fill* и *expand*. Ниже пример.

```
button1=Button(text='Привет')
button1.pack(side=TOP, fill=X)
button2=Button(text='Привет')
button2.pack(side=BOTTOM)
```



Опция *fill* приводит к тому, что виджет заполняет все доступное ему пространство. Это может быть или *X*, *Y* или *BOTH*. Опция *expand* применяется, чтобы позволить виджету расширяться, когда его окно изменяет размеры. Чтобы активировать его используйте *expand=YES*.

Обратите внимание Вы можете использовать *pack* для одних фреймов, *grid* для других; просто не смешивайте *pack* и *grid* в одном фрейме, или Tkinter совсем не будет знать, что делать.

17.11 StringVar

В разделе 16.5 мы видели как связывать переменную Tkinter, называемую *IntVar*, чтобы проверить кнопку или флажок. В Tkinter имеется другой тип переменной, называемой *StringVar*, которая содержит строки. Этот тип переменной может быть использован, чтобы изменить текст в метке или кнопке или в некоторых других виджетах. Мы уже знаем как изменить текст используя метод *configure*, а *StringVar* предоставляет другой способ это сделать.

Чтобы привязать виджет к *StringVar* примените опцию виджета *textvariable*. *StringVar* содержит методы *get* и *set*, также как *IntVar* и всякий раз, когда Вы устанавливаете переменную, любой виджет, который привязан к ней, автоматически обновится.

Вот простой пример, который связывает две метки к одной и той же *StringVar*. Также имеется кнопка, которая при нажатии на нее будет чередовать значение *StringVar* (и следовательно текст в метках).

```
from tkinter import *

def callback():
    global count
    s.set('Пока' if count%2==0 else 'Привет ')
    count +=1

root = Tk()

count = 0
s = StringVar()
s.set('Привет ')

label1 = Label(textvariable = s, width=10)
label2 = Label(textvariable = s, width=10)
button = Button(text = 'Нажми меня ', command = callback)

label1.grid(row=0, column=0)
label2.grid(row=0, column=1)
button.grid(row=1, column=0)

nainloop()
```

17.12 Больше с GUI

Мы пропустили довольно многое о Tkinter. Смотри Lundh's *Introduction to Tkinter* [2], чтобы узнать больше. Tkinter является универсальным и простым для работы с ним, но если Вам потребуется что-то более серьезнее, то существуют другие сторонние GUI для Python.

Глава 18

Дальнейшее графическое программирование

18.1 Python 2 против Python 3

На момент написания, самая последней версией Python являлась 3.2, и весь код в этой книге разработан для запуска в Python 3.2. Сложность заключается в том, что с версии 3.0 Python нарушил совместимость со старыми версиями Python. Код написанный в этих старых версиях не всегда будет работать в Python 3.Проблема с этим заключается в том, имелся ряд полезных библиотек, написанных для Python 2, которые на момент написания не были еще перенесены в Python 3. Мы хотим использовать эти библиотеки, поэтому мы будем должны узнать немного о Python 2. К счастью, существуют только несколько больших отличий о которых не стоит беспокоиться

Деление Оператор деления, /, в Python 2, когда используется с целыми числами ведет себя как //. Например, 5/4 в Python 2 определяется в 1, в то время как 5/4 в Python 3 определяется в 1.2. Именно таким образом оператор деления ведет себя в ряде других языков программирования. В Python 3 было сделано решение сделать поведение оператора деления таким же, каким мы пользуемся в математике.

В Python 2, если Вы желаете получить 1.25 делением 5 на 4, то Вам необходимо сделать 5/4.0. По крайней мере один из аргументов должен быть числом с плавающей точкой, для того чтобы результат был числом с плавающей точкой. Если Вы делите две переменные, тогда вместо x/y, Вам возможно нужно сделать $x/\mathbf{float}(y)$.

print Функция **print** в Python 3 на самом деле была инструкцией **print** в Python 2. Поэтому в Python 2 Вы бы написали:

```
print 'Привет'
```

без каких-либо круглых скобок. Этот код больше не будет работать в Python 3 потому, что инструкция **print** сейчас является функцией **print**, а функции требуются круглые скобки. Также, текущая функция **print** имеет эти полезные опциональные аргументы *sep* и *end*, которые недоступны в Python 2.

input Python 2 эквивалентом функции **input** Python 2 является **raw_input**.

range Функция **range** может быть неэффективной с очень большим диапазоном в Python 2. Причина заключается в том, что в Python 2, если Вы используете **range**(10000000), Python будет создавать список из 10 миллионов чисел. Инструкция **range** в Python 3 более эффективная и вместо того, чтобы создавать все 10 миллионов объектов сразу, он создает их по мере необходимости. Функцией Python 2, которая действует как **range** Python 3, является **xrange**.

Форматирование строк Форматирование строк в Python 2 немного другое чем в Python 3. При использовании кодов форматирования внутри фигурных скобок в Python 2, Вам нужно определить номер аргумента. Сравните примеры ниже:

```
Python 2: 'x={ 0:3d}, y={ 1:3d}, z={ 2:3d}'.format(x, y, z)
Python 3: 'x={ :3d}, y={ :3d}, z={ :3d}'.format(x, y, z)
```

Что касается Python 3.1, назначение номера аргумента было сделано опционально.

Существует также старый стиль форматирования, который Вы можете видеть время от времени и который использует оператор %. Пример показан ниже рядом с соответствующим новым стилем.

```
Python 2:'x=%3d, y=%6.2f, z=%3s ' % (x, y, z)

Python 3: 'x={ :3d}, y={ :3d}, z={ :3d}'.format(x, y, z)
```

Имена модулей Некоторые модули были переименованы и реорганизованы. Ниже представлены несколько измененных имен Tkinter.

Python 2	Python 3
Tkinter	tkinter
ScrolledText	tkinter.scrolledtext
tkMessageBox	tkinter.messagebox
tkFileDialog	tkinter.filedialog

Существует ряд других модулей, мы увидим их позднее, которые были переименованы, большинство просто представлены в нижнем регистре. Например, *Queue* в Python 2, в Python 3 сейчас называется *queue*.

Генераторы словарей Генераторы словарей не представлены в Python 2.

Другие изменения Имеется довольно много других изменений в языке, но большинство из них с возможностями более расширенными, чем мы рассматриваем здесь.

Импорт будущего поведения Следующий импорт позволяет нам использовать поведение деления Python 3 в Python 2.

```
from __future__ import division
```

Имеются много других вещей, которые Вы можете импортировать из будующего.

18.2 Библиотека Python для работы с изображениями

The Python Imaging Library (PIL)(Библиотека Python для работы с растровыми изображениями) содержит полезные инструменты для работы с изображениями. На момент написания библиотека доступна для Python 2.7 и выше. Библиотека не является частью стандартного дистрибутива Python, поэтому Вам нужно будет загрузить и установить ее отдельно. Впрочем, она устанавливается легко.

PIL не поддерживается с 2009, но существует проект, называемый Pillow, который почти совместим с PIL и работает в Python 3 и позднее.

Мы рассмотрим просто несколько особенностей здесь. Хорошей ссылкой является The Python Imaging Library Handbook.

Использование изображений отличных от GIF с Tkinter Tkinter, как мы уже увидели, не может использовать JPEG и PNG. Но он может, если мы используем его в сочетании с PIL. Далее простой пример:

```
from Tkinter import *
from PIL import Image, ImageTk

root = Tk()
cheetah_image = ImageTk.PhotoImage(Image.open('cheetah.jpg'))

button = Button(image=cheetah_image)
button.grid(row=0, column=0)

mainloop()
```

Первая строка импортирует Tkinter. Вспомните, что в Python 2, название начинается с заглавной буквы - *Tkinter*. Следующая строка импортирует несколько объектов из PIL. Далее, там, где бы мы применили объект Tkinter *PhotoImage* для загрузки изображения, мы вместо этого используем комбинацию двух функций PIL. Мы можем, затем, использовать изображение как обычное в наших виджетах.

Изображения PIL - это Библиотека Python для работы с растровыми изображениями и поэтому содержит большое количество средств для работы с изображениями. Мы просто покажем простой пример здесь. Программа ниже отображает фото и когда пользователь щелкает на кнопку, изображение преобразуется в оттенки серого.

```
from Tkinter import *
from PIL import Image, ImageTk

def change():
    global image, photo
    pix = image.load()
    for i in range(photo.width()):
        for j in range(photo.height()):
            red, green, blue = pix[i, j]
            avg = (red+green+blue)//3
            pix[i, j] = (avg, avg, avg)      photo = ImageTk.PhotoImage(image)
canvas.create_image(0, 0, image=photo, anchor=NM)

def load_file(filename):
    global image, photo
    image=Image.open(filename).convert('RGB')
    photo=ImageTk.PhotoImage(image)
    canvas.configure(width=photo.width(), height=photo.height())
    canvas.create_image(0,0,image=photo,anchor=NW)
    root.title(filename)
```

```
root = Tk()
button = Button(text='Изменить', font=('Verdana', 18), command=change)
canvas = Canvas()
canvas.grid(row=0)
button.grid(row=1)
load_file('pic.jpg')

mainloop()
```

Давайте сначала взглянем на функцию *load_file*. Множество инструментов для изображений имеются в модуле Image. Мы даем имя, *image*, объекту созданному инстркуцией *Image.open*. Мы также применяем метод *convert*, чтобы преобразовать изображение в RGB(Красный-Зеленый-Синий) формат. Через минуту мы увидим, почему. Следующая строка создает объект *ImageTk*, называемый *photo*, который втягивается на холст Tkinter. Объект *photo* имеет методы, которые позволяют нам получать его ширину и высоту, поэтому мы можем настроить размер холста соответственно.

Сейчас взглянем на функцию *change*. У объекта *image* есть метод, называемый *load*, который предоставляет доступ к индивидуальным пикселям, составляющих изображение. Он возвращает двухмерный массив значений RGB. Например, если пиксель в верхнем левом углу изображения чисто белый, тогда *pix*[0,0] будет (255, 255, 255). Если следующий пиксель вправо будет чисто черным, то *pix*[1,0] будет (0, 0, 0). Чтобы преобразовать изображение в оттенки серого, мы принимаем, для каждого пикселя среднее значение его красного, зеленого и синего компонентов и сбрасываем красный, зеленый и синий компоненты так, чтобы они равнялись тому среднему значению. Вспомните, что если красный, зеленый и синий будут те же самыми, тогда цвет будет серого оттенка. После изменений всех пикселей, мы создадим новый объект *ImageTk* из измененных данных о пикселях и отобразим его на холсте.

Вы можете сильно с этим позабавиться. Попытайтесь изменить функцию change. Например, если мы применяем следующую строку в функции *change*, то получим эффект, который выглядит как фото негатив:

```
pix[i, j] = (255-red, 255-green, 255-blue)
```

Попробуйте посмотреть, какие интересные эффекты вы можете придумать.

Обратите внимание, однако, что этот способ обработки изображений является медленным, ручным методом. PIL имеет ряд функций, которые намного быстрее для обработки изображений. Вы можете очень легко изменить яркость, оттенок и контраст изображений, изменить их размер, вращать их и гораздо больше. Смотрите справочный материал по PIL, чтобы узнать больше.

putdata Если Вас интересует рисование математических объектов, например, фракталов, построение точек попиксельно, то в Python может быть очень медленно. Одним из способов ускорить процесс является применение метода *putdata*. Способом, с которым он работает заключается в том, что Вы обеспечиваете его списком RGB значений пикселей, и он будет копировать его в Ваше изображение. Ниже представлена программа, которая вычерчивает сетку размером 300 x 300 случайных цветов.

```
from random import randint
from Tkinter import *
from PIL import Image, ImageTk
```

```
root = Tk()
canvas = Canvas(width=300, height=300)
canvas.grid()
image=Image.new(mode='RGB',size=(300,300))

L = [(randint(0,255), randint(0,255), randint(0,255))
        for x in range(300) for y in range(300)]

image.putdata(L)

photo=ImageTk.PhotoImage(image)
canvas.create_image(0,0,image=photo,anchor=NW)
mainloop()
```

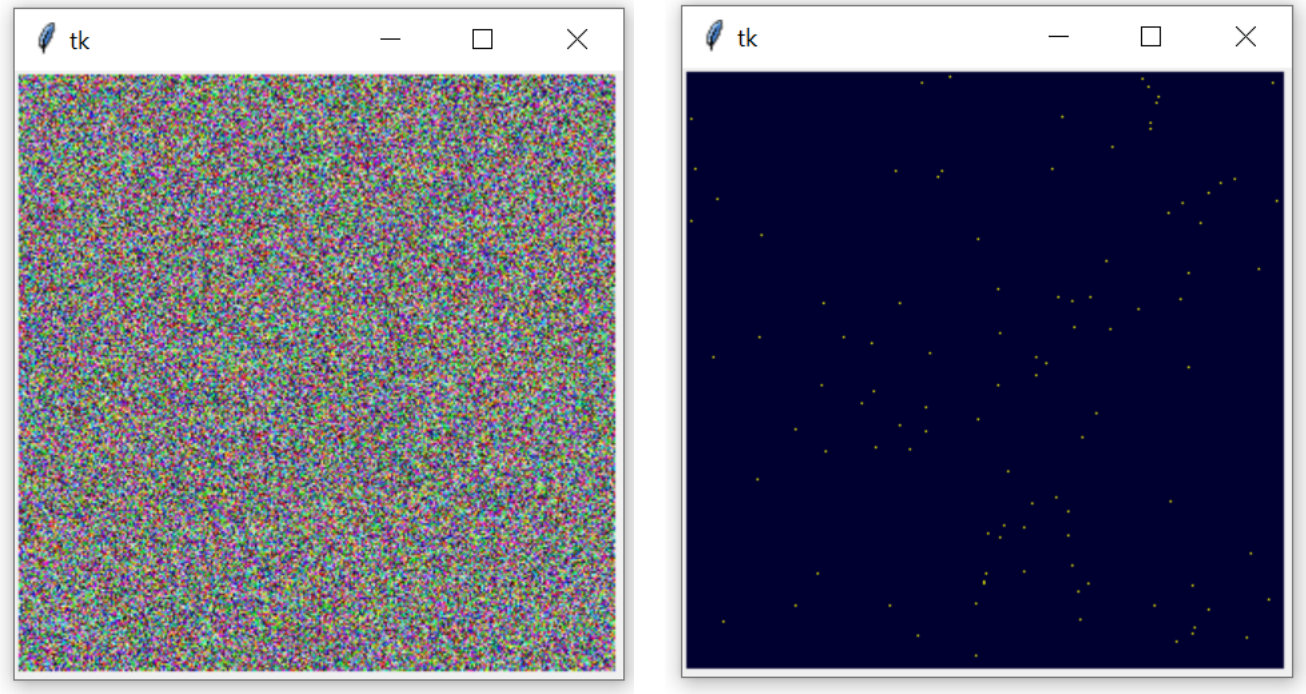


Рис 18.1(Слева) пример putdata (Справа) пример ImageDraw

ImageDraw Модуль *ImageDraw* предоставляет другой способ рисования на изображениях. Он может быть использован для рисования прямоугольников, кругов, точек и более, точно как холсты Tkinter, но быстрее. Далее короткий пример, который заполняет изображение темно-синим цветом, а затем случайным образом распределяет 100 желтых точек.

```
from random import randint
from Tkinter import *
from PIL import Image, ImageTk, ImageDraw

root = Tk()
canvas = Canvas(width=300, height=300)
canvas.grid()
image=Image.new(mode='RGB',size=(300,300))
draw = ImageDraw.Draw(image)

draw.rectangle([(0,0),(300, 300)],fill='#000030')
L = [(randint(0,299), randint(0, 299)) for i in range(100)]
```

```
draw.point(L, fill='yellow')

photo=ImageTk.PhotoImage(image) canvas.create_image(0,0,image=photo,anchor=NW)
mainloop()
```

Чтобы использовать *ImageDraw*, мы должны сначала создать объект *ImageDraw* и привязать его к объекту *Image*. Метод *draw.rectangle* работает подобно методу холстов *create_ rectangle*, за исключением некоторых различий со скобками. Метод *draw.point* используется для построения отдельных пикселей. Его хорошей особенностью является то, что мы можем передать список точек вместо того, чтобы вычерчивать каждый объект в списке отдельно. Передача списка также намного быстрее.

18.3 Pygame

Pygame - это библиотека для создания двухмерных игр в Python. Она может быть использована для возможности делать игры с уровнем старых аркад или Nintendo игр. Она может быть загружена и легко установлена из www.pygame.org.. Там имеется ряд руководств, чтобы помочь Вам начать. Я не знаю многого про Pygame, поэтому я не рассматриваю это здесь, хотя, возможно, в более позднем издании я это сделаю.

Часть III

Промежуточные темы(средний уровень)

Глава 19

Разнообразные темы III

В этой главе мы рассмотрим множество полезных тем.

19.1 Изменчивость и ссылки

Если *L* список и *s* строка, тогда *L[0]* дает первый элемент списка, а *s[0]* первый элемент строки. Если нам нужно изменить первый элемент списка на 3, то *L[0]*=3 сделает это. Но мы не сможем изменить строку таким способом. Причина заключается в том, как Python обрабатывает списки и строки. Считается, что списки (и словари) *изменяемые* - это означает, их содержание может изменяться. Строки, с другой стороны, являются *неизменяемыми*, они не могут быть изменены. Причина по которой строки являются неизменяемыми - частично для производительности (неизменяемые объекты быстрее) и частично потому, что строки считаются фундаментальными точно так же, как и числа. Это делает некоторые другие стороны языка проще также.

Создание копий Другим местом в котором у списков и строк имеются отличия - это когда мы пытаемся делать копии. Рассмотрим следующий код:

```
s = 'Привет'
copy = s
s = s + '!!!'
print('s сейчас', s, 'Копия:', copy)
```

```
s сейчас: Привет !!!        Копия: Привет
```

В коде выше, мы делаем копию *s*, а затем изменяем *s*. Все работает так, как мы интуитивно ожидаем. Сейчас взглянем на подобный код со списками:

```
L = [1, 2, 3]
copy = L
L[0] = 9
print('L сейчас:', L, 'Копия:', copy)
```

```
L сейчас: [9, 2, 3]        Копия: [9, 2, 3]
```

Мы можем видеть, код списка не работает, как мы могли ожидать. Когда мы изменили *L*, копия также получила изменения. Как упоминалось в главе 7, правильным способом сделать копию *L* является *copy*=L[:]. Ключом к пониманию этих примеров являются *ссылки*.

Ссылки Все в Python является объектами. Они включают - числа, строки и списки. Когда мы делаем простое присваивание переменной, например x = 487, на самом деле происходит следующее: Python создает целочисленный объект со значением 487, а переменная *x* действует как *ссылка* на тот объект. Это не то, что значение 487 хранится в ячейке памяти, называемой *x*, а скорее, что 487 хранится где-то в памяти, и *x* указывает на то местоположение. Если мы пойдем и объявим *y*=487, тогда *y* укажет на ту же самую ячейку памяти.

С другой стороны, если мы пойдем далее и скажем *x*=721, произойдет то, что мы создаем новый целочисленный объект со значением 721 где-то в памяти и *x* сейчас указывает на то.Число 487 до сих пор существует в памяти, где оно было и будет оставаться там, по крайней мере, пока не останется ничего, указывающего на него, в этот момент его ячейка памяти будет свободна для использования чего-либо ещё.

Все объекты обрабатываются тем же самым способом. Когда мы устанавливаем *s*='Привет', строковый объект *Привет* находится где-то в памяти и *s* является ссылкой на него. Затем, когда мы говорим *copy*=*x*, то мы действительно говорим, что *copy* является другой ссылкой на 'Привет'. Если, затем, мы делаем s = s + '!!!', то происходить, что создаётся новый объект 'Привет!!!' и поэтому мы присоединяем *s* к нему, *s* сейчас является ссылкой на тот новый объект 'Привет!!!'. Вспомните, что строки неизменяемы, поэтому 'Привет' не изменяется на что-либо. Вернее, Python создает новый объект и указывает переменную *s* на него.

Когда мы определяем *L*=[1,2,3], мы создаем объект списка, а ссылка *L* на него. Когда мы говорим *copy*=*L*, мы создаем другую ссылку на объект [1,2,3]. Когда мы делаем *L*[0]=9, из-за того, что списки изменяемы, список [1,2,3] изменится, *на месте*, в [9,2,3]. Новый объект не создается. Список [1,2,3] сейчас удаляется, а так как *copy* до сих пор указывает на то же самое место, значение которого [9,2,3].

Еще одно замечание, чтобы донести суть до конца. Если мы устанавливаем x = 487, а затем y=721, то сначала мы создаем целочисленный объект 487 и указываем *x* на него. Далее, когда мы определяем x=721, мы создаем новый целочисленный объект 721 и указываем x на него. Конечный эффект заключается в том, что кажется, что "значение" x меняется, но на самом деле меняется то, на что указывает x.

Сборка мусора Внутренне Python поддерживает счет количества ссылок имеющихся к каждому объекту. Когда количество ссылок на объект падает до 0, тогда объект больше не нужен и память, которую он использовал становится снова доступной.

19.2 Кортежи

Кортежи, по существу, неизменяемые списки. Ниже представлены - список с тремя элементами и кортеж с тремя элементами:

```
L = [1,2,3]
t = (1,2,3)
```

Кортежи заключены в круглые скобки, хотя скобки на самом деле опциональны. Индексация и срезы работают аналогично спискам. Подобно спискам, Вы можете получить длину кортежа использованием функции **len**, и как списки, у кортежей имеются методы *count* и *index*. Однако, так как кортеж неизменяем, он не имеет никаких других методов, которые имеют списки, например, *sort* и *reverse*, так как они изменяют список.

Мы уже видели кортежи в нескольких местах. Например, шрифты в Tkinter определены как пары,

например, ('Verdana', 14), а иногда как тройки. Метод словаря *items* возвращает список кортежей. Также, когда мы используем следующую короткую запись для изменения значений двух или более переменных, мы в действительности используем кортежи:

```
a,b = b,a
```

Одной из причин, почему существуют и списки и кортежи, является то, что в некоторых ситуациях Вам понадобится неизменяемый тип списка. Например, списки не могут служить ключами в словарях, потому что значения списков могут изменяться и это был бы кошмаром, Python придется следить за словарями. Кортежи, однако, могут служить ключами в словарях. Далее пример присвоения счета командам студентов:

```
grades = { ('John', 'Ann') : 95, ('Mike', 'Tazz'): 87 }
```

Также, в ситуациях где скорость действительно имеет значение, кортежи, как правило, быстрее чем списки. Гибкость списков сопровождается соответствующей ценой в скорости.

tuple Чтобы превратить объект в кортеж, используйте *tuple*. В следующем примере список и строка преобразуются в кортежи.

```
t1 = tuple([1,2,3])
t2 = tuple('abcde')
```

Обратите внимание () - это пустой кортеж. Способом, чтобы получить кортеж с одним элементом, является, например этот: (1,). Что-то, подобно (1), работать не будет, потому что это просто имеет значение 1, как при обычном вычислении. Например, в выражении 2 + (3*4), нам не нужно, чтобы (3*4) было кортежем, нам нужно, чтобы это вычислялось как число.

19.3 Множества

В Python имеется тип данных, который называется *множество*. Множества работают как математические множества. Они во многом похожи на списки без повторов. Множества обозначаются фигурными скобками, как внизу:

```
S = { 1,2,3,4,5}
```

Вспомните, что фигурные скобки также используются в обозначении словарей и { } является пустым словарем. Чтобы получить пустое множество, используйте функцию **set** без аргументов, как:

```
S = set()
```

Эта функция **set** может также быть использована, чтобы преобразовать объекты в множества. Далее два примера:

```
set([1,4,4,4,5,1,2,1,3])
set('это тест')
```

```
{1,2,3,4,5}
{'т', 'э', 'о', ' ', 'е', 'с'}
```

Обратите внимание, что Python будет хранить данные во множестве в любом порядке, в каком он захочет, не обязательно в том, который Вы определили. Имеют значение данные во множестве, а не порядок этих данных. Например, Вы не можете сделать s[0].

Работа с множествами Имеются несколько операторов, которые работают со множествами.

Оператор	Описание	Пример
|	объединение	{ 1,2,3} | { 3,4} →{ 1,2,3,4}
&	пересечение	{ 1,2,3} & { 3,4} →{ 3}
-	различие	{ 1,2,3} - { 3,4} →{ 1,2}
^	симметричная разница	{ 1,2,3} ^{ 3,4} →{ 1,2,4}
in	является элементом	3 in { 1,2,3} →True

Симметричная разница двух множеств дает элементы, которые имеются в одном или другом множестве, но не в обеих сразу. Ниже представлены некоторые полезные методы:

Метод	Описание
S.add(x)	Добавить x к множеству
S.remove(x)	Удалить x из множества
S.issubset(A)	Возвращает **True** если S $\subset$ A и **False** иначе
S.issuperset(A)	Возвращает **True** если A $\subset$ S и **False** иначе

Наконец, мы можем сделать генераторы множеств подобно генераторам списков:

```
s = { i ** 2 for i in range(12)}
```

```
{0, 1, 4, 100, 81, 64, 9, 16, 49, 121, 25, 36}
```

Генераторы множеств не представлены в Python 2.

Пример: удаление повторяющихся элементов из списков Мы можем использовать тот факт, что множества могут не иметь повторений, чтобы удалить все повторения из списка. Ниже пример:

```
L = [1, 4, 4, 4, 5, 1, 2, 1, 3]
L = list(set(L))
```

После этого, L будет равно [1, 2, 3, 4, 5].

Пример: игра слов Здесь пример инструкции if, которая использует функцию set, чтобы увидеть, является ли каждая буква слова - *a,b,c,d* или *e*:

```
if set(word).containedin('abcde')
```

19.4 Юникод

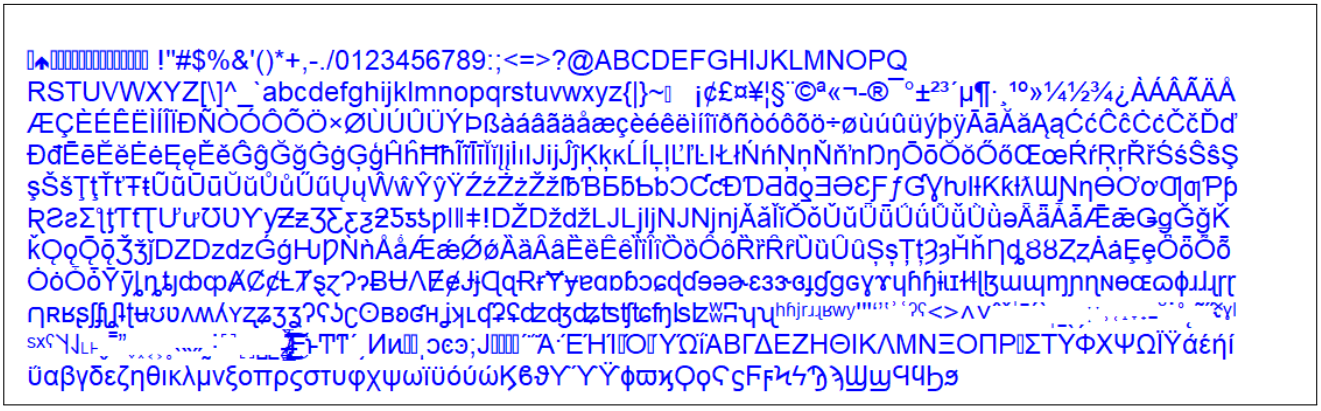
Это было раньше, когда компьютеры могли отображать только 255 различных символов, называемых ASCII символами. В этой системе, символам отводился один байт памяти, каждому, что давало 255 возможных символов, каждый с соответствующим числовым значением. Символы от 0 до 31 содержат различные символы контроля, включая '\n' и '\t'. После этого появились некоторые специальные символы, затем числа, заглавные буквы, прописные буквы и еще немного

символов. Помимо того, существует множество других символов, включая некоторые международные символы.

Однако, 255 символов не совсем достаточно, чтобы представить все символы используемых в алфавитах мира. Новым стандартом является *Unicode*, который применяет больше чем один байт для хранения информации о символе. Unicode, на данный момент, поддерживает более 65000 символов."Стандарт"не совсем верное здесь слово, так как в действительности в использовании имеются несколько стандартов и это может вызвать некоторые проблемы. Если Вам нужно работать с данными unicode, сделайте сначала некоторые исследования в нем, чтобы узнать о всем безумии.

В Unicode, символы до сих пор имеют числовые эквиваленты. Если Вы захотите переключаться между символом и его числовым эквивалентом и обратно, используйте встроенные функции **chr** и **ord**. Например, используйте **ord**('A'), чтобы получить числовое значение 'A', а **chr**(65), чтобы получить символ с числовым значением 65. Ниже, короткий пример, который выводит первые 1000 символов Unicode.

```
print(' '.join([chr(i) for  i  in range(1000)]))
```



Python поддерживает Unicode, и в строках и в именах переменных, функций и так далее. Имеются некоторые различия между Python 2 и Python 3 в поддержке Unicode.

19.5 sorted

Сначала определение: *итерируемый*(перебираемый) - это объект, который позволяет Вам поочередно проходить его содержание. Перебираемые объекты включают списки, кортежи, строки и словари. Существует ряд методов Python, которые работают на любом итерируемом объекте.

Функция **sorted** является встроенной функцией, которая может быть использована для сортирования итерируемого объекта. Как первый пример, мы можем использовать его для сортирования списка. Скажем L = [3, 1, 2]. Далее определяем M к [1, 2, 3].

```
M = sorted(L)
```

Различие между **sorted** и *L.sort* в том, что *L.sort*() изменяет начальный список *L*, а **sorted**(L) нет.

Функция **sorted** может быть применима на других итерируемых объектах. Результатом будет отсортированный список. Например, **sorted**('xyzab ') возвращает список ['a', 'b', 'x', 'y', 'z']. Если мы в реальности захотим результат в виде строки, то мы можем использовать метод *join*:

```
s = ' '.join(sorted('xyzab '))
```

Это сделает *s* равной 'xyzab '.

Использование **sorted** на словаре, отсортирует ключи.

19.6 Оператор if-else

Это удобный оператор, который может применен, чтобы объединить инструкцию if/else в одну строку. Ниже пример:

```
x = 'a'  if y == 4  else  'b'
```

Это эквивалентно следующему:

```
if y == 4:
    x = 'a'
else:
    x = 'b'
```

Далее другой пример с двумя образцами вывода:

```
print('He scored ' , score, 'point', 's' if  score>1 else  '.', sep='')
```

```
He scored 5 points.
He scored 1 point
```

19.7 continue

Инструкция **continue** является двоюродным братом инструкции **break** для циклов. Когда инструкция **continue** встречается в цикле for, программа игнорирует весь код в цикле после неё и переходит назад в начало цикла, передвигая счетчик цикла при необходимости. Далее пример. Код справа достигает тот же саый эффект, что и код слева.

```
for s in L:
    if s not in found:
        count +=1
        if s[0] == 'a':
            count2+=1
```

```
for s in L:
    if s in found: continue
    count +=1
    if s[0] == 'a':
        count2+=1
```

Инструкция **continue** - это то, без чего Вы конечно можете обойтись, но Вы время от времени можете видеть ее и иногда она может упростить код.

19.8 eval и exec

Функции **eval** и **exec** позволяют программе выполнять код Python во время её выполнения. Функция **eval** применяется для простых выражений, в то время как **exec** может выполнять произвольной длины блоки кода.

eval Мы видели **eval** много раз перед инструкциями input. Одной хорошей особенностью применения **eval** с инструкцией input, является то, что пользователю не нужно просто вводить число.

Он может ввести выражение и Python его вычислит. Например, скажем, у нас имеется следующее:

```
num = eval(input('Введите число: '))
```

Пользователь может ввести 3*(4+5) и Python вычислит это выражение.

Ниже пример **eval** в действии.

```
def countif(L, condition):
    return len([ i  for  i  in L if  eval(condition)])
```

Она ведет себя как функция электронной таблицы **COUNTIF**. Она считает сколько элементов в списке удовлетворяет определенному условию. Что **eval** делает для нас здесь то, что позволяет пользователю определить условие в виде строки. Например, *countif*(L, 'i>5') возвратит сколько элементов в *L* больше чем 5. Далее другая распространенная функция электронной таблицы:

```
def sumif(L, condition):
    return sum([ i  for  i  in L if  eval(condition)])
```

exec Функция **exec** принимает строку содержащую Python код и выполняет его. Далее пример:

```
s = """x=3
            for i in range(4):
                print(i*x) """
exec(s)
```

Одним из хороших применений функции **exec** является то, чтобы дать пользователю программы определить математические функции для использования во время выполнения программы. Далее код, который делает это:

```
s = input('Введите функцию: ')
exec('def f(x):  return ' + s)
```

Я использовал этот код в программе для построения графика, которая позволяет пользователям вводить уравнения, для построения графиков, и я применил его в прграмме, где пользователь может ввести функцию, а программа будет количественно округлять его корни.

Вы можете применять **exec**, чтобы Ваша программа генерировала все виды кода на Python пока она выполняется. Это позволит Вашей программе существенно изменять себя во время выполнения.

Обратите внимание В Python 2 **exec** является инструкцией, не функцией, поэтому Вы можете видеть её применение без круглых скобок в старом коде.

Вопрос безопасности Функции **eval** и **exec** могут быть опасными. Всегда существует случай, когда Ваши пользователи могли ввести некоторый код, который мог сделать что-то опасное в машине. Они могли также применить его для проверки значений Ваших переменных(что могло быть плохо, по некоторым причинам, Вы могли хранить пароли в переменных). Поэтому, Вам нужно быть осторожными при применении этих функций в коде, где важна безопасность. Одним вариантом для ввода данных без **eval** является что-то подобное этому:

```
num = int(input('Введите число: '))
```

Это предполагает, что *num* целое число. Применяйте **float** или **list** или что-либо еще подходящее к данным, которые Вы ожидаете.

19.9 enumerate и zip

Встроенная функция **enumarate** принимает итерируемый объект и возвращает новый итерируемый объект состоящий из пары(i, x), где i является индексом, а x соответствующий элемент из итерируемого объекта. Например:

```
s = 'abcde'
for (i, x) in enumerate(s):
    print(i + 1, x)
```

```
1  a
2  b
3  c
4  d
5  e
```

Возвращаемый объект является чем-то похожим на список пар, но не точно. Следующее даст список пар:

```
list(enumerate(s))
```

Цикл for выше эквивалентен следующему:

```
for  i  in  range(len(s)):
    print(i + 1, s[i])
```

Код **enumerate** может быть короче или понятнее в некоторых ситуациях. Далее пример, который возвращает список индексов всех единиц в строке:

```
[ j for  (j,c) in  enumerate(s)  if  c== '1' ]
```

zip Функция **zip** принимает два итерируемого объекта и "запаковывает"их в один, который содержит пары (x,y), где x из первого итерируемого объекта, а y из второго. Далее пример:

```
s = 'abc'
L = [10, 20, 30]
z = zip(s, L)
print(list(z))
```

```
[ (' a ', 10),  (' b ' 20,)  (' c ', 30 ]
```

Подобно с **enumerate**, результатом функции **zip** будет не совсем список, но если мы выполним **list**(**zip**(s, L)), мы можем из этого получить список.

Ниже пример, который использует **zip**, чтобы создать словарь из двух списков.

```
L = [ 'один ',  'два ',  'три ' ]
M = [ 4, 9, 15]
d = dict(zip(L, M))
```

```
{ 'три': 15,  'два': 9,  'один': 4 }
```

Метод может быть использован для создания словаря в то время когда Ваша программа выполняется.

19.10 copy

В модуле *copy* содержится пара полезных методов, *copy* и *deepcopy*. Метод *copy* может быть использован, например, чтобы сделать копию объекта из класса определенного пользователем. В качестве примера, предположим, у нас есть класс, называемый *Users* и мы хотим сделать копию определенного пользователя *u*. Мы могли сделать следующее:

```
from copy import copy
u_copy = copy(u)
```

Но у метода *copy* имеются определенные ограничения, как и у других методов копирования, например M=L[:] для списков. Для примера, предположим L = [1, 2, 3], [4, 5, 6]. Если мы создаем копию этого выполнением M=L[:], а затем задаем L[0][0]=100, то это повлияет также на M[0][0]. Это происходит потому, что копия является только *поверхностной копией* - ссылки на подсписки, составляющие L,которые были скопированы, вместо копий этих подсписков. Что-то подобное может быть проблемой в любой момент, когда мы копируем объект, который сам состоит из других объектов.

Метод *deepcopy* используется в такого рода ситуации, когда копируется только значения, а не ссылки. Ниже, как это бы работало:

```
from copy import deepcopy
M = deepcopy(L)
```

19.11 Больше о строках

Имеется немного больше фактов о строках про которые мы ещё не говорили.

translate Метод *translate* применяется, чтобы переводить строки посимвольно. Перевод делается в два шага. Сначала, применяется *maketrans* для создания специального типа словаря, который определяет, как данные будут переведены. Вы определяете обычный словарь, а он создает новый, который используется в переводе. Далее передаете тот словарь методу *translate*, чтобы сделать перевод. Далее простой пример:

```
d = str.maketrans( {'a':'1', 'b':'2'}  )
print('abaab'.translate(d))
```

Результат будет '12112'

Далее следует пример, в котором мы применяем *translate* для реализации простого шифра замены. Шифр замены является простым способом зашифровать сообщение, в котором каждая буква заменена другой буквой. Например, возможно каждая буква *а* на *к* , а каждая *б* на *в* и так далее. Ниже код:

```
from random import shuffle

# создание ключа
alphabet = 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя'
L = list(alphabet)
shuffle(L)

# создание кодирующего и декодирующего словарей
```

```
encode_dict = str.maketrans(dict(zip(alphabet, L)))
decode_dict = str.maketrans(dict(zip(L, alphabet)))

# кодирование и декодирование 'это секрет'
s = 'это секрет'.translate(encode_dict)
t = s.translate(decode_dict)
print(alphabet, ' '.join(L), t, s, sep='\ n')
```

```
абвгдеёжзийклмнопрстуфхцчшщъыьэюя
квиблюндмяруфйъзхоцжьапчшщгтыэесё
это секрет
ежз цюуоюж
```

Способ, с которым это работает, следующий: сначала мы создаем ключ шифрования, который говорит на какую букву заменится *а*, на какую букву заменится *б* и так далее. Это выполняется перемешиванием алфавита. Далее мы создаем таблицу преобразования для кодирования и декодирования, используя прием **zip** раздела 19.9 для создания словарей. Наконец, применяя метод *translate* выполняем фактическую замену.

partition Метод *partitition* похож на метод *split* списка. Различие проиллюстрировано ниже:

```
'3.14159'.partition('.')
'3.14159'.split('.')
```

```
( '3',  '.',  '14159')
[ '3',  '14159' ]
```

Разница заключается в том, что аргумент в функции возвращается как часть выходных данных. Метод *partitition* также возвращает кортеж вместо списка. Далее пример, который вычисляет производную простого одночлена введенного в виде строки. Правило для производных заключается в том, что производная ax^n равняется $nax^(n-1)$.

```
s = input('Введите одночлен: ')
coeff, power = s.partition('x^')
print( ' { }  'x^' { } .format(int(coeff) * int(power), int(power) - 1))
```

```
Введите одночлен: 2x^12
24x^11
```

Обратите внимание Эти методы и многие другие могли быть сделаны непосредственно, просто используя базовые инструменты языка, например, циклы for, инструкции if и так далее. Идея, однако, заключается в том, что эти вещи, которые обычно выполняются, превращаются в методы или классы, которые являются частью стандартного дистрибутива Python. Это может помочь Вам не изобретать колесо и они могут также сделать Ваши программы более надежными и легко читаемыми.

Сравнение строк Сравнение строк происходит в алфавитном порядке. Например, следующее выведет *Да*.

```
if  'that' < 'this':
    print('Да')
```

Кроме того, если строка содержит символы отличные от букв, сравнение основано на **ord** значении символов.

19.12 Разнообразные приёмы и трюки

Ниже представлено несколько полезных приемов:

Инструкции на одной строке Вы можете написать инструкцию if и инструкции, которые идут с ней на одной строке.

```
if x==3:  print('Привет')
```

Вы можете также расположить несколько инструкций в строке, если Вы разделяете их точкой с запятой. Например:

```
a=3;  b=4;  c=5
```

Не злоупотребляйте не одним из этих, так как они могут сделать Ваш код трудночитаемым. Хотя, иногда, с ними код может быть понятнее.

Вызов различных методов Вы можете вызвать несколько методов подряд, как ниже:

```
s = open('file.txt').read().upper()
```

Этот пример читает содержание файла, затем преобразует все в верхний регистр и хранит результат в *s*. Снова, будьте осторожны, не переборщите с таким количеством методов подряд или Ваш код может быть трудночитаем.

None В добавление к **int**, **float**, **str**, **list** и так далее, в Python имеется тип данных, называемый **None**. Он по существу является Python версией *ничего*. Он показывает, что ничего не имеется, когда Вы могли ожидать там что-то, такое как возвращаемое значение функции. Вы можете увидеть его появление и здесь и там.

Строка документирования Когда определяется функция, Вы можете определить строку, которая содержит информацию о том как функция работает. Затем, любой кто применяет функцию, может воспользоваться функцией Python **help**, чтобы получить информацию о функции. Далее пример:

```
def square(x):
    """Возвращает x в квадрате. """
    return x**2
```

```
help(square)
Help on function square in module __main__:

square(x)
    Возвращает x в квадрате.
```

Вы также можете применять строку документирования прямо сразу после инструкции **class**, чтобы предоставить информацию о Вашем классе.

19.13 Запуск Ваших Python программ на других компьютерах

Ваши программы могут быть запущены на других компьютерах, на которых установлен Python. Компьютеры с Macs и Linux обычно имеют установленный Python, хотя версия не может быть той которую Вы используете и эти компьютеры не могут иметь дополнительные библиотеки, которые Вы применяете. Вариантом на Windows является *py2exe*. Это сторонний модуль, который превращает программы на Python в исполняемые. На данный момент он доступен только для Python 2. Он может быть слегка затруднен в применении. Далее представлен скрипт, который можно применить только если Вы установили *py2exe*.

```
import os

program_name = raw_input('Введите имя программы: ')
if program_name[-3:] != '.py' :
    program_name += '.py'

with open('temp_py2exe.py', 'w') as fp:
    s = 'from distutils.core import setup \n'
    s += "import py2exe \nsetup (console=[ '"
    s += program_name + " ' ] ) "
    fp.write(s)

os.system('c: \Python26\python temp_py2exe py2exe')
```

Если все работает, должно всплыть окно и Вы увидете кучу всего происходящего быстро. Полученный исполняемый файл появится в новом подкаталоге каталога, где находится Ваш файл на Python, называемый *dist*. В том подкаталоге будет несколько других файлов, которые будет нужно подключить к Вашему исполняемому.

Глава 20

Полезные модули

Python поставляется с сотнями модулей, которые делают все что угодно. Существуют также сторонние модули, доступные для загрузки из интернета. В этой главе рассматриваются несколько модулей, которые я нахожу полезными.

20.1 Импортирование модулей

Существуют несколько различных способов импортирования модулей. Далее следуют несколько способов импортирования некоторых функций из модуля *Random*.

```
from random import randint, choice
from random import *
import random
```

1. Первый способ импортирует просто две функции из модуля.
2. Второй способ импортирует каждую функцию из модуля. Обычно Вам следует избегать делать это, так как модуль может содержать некоторые имена, которые будут конфликтовать с Вашими собственными именами переменных. Например, если Ваша программа использует переменную называемую *total* и Вы импортируете модуль, который содержит функцию, называемую *total*, то могут быть проблемы. Некоторые модуля, однако, как *tkinter*, являются довольно безопасными, чтобы импортировать таким способом.
3. Третий способ импортирует модуль полностью таким образом, что не будет пересекаться с Вашими именами переменных. Чтобы использовать функцию из модуля предваряйте её *random* с последующей точкой. Например, *random.randint*(1, 10).

Изменение имен модулей Ключевое слово as может быть использовано, чтобы изменить имя, которое Ваша программа использует ссылаясь на модуль или объект из модуля. Далее три примера:

```
import numpy as np
from itertools import combination_with_replacement as cwr
from math import log as ln
```

Расположение Обычно, инструкции import располагаются в начале программы, но ограничений нет. Они могут находиться где-либо при условии, что они идут перед кодом, который использует модуль.

Получение помощи Чтобы получить помощь по модулю (скажем модудь *random*) в оболочке Python, импортируйте его используя третий способ сверху. Далее **dir**(random) выдаст список функций и переменных в модуле, а **help**(random) даст Вам довольно длинное описание что все делает. Чтобы получить помощь по определенной функции, например *randint*, введите **help**(random.randint

20.2 Даты и время

Модуль *time* содержит некоторые полезные функции для работы со временем.

sleep Функция *sleep* останавливает Вашу программу на определенное время(в секундах). Например, чтобы остановить Вашу программу на 2 секунды или 50 миллисекунд, используйте следующее:

```
sleep(2)
sleep(.05)
```

Синхронизация объектов Функция *time* может быть использована для сиинхронизации объектов. Ниже пример:

```
from time import time
start = time()
# делать что-то
print('Это занимает', round(time()- start, 3), 'секунд.')
```

Для другого примера, смотри раздел 17.6, как разместить таймер обратного отсчета в GUI.

Размерность функции *time* в миллисекундах на Windows и микросекундах в Linux. Пример выше использует целые секунды. Если Вам нужно разрешение в миллисекундах, используйте следующую инструкцию print:

```
print(' { : . 3f } seconds'.format(time()- start))
```

Вы можете применить немного математики для этого, чтобы получить минуты и часы. Ниже пример:

```
t = time() - start
secs = t%60
mins = t//60
hours = mins//60
```

Кстати, когда Вы вызываете *time*(), Вы получаете довольно странное значение, как 1306372108.045. Это количество секунд прошедших с 1 января 1970 года.

Даты Модуль *datetime* позволяет нам работать с датами и временем вместе. Следующая строка создает объект *datetime*, который содержит текущую дату и время:

```
from datetime import datetime
d = datetime(1, 1, 1).now()
```

У объекта *datetime* имеются атрибуты - *year*, *month*, *day*, *hour*, *minute*, *second* и *microsecond*. Далее короткий пример:

```
d = datetime(1, 1, 1).now()
print( ' { } :{ :02d } { } / { } / { } ' . format(d.hour, d.minute, d.month, d.day, d.year))
```

```
7:33 2/1/2011
```

Часы представлены в 24-часовом формате. Чтобы получить 12-часовой формат Вы можете сделать следующее:

```
am_pm = 'am' if d.hour<12 else 'pm'
print( ' { } : }  { } / { } / { } ' . format(d.hour%12, d.minute, am_pm))
```

Альтернативный способ отображения даты и времени заключается в использовании метода *strftime*. Он применяет множество кодов форматирования, которые позволяют Вам отображать дату и время, включая информацию о дне недели, am_pm и так далее.

Далее представлены коды форматирования:

Код	Описание
%c	дата и время в формате соответствующие местному соглашению
%x, %X	%x- это дата, а %X - время, отформатированные подобно %c
%d	день месяца
%j	день года
%a,%A	день недели(%a сокращенное название дня недели)
%m	месяц(01-12)
%b,%B	название месяца(%b сокращенное название месяца)
%y,%Y	год(%y 2-значное, %Y 4-значное)
%H, %I	часы (%H 24-часовой, %I 12-часовой)
%p	am или pm
%M	минуты
%S	секунды

Вот пример:

```
print(d.strftime('%A  %x'))
```

```
Tuesday 02/01/11
```

Далее другой пример:

```
print(d.strftime('%c')) print(d.strftime('%I%p on %B %d'))
```

```
02/01/11 07:33:14
07AM on February 01
```

Начальные нули немного раздражают. Вы могли объединить *strftime* с первым способом, который мы узнали, чтобы получить кавсивый вывод:

```
print(d.strftime( ' { }  %p on %B  { }  ' ) . format(d.hour%12, d.day) )
```

```
7AM on February 1
```

Вы также можете создать объект *datetime*. Когда делаете так, Вы должны определить год, месяц и день. Другие атрибуты опциональны. Вот пример:

```
d = datetime(2011, 2, 1, 7, 33)
e = datetime(2011, 2, 1)
```

Вы можете сравнивать объекты *datetime* применяя <, > и != операторы. Вы можете также делать арифметические вычисления на объектах *datetime*, однако мы не будем здесь рассматривать это. В действительности, существует много больше, что Вы можете сделать с датами и временем.

Другим хорошим модулем является *calendar*, который Вы можете использовать для вывода календарей и делать более сложные вычисления с датами.

20.3 Работа с файлами и каталогами

Модуль *os* и субмодуль *os.path* содержат функции для работы с файлами и каталогами.

Изменение каталога Когда Ваша программа открывает файл, то предполагается, что он будет в том же самом каталоге, как и сама Ваша программа. Если нет, то Вам нужно определить каталог, как ниже:

```
s = open('c:/users/heinold/desktop/file.txt').read()
```

Если у Вас есть много файлов, которые нужно прочитать, все в том же самом каталоге, Вы можете использовать *os.chdir*, чтобы изменить каталог. Далее пример:

```
os.chdir('c:/users/heinold/desktop/file.txt')
s = open('file.txt').read()
```

Получение текущего каталога Функция *getcwd* возвращает путь текущего каталога. Это будет каталог в котором находится Ваша программа или каталог, который Вы изменили с *os.chdir*.

Получение файлов в каталоге Функция *listdir* возвращает список составляющих каталог, включая все файлы и подкаталоги. Если Вам нужны просто файлы, без подкаталогов или наоборот, то модуль *os.path* содержит функции *isfile* и *isdir*, чтобы сообщить, что данные являются файлами или каталогами. Вот пример, который просматривает все файлы в каталоге и выводит имена этих файлов, которые содержат слово 'hello'.

```
import os

directory = 'c:/users/heinold/desktop/'
files = os.listdir(directory)
for f in files:
    if os.path.isfile(directory + f):
        s = open(directory + f).read()
        if 'hello' in s
            print(f)
```

Изменение и удаление файлов Далее несколько полезных функций. Только будьте осторожны здесь.

Функция	Описание
mkdir	создание каталога
rmdir	удаление каталога
remove	удаление файла
rename	переименование файла

Первые две функции принимают путь к каталогу, как единственный аргумент. Функция *remove* принимает единственное имя файла. Первым аргументом *rename* является старое имя, а вторым новое имя.

Копирование файлов В модуле *os* не имеется функций для копирования файлов. Взамен используйте функцию *copy* в модуле *shutil*. Вот пример, который берет все файлы в каталоге и делает копию каждого, каждое имя скопированного файла начинается с *Copy of*:

```
import shutil

directory = 'c:/users/heinold/desktop/'
files = os.listdir(directory)
for f in files
    if os.path.isfile(directory + f):
        shutil.copy(directory + f, directory + 'Copy of ' + f)
```

Больше с os.path Модуль *os.path* содержит ряд других функций, которые полезны для работы с файлами и каталогами. Разные операционные системы имеют различные соглашения о том, как они обрабатывают пути, и функции в *os.path* позволяют Вашим программам работать в различных операционных системах не учитывая особенности каждой. Далее несколько примеров(на моей системе Windows):

```
print(os.path.split('c:/users/heinold/desktop/file.txt'))
print(os.path.basename('c:/users/heinold/desktop/file.txt'))
print(os.path.dirname('c:/users/heinold/desktop/file.txt'))
print(os.path.join('directory', 'file.txt'))
```

```
('c:/users/heinold/desktop ', 'file.txt ')
file.txt
c:/users/heinold/desktop
directory\\file.txt
```

Обратите внимание, что стандартным разделителем в Windows является обратная косая черта. Прямая косая тоже работает.

Наконец, двумя другими функциями, которые Вы найдете полезными, значатся - функция *exists*, которая проверяет существование файла или каталога, и *getsize*, которая определяет размер файла. Имеется много других функций в *os.path*. Смотрите документацию Python [1] для дополнительной информации.

os.walk Функция *os.walk* позволяет Вам сканировать каталог и все его подкаталоги. Вот простой пример, который находит все Python файлы на моем рабочем столе или подкаталогах моего рабочего стола:

```
for (path, dirs, files) in os.walk('c:/users/heinold/desktop/'):
    for filename in files:
        if filename[-3:] == '.py':
            print(filename)
```

20.4 Запуск и завершение программ

Запуск программ Существуют несколько разных способов для Вашей программы запустить другую программу. Один из них применяет функцию *system* в модуле *os*. Далее пример:

```
import os

os.chdir('c:/users/heinold/desktop')
os.system('file.exe')
```

Функция *system* может быть использована для запуска команд, которые Вы можете запускать в командной строке. Другим способом запуска Ваших программ является применение функции *execv*.

Завершение Ваших программ Модуль *sys* содержит функцию, называемую *exit*, которая может использована для завершения Вашей программы. Вот простой пример:

```
import sys
ans = input('Завершить программу?')
if ans.lower() == 'да'
    sys.exit()
```

20.5 Zip файлы

Zip файл представляет собой сжатый файл или каталог файлов. Следующий код извлекает все файлы из zip файла, *filename.zip* на мой рабочий стол:

```
import zipfile
z = zipfile.ZipFile('filename.zip')
z.extractall('c:/users/heinold/desktop/')
```

20.6 Получение файлов из интернета

Для получения файлов из интернета существует модуль *urllib*. Вот простой пример:

```
from urllib.request import urlopen
page = urlopen('http://www.google.com')
s = page.read().decode()
```

Функция *urlopen* возвращает объект, который сильно похож на файловый объект. В примере выше, мы применяем методы *read*() и *decode*(), чтобы прочитать все содержание страницы в строку *s*.

Строка *s*, в примере выше, заполнена текстом HTML файла, который не приятно читать. В Python имеются модули для анализа HTML, но мы не будем рассматривать их здесь. Код выше полезен для загрузки обычных текстовых файлов данных из интернета.

Для чего-либо более сложного чем это, рассмотрите применение сторонней requests library

20.7 Звук

Легкий способ получить некоторые простые звуки в Вашей программе заключается в применении модуля *winsound*. Он работает только с Windows, однако. Одна функция в *winsound* называется *Beep*, которая может быть использована для проигрывания тона с заданной частотой для заданного количества времени. Ниже простой пример, который проигрывает звук 500 Гц длительностью 1 секунда.

```
from winsound import Beep
Beer(500, 1000)
```

Первый аргумент *Beep* - это частота, второй продолжительность в миллисекундах.

Другой функцией в *winsound* является *Playsound*, которая может быть применена для проигрывания WAV файлов. Здесь пример:

```
from winsound import PlaySound
PlaySound('soundfile.wav', 'SND_ALIAS')
```

С другой стороны, если Вы установили Pygame, это легко проиграть любой тип обычного звукового файла. Это показано ниже, и это работает на системах отличных от Python:

```
import pygame
pygame.mixer.init(18000, -16, 2, 1024)
sound = pygame.mixer.Sound('soundfile.wav')
sound.play()
```

20.8 Ваши собственные модули

Создание своих собственных модулей легко. Просто напишите свой код на Python и сохраните его в файле. Затем Вы можете импортировать свой модуль используя инструкцию **import**.

Глава 21

Регулярные выражения

Строковый метод *replace* используется для замены всех вхождений одной строки на другую, а метод *index* применяется, чтобы найти первое вхождение подстроки в строке. Но иногда Вам понадобиться сделать более сложный поиск или замену. Например, Вам может понадобиться найти все вхождения строки вместо просто одного. Или возможно Вы захотите найти все вхождения двух букв с последующим числом. Или возможно Вам понадобиться заменить каждое 'qu', которое находится в начале слова на 'Qu'. Это является тем, для чего предназначены регулярные выражения. Инструменты для работы с ними находятся в модуле *re*.

Существует некоторый синтаксис для изучения, чтобы понимать регулярные выражения. Далее следует один пример, который даёт Вам представление как они работают:

```
import re
print(re.sub(r '( [LRUD]) (\d+)', '***', 'Locations L3 and D22 full.'))
```

21.1 Введение

sub Функция *sub* работает следующим образом:

```
sub(pattern, replacement, string)
```

Она выполняет поиск *string*(строки) по *pattern*(шаблону) и заменяет что-либо соответствующему этому шаблону на строку *replacement*(замена). Все предстоящие примеры будут показаны с *sub*, но существуют другие вещи, которые мы можем делать с регулярными выражениями, помимо замены. Мы перейдем к ним после обсуждения синтаксиса регулярных выражений.

Необработанные строки Множество шаблонов используют обратные слеши. Однако, обратные слеши в строках применяются для управляющих символов, например новая строка, \n. Чтобы получить обратный слеш в строке нам нужно сделать \\. Это может быстро загромоздить регулярные выражения. Чтобы избежать этого, наши шаблоны будут *необработанными строками*, где обратные слеши могут появиться как есть и не делать ничего особого. Чтобы отметить строку, как необработанную строку, разместите перед ней *r*, по аналогии внизу:

```
s = r'Это необработанная строка. Обратные слеши не делают ничего особенного.'
```

21.2 Синтаксис

Основной пример Мы начнем с регулярного выражения, которое имитирует строковый метод *replace*. Далее пример использования *replace* для замены всех вхождений *abc* на *:

```
' abcdef abcxyz'.replace(' abc ', ' * ')
```

```
*def *xyz
```

Здесь код регулярного выражения, который делает то же самое:

```
re.sub(r' abc ', ' * ', ' abcdef abcxyz ')
```

Квадратные скобки Мы можем употребить квадратные скобки для указания, что нам нужны для сопоставления определенные буквы. Ниже пример, где мы заменяем каждую *a* и *d* на звездочки:

```
re.sub(r ' [ad] ', ' * ', ' abcdef ')
```

```
*bc*ef
```

Далее другой пример, где звездочка заменяет все вхождения *a*, *b* или *c*, за которыми следуют 1, 2 или 3:

```
re.sub(r ' [abc] [123] ', ' * ', ' a1 + b2 + c5 + x2 ')
```

```
* + * + c5 + x2
```

Мы можем задавать диапазон значений — например, [a-j] для букв от a до j. Далее несколько больше примеров диапазонов:

Диапазон	Описание
$[A-Z]$	любая заглавная буква
$[0-9]$	любая цифра
$[A-Za-z0-9]$	любая цифра или число

Немного коротким способом для сопоставления любого числа является \d, вместо [0-9].

Сравнение(сопоставление) любого символа Используйте точку(.), чтобы сравнить (почти) любой символ. Вот пример:

```
re.sub(r ' A.B ', ' * ', ' A2B AxB AxxB A$B')
```

```
* * AxxB *
```

Шаблон сопоставление *A* за которой следует почти любой одиночный символ, а затем *B*.

Исключение: Одним символом не сочетаемым с точкой является символ новой строки. Если Вам нужно, чтобы сравнивался, тоже, расположите ?s в начале Вашего шаблона.

Сопоставление различных копий чего-либо Далее представлен пример, где мы сопоставляем *A* с последующей одной или более *B*:

```
re.sub(r ' AB+ ', ' * ', ' ABC ABBBBBBC AC ')
```

```
*C *C AC
```

Мы употребили символ +, чтобы указать, что нам нужно сопоставление одну или более *B* здесь. Существуют аналогичные вещи, которые мы можем использовать для определения различного количества *B* здесь. Например, употребление * вместо + будет сопоставлять ноль или больше *B*.(Это означает, что *AC* в примере выше было бы заменено **C* потому что *A* считается, как *A* с последующем нулем *B*.) Ниже приведена таблица того, что Вы можете делать:

Код	Описание
+	сопоставлять 1 или более появлений
⋆	сопоставлять 1 или более появлений
?	сопоставлять 0 или 1 появление
{ m}	сопоставлять точно m появлений
{ m, n}	сопоставлять между m и n появлений, включительно

Далее пример, который сопоставляет *A* с последующими от 3 до 6 *B*:

```
re.sub(r ' AB { 3,6 } ', ' * ', ' ABB ABBB ABBBB ABBBBBBBBB ' )
```

```
'ABB * * *BBB '
```

Здесь, мы не сопоставляем *ABB* потому, что за *A* следуют только две *B*. Следующие две части совпадают, так как за *A* следуют три *B* на втором месте, а на третьем - четыре *B*. В последней части находится *A* с девятью следуемыми *B*. Что совпадает, *A* с первыми шестью *B*.

Обратите внимание, что сопоставление в последней части выше является *жадным*; то есть, берется так много *B*, сколько допустимо. Допустимо брать между тремя и шестью *B*, а берется все шесть. Чтобы получить противоположный эффект, взять так мало, сколько позволено, добавьте ?, как ниже:

```
re.sub(r ' AB { 3,6 } ? ', ' * ', ' ABB ABBB ABBBB ABBBBBBBBB ' )
```

```
'ABB * * *BBBBBB '
```

? может следовать за любыми числовыми определителями, например +?, -?, ?? и так далее.

Символ | Символ | действует как "or". Ниже пример:

```
re.sub(r ' abc|xyz', ' * ', ' abcdefxyz123abc ')
```

```
'*def*123* '
```

В примере выше, каждый раз, когда мы сталкиваемся с *abc* или *xyz*, то заменяем их на звездочку.

Сопоставление только в начале или конце Иногда Вам не нужно сопоставлять каждое вхождение чего-либо, возможно только первое или последнее вхождение. Чтобы сопоставить только первое вхождение чего-либо, начните шаблон(pattern) с символа ^. Чтобы сопоставить только последнее вхождение, закончите шаблон символом $. Ниже примеры:

```
re.sub(' ^abc ', ' * ', ' abcdefgabc ')
re.sub(' abc$ ', ' * ', ' abcdefgabc ')
```

```
*defgabc
abcdefg*
```

Специальные управляюшие символы Мы видели, что + и * имеют специальные значения. Что если нам понадобиться сопоставить знак плюс? Чтобы сделать это, примените обратный слеш, например \+. Далее пример:

```
re.sub(r ' AB\+ ', ' * ', ' AB+C ')
```

```
*C
```

Также в шаблоне, \n представляет новую строку.
Просто снова обратите внимание на необработанные строки — если мы не используем их для шаблонов, каждый обратный слеш должен бы быть сдвоенным. Например, r ' AB\+' должен бы быть ' \\+ '

Последовательности обратного слеша

- \d сопоставляет любую цифру, а \D любую не цифру. Например:

```
re.sub(r ' \d ', ' * ', ' 3 + 14 = 17 ')
re.sub(r ' \D ', ' * ', ' 3 + 14 = 17 ')
```

```
* + * = **
3* * *14* * *17
```

- \w сопоставляет любую букву или число, а \W все остальное. Вот пример:

```
re.sub(r ' \w ', ' * ', ' Это тест. Или это? ')
re.sub(r ' \W ', ' * ', ' Это тест. Или это? ')
```

```
'* * *  * * * * . * * *  * * *? '
'Это*тест**Или*это*'
```

Это хороший способ для работы со словами.

- \s сопоставляет пробел, а \S не пробел. Ниже пример:

```
re.sub(r ' \s ', ' * ', ' Это тест. Или это? ' )
re.sub(r ' \S ', ' * ', ' Это тест. Или это? ' )
```

```
'Это*тест.*Или*нет? '
'***  ***** *** **** '
```

Предшествующее и последующее сопоставление Иногда Вам понадобиться сопоставить вещи если им что-то предшествует или что-то за ними последует.

Код	Описание
(?=)	сопоставляется только если последуется
(?!)	сопоставляется только если не последуется
(?<=)	сопоставляется только если предшествуется
(?<!)	сопоставляется только если не предшествуется

Ниже пример, который сопоставляет слово *the* только если за ним следует *cat*:

```
re.sub(r ' the(?= cat)', ' * ', ' the dog and the cat')
```

```
'the dog and * cat '
```

Далее пример, который сопоставляет слово *the* только, если перед ним находится пробел:

```
re.sub(r '(?<= )the', ' *', ' Athens is the capital. ')
```

```
Athens is * capital.
```

Следующий пример будет сопоставлять слово *the* только если перед ним или после него нет букв, таким образом Вы можете применить его для замены вхождений слова *the*, но не вхождения *the* в состав других слов.

```
re.sub(r ' (?<!\w [Tt]he(?!\w)', ' * ', ' The cat is on the lathe there. ')
```

```
* cat is on * lathe there.
```

Флаги Существует ряд флагов, которые Вы можете применять, чтобы влиять на поведение регулярных выражений. Мы взглянем на несколько из них здесь.

- (?i) — Это для игнорирования регистра. Вот пример:

```
re.sub(' (?i) ab ', ' * ', ' ab AB ')
```

```
*  *
```

- (?s) — Вспомните, что символ . сопоставляет любой символ, исключая символ новой строки. Этот флаг позволяет этому символу также сопоставлять символы новой строки.

- (?x) — Регулярные выражения могут быть длинными и сложными. Этот флаг позволяет Вам использовать более многословный, многострочный формат, в котором пробелы игнорируются. Вы также можете добавить комментарии. Ниже пример:

```
pattern = r """(?x) [AB]\d+   # Сопоставлять A или B с последующими цифрами
                    [CD]\d+   # Сопоставлять C или D с последующими цифрами
                    """
print(re.sub(pattern, ' * ', ' A3C9 and C1B17 '))
```

```
* and *
```

21.3 Краткое содержание

Имеется много для запоминания. Далее следует краткое содержание:

Код	Описание
[]	любые символы внутри скобок
.	любой символ, исключая новую строку
+	1 или более предшествующего
*	0 или более предшествующего
?	0 или 1 предшествующего
{ m}	точно m предшествующего
{ m,n}	между m и n(включительно) предшествующего
?	со следующими +,*,?,{ m} и { m,n} — брать сколько возможно
\	специальный управляющий символ
|	"or"
^	(в начале шаблона) сопоставляется первое вхождение
$	(в конце шаблона) сопоставляется последнее вхождение
\d \D	любая цифра(не цифра)
\w \W	любая буква или цифра(не буква или не цифра)
\s \S	любой пробел(не пробел)
(?=)	только если последуется
(?!)	только если не последуется
(?<=)	только если предшествуется
(?<!)	только если не предшествуется
(?i)	флаг, чтобы игнорировать регистр
(?s)	флаг,для того чтобы символ . , сопостовлял символ новой строки тоже
(?x)	флаг для доступа к многострочному формату

Далее приведены короткие примеры:

Выражение	Описание
' abc '	точная строка *abc*
'[*ABC*]'	*A*, *B* или *C*
' [*a* − *zA* − *Z*][0 − 9]'	сопоставляет букву с последующей цифрой
' [*a*..]'	*a* с последующими двумя символами(исключая новую строку)
' a+ '	одна или более букв *a*
' a* '	любое число *a* даже не одной
' a? '	ноль или одна *a*
' a { 2 } '	точно две *a*
' a { 2, 4 } '	две, три или четыре *a*
' a+? '	одна или больше *a*, взятых сколько возможно
' a \. '	*a* следующее в периоде
' ab|zy '	*ab* или *zy*
' ^a'	первое *a*
' a$'	последнее *a*
' \d'	каждая цифра
' \w '	каждая буква или число
' \s '	каждый пробел
' \D '	все, исключая цифры
' \W '	все, исключая буквы и числа
' \S '	все, исключая пробелы
' a(?=b) '	каждая *a* с последующей *b*
' a(?!b) '	каждая *a* с не последующей *b*
' (?<=b)a'	каждая *a* с предществующей *b*
' (?<!b)a'	каждая *a* с не предшествующей *b*

Обратите внимание Обратите внимание, во всех примерах этого раздела, мы имеем дело с неповторяющимися шаблонами. Например, если мы посмотрим на шаблон ' aba ' в строке ' abababa ', то увидим, что имеется несколько повторяющихся совпадений. Все наши сопоставления сделаны слева и не учитывают повторений. Например, у нас имеется следующее:

```
re.sub(' aba ', ' * ', ' abababa ')
```

```
'*b* '
```

21.4 Группы

Применение круглых скобок вокруг части выражения создает *группу*, которая содержит текст, который сопоставляется с шаблоном. Вы можете использовать это, чтобы делать более сложные замещения. Ниже пример, который переводит в нижний регистр каждую заглавную букву, за которой следует прописная буква:

```
def modify(match):
    letters = match.group()
    return letters.lower()
re.sub(r ' ( [A − Z]) [a − z]', modify, 'PEACH Apple ApriCot ')
```

```
PEACH apple apricot
```

Функция *modify* завершается, когда вызывается три раза, один за другим, когда происходит сопоставление. Функция *re.sub* автоматически посылает объект сравнения в функцию *modify*. Объект содержит информацию о сравниваемом тексте. Метод объекта *group* возвращает сам сопоставляемый текст.

Если вместо *match.group* мы применим *match.groups*, тогда мы сможем дополнительно разбить сопоставление в соответствии с группами, определенными круглыми скобками. Далее пример, который сопоставляет заглавную букву с последующей цифрой и конвертировать каждую букву в нижний регистр, добавляя 10 к каждому числу:

```
def modify(match):
    letter, number = match.groups()
    return letter.lower() + str(int(number) + 10)
re.sub(r ' ( [A − Z]) (\d ', modify, ' A1 + B2 + C7 ')
```

```
a11 + b12 + c17
```

Метод *groups* возвращает сопоставляемый текст как кортежи. Например, в программе выше кортежи возвращаются, как представлено ниже:

```
Первое сопоставление: ( 'A', '1')
Второе сопоставление: ( 'B', '2')
Третье сопоставление: ( 'C', '7')
```

Обратите внимание, что мы можем получить эту информацию передавая аргументы в *match.group*. Для первого сопоставления, *match.group*(1) является 'A ', а *match.group*(2) - это 1.

21.5 Другие функции

- *sub* — Мы видели много примеров с *sub*. Одну вещь мы не упомянули, то существует опциональный аргумент *count*, который определяет сколько сопоставлений (слева) делать. Ниже пример:

```
re.sub(r ' a ', ' * ', ' abababababa ', count=2)
```

```
'*b*bababa '
```

- *findall* — функция *findall*, которая возвращает список всех найденных совпадений. Вот пример:

```
re.findall(r ' [AB]\d', ' A3 + B2 + A9 ')
```

```
[' A3 ', ' B2 ', ' A9 ']
```

Как другой пример, чтобы найти все слова в строке, Вы можете сделать следующее:

```
re.findall(r ' \w+ ', s)
```

Это лучше, чем использование *s.split*(), потому что *split* не обрабатывает знаки пунктуации, в то время как регулярные выражения делают.

- *split* — Функция *split* аналогична строковому методу *split*. Версия регулярного выражения позволяет нам разделять на что-то более общее, чем это делает строковый метод. Вот пример, который разделяет алгебраическое выражение при + или -.

```
re.split(r ' \+|\- ', ' 3x + 4y - 12x^2 + 7 ')
```

```
[ ' 3x ', '4y ',' 12x^2 ', ' 7 ']
```

- *match* и *search* — Они пригодятся, если Вам нужно знать, что совпадения происходят. Различие между двумя этими функциями заключаются в том, что *match* проверяет только, соответствует ли начало строки шаблону, в то время как *search* перебирает всю строку пока не найдет соответствие. Обе возвращают **None**, если не найдут соответствия и объект Соответствия, если найдут. Ниже представлены примеры:

```
if(re.match(r ' ZZZ ', ' abc ZZZ xyz '))
    print(' Совпадение найдено в начале ')
else:
    print(' В начале совпадений нет ')

if(re.search(r ' ZZZ ', ' abc ZZZ xyz '))
    print(' Совпадение найдено в строке ')
else:
    print(' Совпадений не найдено ')
```

```
В начале совпадений нет
Совпадение найдено в строке
```

Объект Соответствия возвращаемый этими функциями содержит в себе информацию. Скажем, у нас есть следующее:

```
a = re.search(r ' ( [ ABC ] ) (\d) ', ' = A3 + B2 + C8 ')
a.group()
a.group(1)
a.group(2)
```

Помните, что *search* будет сообщать только о первом совпадении, которое она обнаружит в строке.

```
' A3 '
' A '
' 3 '
```

- *finditer* — Она возвращает итератор объекта Соответствия, когда мы можем итерировать(проходить цикл), например:

```
for s in re.finditer(r ' ( [ AB ] ) (\d)', ' A3 + B4'):
    print(s.group(1))
```

```
A
B
```

Обратите внимание, что она немного более общая, чем функция *findall*, в которой *findall* возвращает строки соответствия, в то время как *finditer* возвращает что-то похожее на список объектов Соответствия, которые дают нам доступ к информации группы.

- *compile* — Если Вы собираетесь повторно применять тот же самый шаблон, то Вы сможете сохранить немного времени предварительной сборкой шаблона, как показано ниже:

```
pattern = re.compile(r ' [ AB ] \d ')
pattern.sub(' * ', ' A3 + B4 ')
pattern.sub(' x ', ' A8 + B9 ')
```

```
* + *
x + x
```

Когда Вы собираете выражение, для многих методов Вы можете определить опциональные начальные и конечные индексы в строке. Далее пример:

```
pattern = re.compile(r ' [ AB ] \d ')
pattern.findall(' A3 + B4 + C9 + D8 ', 2, 6)
```

```
[ 'B4']
```

21.6 Примеры

Римские числа Здесь мы применяем регулярные выражения, чтобы перевести римские числа в обычные.

```
import re

d = {  'M':1000, 'CM':900, 'D':500, 'CD':400, 'C':100, 'XC':90,
       'L':50, 'XL':40, 'X':10, 'IX':9, 'V':5, 'IV':4 , 'I':1}

pattern = re.compile(r"""(?x)
                         (M{ 0,3}  (CM)?
                         (CD)? (D)? (C{ 0,3} )
                         (XC)? (XL)? (L)? (X{ 0,3} )
                         (IX)? (IV)? (V)? (I{ 0,3} )""")

num = input('Введите римскую цифру: ').upper()
m = pattern.match(num)

sum = 0
for  x  in  m.groups():
   if  x!=None  and  x!=' ':
       if  x in  [ ' CM ', 'CD ', ' XC ', ' XL ', ' IX ', ' IV ' ]:
           sum+=d[ x ]
       elif x[ 0 ]  in  ' MDCLXVI ':
           sum+=d[ x[0] ] * len(x)
print(sum)
```

```
Введите римскую цифру: MCMXVII
1917
```

Регулярное выражение само довольно простое. Оно ищет до трех *M* с последующим нулем или одну *CM* с последующим нулем или одну *CD* и так далее, и хранит каждую из них в группе. Затем цикл for прочитывает эти группы и использует словарь,чтобы добавить соответствующие значения к текущей сумме.

Даты Здесь мы применяем регулярное выражение, чтобы взять дату в подробном формате, например февраль 6, 2011, и преобразовать её в сокращенный формат, мм/дд/гг(без начальных нулей). Вместо того, чтобы зависеть от пользователя, который вводит дату в точно правильном способе, мы можем использовать регулярное выражение, которое допускает разнообразие и ошибки. Например, эта программа будет работать независимо от того, прописывает ли пользователь название месяца целиком или сокращает его(с точкой или без). Заглавные буквы не имеют значения и также не имеет значение если даже он вводит название месяца правильно. Пользователь просто должен ввести первые три буквы правильно. Также не имеет значение сколько пробелов он применяет и ставит ли запятую после дня.

```
import re

d = { 'jan':'1', 'feb':'2', 'mar':'3', 'apr':'4',
      'may':'5', 'jun':'6', 'jul':'7', 'aug':'8',
      'sep':'9', 'oct':'10', 'nov':'11', 'dec':'12' }

date = input('Введите дату: ')
m = re.match('([A-Za-z]+)\.?\s*(\d{ 1,2 }),?\s*(\d{ 4 })', date)
print('{ } / { } / { } '.)format(d[m.group(1) .lower() [:3]],
                                  m.group(2), m.group(3) [-2:]))
```

```
Введите дату: feb. 6, 2011
2/6/11
```

Первая часть регулярного выражения, ([A-Za-z]+) \.?, следит за названием месяца. Оно соответствует количеству букв в названии месяца, которые пользователь вводит. \.? соответствует 0 или 1 точке после названия месяца, поэтому пользователь может вводить точку или нет. Круглые скобки вокруг букв сохраняют результат в группе 1.

Дальше, мы находим день: \s*(\d{ 1,2}). Первая часть \s* соответствует нулю или больше символов пробела, поэтому не имеет значения сколько пробелов пользователь расположит между месяцем и днем. Остаток соответствует одной или двум цифрам и сохраняет это в группе 2.

Остаток выражения, ?\s*(\d{ 4 }), находит год. Затем мы используем результаты для создания сокращенной даты. Сложной частью здесь является первая часть, которая принимает название месяца, переводит его в нижний регистр и берет только первые три символа, и использует их как ключи словаря. Таким способом, при условии, что пользователь правильно введет три буквы названия месяца, программа будет понимать это.

Глава 22

Math

Эта глава является коллекцией тем, которые в какой то степени по своей сути, хотя многие из них вызывают общий интерес.

22.1 Модуль math

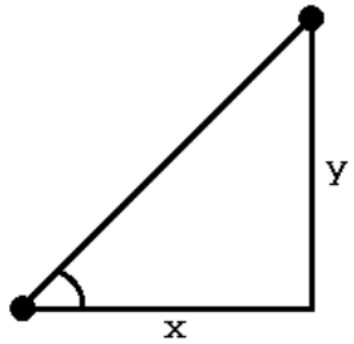
Как упоминалось в разделе 3.5, модуль *math* содержит некоторые обычные математические функции. Далее представлены большинство из них:

Функция	Описание
sin,cos,tan	тригонометрические функции
asin,acos,atan	обратные тригонометрические функции
atan2(y,x)	дает arctan(y/x) с правильным поведением знака
sinh,cosh,tanh	гиперболические функции
asinh,acosh,atanh	обратные гиберболические функции
log, log10	натуральный логарифм, десятичный логарифм
log1p	*log*(1+x), более точно около 1, чем *log*
exp	экспоненциальная функция e^x
degrees, radians	преобразует из радиан в градусы или наоборот
floor	*floor*(x) наибольшее целое число $\leq x$
ceil	*ceil*(x) наименьшее целое число $\geq x$
e,pi	постоянные e, π
factorial	факториал
modf	возвращает пару(дробную часть, целую часть)
gamma, erf	функция Γ и функция Error

Обратите внимание Обратите внимание, что функции *floor* и **int** ведут себя одинаково с положительными числами, но различно с отрицательными. Например, *floor*(3.57) и **int**(3.57), обе возвращают целое число 3. Однако, *floor*(-3.57) возвращает -4, наибольшее целое число меньше чем или равно -3.57, в то время как **int**(-3.57) возвращает -3, которое получено отбрасыванием десятичной части числа.

atan2 Функция *atan2* полезна для определения угла между двумя точками. Допустим x и y - это расстояния между точками x и y направлениях. Тангес угла, на рисунке ниже, определяется y/x.

Тогда арктангес(y/x) дает сам угол.



Но если угол был 90°, это бы не работало, так как x был бы 0. Нам понадобились бы особые случаи, для обработки, когда x<0 и когда y<0. Функция *atan2* обрабатывает все это. Выполнение *atan2*(y,x) вернет арктангес(y/x) со всеми случаями обработанными правильно.

Функция *atan2* возвращает результат в радианах. Если Вы пожелаете градусы, делайте следующее:

```
angle = math.degrees(atan2(y,x))
```

Результирующий угол от *atan2* будет между -π и π($-180°$ и 180°). Если Вы захотите, чтобы он находился в диапазоне 0 и 360°, делайте следующее:

```
angle = math.degrees(atan2(y,x))
angle = (angle+360) % 360
```

22.2 Научная запись

Взгляните на следующий код:

```
100.1**10
```

```
1.0100451202102516e + 20
```

Полученное значение отображается в научной записи. Это 1.0100451202102516 × $10^2$0. e+20 обозначает $10^2$0. Далее другой пример:

```
0.15**10
```

```
5.7665039062499975e − 09
```

Это равно 5.7665039062499975 × 10^{-9}.

22.3 Сравнение чисел с плавающей точкой

В разделе 3.1, мы увидели, что некоторые числа, например .1, не представлены точно на компьютере. Математически, после того как код ниже выполнится, x должен быть 1, но из-за накопленных ошибок, он действительно равен 0.9999999999999999.

```
x = 0
for  i  in  range(10):
    x+=0.1
```

Это означает, что следующая инструкция if окажется **False**:

```
if  x==1:
```

Более надежным способом для сравнения чисел с плавающей точкой x и y является ли проверка отличия между двумя числами достаточно маленькой, как ниже:

```
if  abs(x-y)<10e-12:
```

22.4 Дроби

Существует модуль, называемый *fractions*, для работы с дробями. Вот простой пример модуля в работе:

```
from  fractions  import Fractions
r = Fractions(3, 4)
s = Fractions(1, 4)
print(r+s)
```

```
Fraction(1, 1)
```

Вы можете выполнять основные арифметические операции с объектами *Fractions*, а также сравнивать их, беря их абсолютные значения и так далее. Далее дополнительные примеры:

```
r = Fraction(3, 4)
s = Fraction(2, 8)
print(s)
print(abs(2*r-3))
if r>s:
    print('r больше')
```

```
Fraction(1,4)
Fraction(3,2)
r больше
```

Обратите внимание, что *Fractions* автоматически преобразует дроби к несократимому виду. Пример ниже показывает как получить числитель и знаменатель:

```
r = Fraction(3,4)
r.numerator
r.denominator
```

```
3
4
```

Преобразование в и из чисел с плавающей точкой Чтобы преобразовать дробь в число с плавающей точкой, применяйте **float**, как ниже:

```
float(Fraction(1,8))
```

```
0.125
```

С другой стороны, скажем, нам нужно конвертировать 0.3 в дробь. К сожалению, мы не должны выполнять *Fraction*(0.3), потому что, как упоминалось, некоторые числа, включая 0.3, не представлены точно на компьютере. В действительности, *Fraction*(0.3) возвратит следующий объект *Fraction*:

```
Fraction(5404319552844595, 18014398509481984)
```

Вместо этого используйте строковую запись, как ниже:

```
Fraction('0.3')
```

Ограничение знаменателя Одним полезным методом является *limit_ denominator*. Для заданного объекта *Fraction*, *limit_ denominator*(x) находит самую близкую дробь к тому значению, чей знаменатель не превышает *x*. Далее несколько примеров:

```
Fraction('.333').limit_denominator(100)
Fraction('.333').limit_denominator(1000)
Fraction('3.14159').limit_denominator(1000)
```

```
Fraction(1, 3)
Fraction(333, 1000)
Fraction(355, 113)
```

Последний пример возвращает достаточно близкое дробное приближение к π. Оно отличается меньше чем 0.0000003.

Самый большой общий делитель Модуль *fractions* содержит полезную функцию, называемую *gcd*, которая возвращает самый большой общий делитель двух чисел. Ниже пример:

```
from fractions import gcd
print('Наибольший общий коэффициент, который имеют 35 и 21 равен ', gcd(35, 21))
```

```
Наибольший общий коэффициент, который имеют 35 и 21 равен 7
```

22.5 Модуль decimal

В Python содержится модуль, называемый *decimal* для выполнения точных вычислений с десятичными числами. Как мы неоднократно обратили внимание, некоторые числа, такие как 0.3, не могут быть представлены точно как числа с плавающей точкой. Далее следует, как получить точное десятичное представление 0.3:

```
from decimal import Decimal
Decimal('0.3')
```

Строка здесь важна. Если мы пропустим её, мы получим десятичное, которое соответствует представлению 0.3 с неточной плавающей точкой.

```
Decimal(0.3)
```

```
Decimal('0.2999999999999999888977697537484345957636831909179687 5')
```

Математика Мы можем использовать обычные математические операторы для работы с объектами Decimal. Например:

```
Decimal(.34) + Decimal(.17)
```

```
Decimal('0.51')
```

Здесь другой пример:

```
Decimal(1) / Decimal(17)
```

```
Decimal('0.058823529411764705882352941l8')
```

Математические функции *exp*, *ln*, *log10* и *sqrt* являются методами десятичных объектов. Например, следующий пример вычисляет квадратный корень 2:

```
Decimal(2).sqrt()
```

```
Decimal('1.41421356237309504880168872')
```

Десятичные объекты также могут быть использованы со встроенными функциями **max**, **min** и **sum**, а также преобразованы в числа с плавающей точкой **float** и строки **str**.

Точность По умолчанию объекты Decimal имеют 28-значная точность. Чтобы изменить её на, скажем, 5-значную, используйте функцию *getcontext*.

```
from decimal import getcontext
getcontext().prec = 5
```

Вот пример, который выводит 100 цифр $\sqrt{2}$:

```
getcontext().prec = 100
Decimal(2).sqrt()
```

```
Decimal('1.41421356237309504880168872420969807856967187537694807317667973
799073247846210703885038753432764157 ')
```

Теоретически не существует ограничений точности, которую Вы можете использовать, но чем выше точность, тем больше памяти требуется и медленнее процессы будут проходить. В общем, даже с маленькими точностями, Decimal объекты медленнее, чем числа с плавающей точкой. Обычные арифметические операции с числами с плавающей точкой достаточны для большинства задач, но хорошо знать, что Вы можете получить высокую точность, если она Вам понадобится.

В модуле *decimal* имеется гораздо больше. Смотри документацию Python [1]

22.6 Комплексные числа

В Python присутствует тип данных для комплексных чисел.

В математике мы имеем $i = \sqrt{-1}$. Число i называется *мнимое число*. *Комплексное число* - это число формы $a + bi$, где a и b реальные числа. Значение a называется *действительной частью*, а b является мнимой частью. В электротехнике символ j используется вместо i, а в Python j применяется для мнимых чисел. Далее пара примеров создания комплексных чисел:

```
x = 7j
x = 1j
x = 3.4 + .3j
x = complex
```

Если число оканчивается на *j* или *J*, Python обрабатывает его как комплексное число.

У комплексных чисел существуют методы *real*() и *imag*(), которые возвращают действительную и мнимую части числа. Метод *conjugate* возвращает комплексное сопряжение(сопряжением *a + bi* является *a - bi*)

Модуль *cmath* содержит одни и те же функции, что модуль *math*, за исключением того, что они работают с комплексными аргументами. Функции включают обычные, обратные и гиперболические тригонометрические функции, логарифмы и экспоненциальную функцию. Он также содержит две функции, *polar* и *rect*, для конвертации между прямоугольными и полярными координатами:

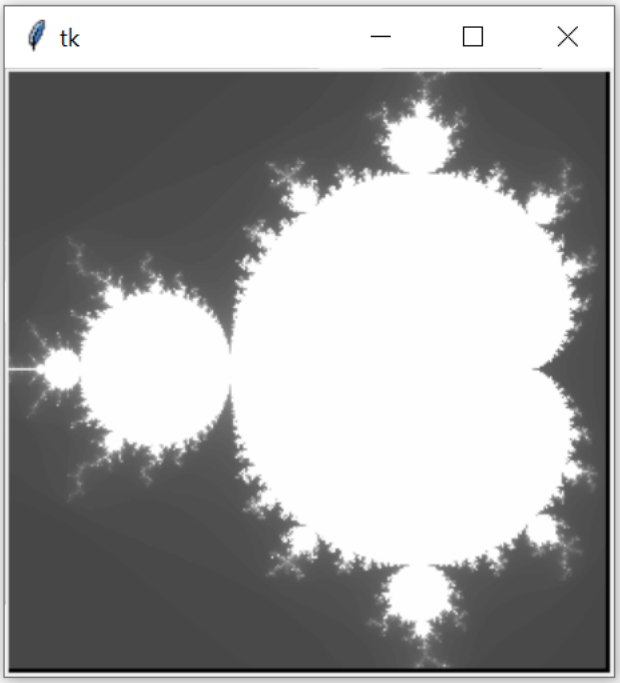
```
cmath.polar(3j)
cmath.rect(3.0, 1.5707963267948966)
```

```
(3.0, 1.5707963267948966)
(1.8369701987210297e-16+3j)
```

Комплексные числа интересны, хотя не все то полезно в повседневной жизни. Одним из классных применений, однако, являются фракталы. Далее программа, которая рисует известное множество Мандельброта. Программа требует PIL и Python 2.6 или 2.7.

```
from Tkinter import *
from PIL import Image, ImageTk, ImageDraw

def color_convert(r, g, b):
    return '# { 0:02x} {1:02x} {2:02x } '.format(int(r*2.55), int(g*2.55), int(b*2.55))

max_iter=75
xtrans=-.5
ytrans=0
xzoom=150
yzoom=-150

root = Tk()
canvas = Canvas(width=300, height=300)
canvas.grid()
image=Image.new(mode='RGB', size =(300, 300))
draw = ImageDraw.Draw(image)

for x in range(300):
    c_x = (x-150)/ float(xzoom) + xtrans
    for y in range(300):
            c = complex(c_x, (y-150)/float(yzoom) + ytrans)
            count=0
            z=0j
            while abs(z)<2 and count<max_iter:
                    z = z*z+c
                    count +=1
            draw.point((x,y),
                            fill=color_convert(count+25, count+25, count+25))
    canvas.delete(ALL)
```

```
    photo=ImageTk.PhotoImage(image)
    canvas.create_image(0,0,image=photo,anchor=NW)
    canvas.update()

mainloop()
```



Код здесь выполняется очень медленно. Существуют способы ускорить его немного, но к сожалению Python медленный для такого рода вещей.

22.7 Больше со списками и массивами

Разреженные списки Список целых чисел, размером 10,000,000 x 10,000,000, требует несколько сотен террабайт памяти, много больше, чем могут хранить большинство твердых дисков. Однако во многих практических приложениях, большинство элементов списка равняются 0. Это позволяет нам сохранить много памяти, просто сохраняя ненулевые значения вместе с их расположением. Мы можем использовать словарь, ключи которого - это местоположения ненулевых элементов, а значения являются значениями, которые хранятся в массиве с этими местоположениями. Например, предположим у нас имеется двухмерный список L чьи элементы все нули, за исключением, что L[10][12] равняется 47, а L[100][245]=18. Вот словарь, который мы бы использовали:

```
d = { (10, 12): 47, (100, 245) : 18 }
```

Модуль array В Python имеется модуль, называемый *array*, который определяет объект array, сильно похожий на список, за исключением того, что его элементы должны быть того же самого типа. Преимущество *array* перед списками заключается в более эффективном использовании памяти и быстрой производительности. Смотрите документацию Python [1], чтобы узнать больше о массивах.

Библиотеки NumPy и SciPy Если Вам понадобится выполнить сложные математические или научные вычисления на массивах, то можете рассмотреть библиотеку NumPy. Она легко загружается и устанавливается. Из руководства пользователя NumPy:

> NumPy является фундаментальным пакетом для научных вычислений в Python. Это библиотека Python, которая предоставляет объект многомерного массива, различные производные объекты(такие как маскированные массивы и матрицы), и ассортимент процедур для быстрых операций на массивах, включая математические, логические, манипулирование формой, сортирование, отбор, ввод/вывод, дискретное преобразование Фурье, базовая линейная алгебра, базовые статистические операции, случайное моделирование и многое другое.

Также существует SciPy, которая основана на NumPy. На этот раз из руководства пользователя SciPy:

> SciPy - это коллекция математических алгоритмов и удобных функций построенная на расширении NumPy для Python. Она значительно расширяет возможности интерактивного сеанса с Python, предоставлением пользователю команды и классы высокого уровня для управления и визуализации данных. С SciPy интерактивный сеанс становится средой обработки данных и прототипирования систем, конкурирующей с такими системами, как MAT-LAB, IDL,Octave, R-Lab и SciLab.

22.8 Случайные числа

Как Python генерирует случайные числа Генератор случайного числа, который Python использует, называется Mersenne Twister(Вихрь Мерсенна). Он надежный и хорошо проверенный. Это детерминированный (определяемый) генератор, означающий, что он применяет математический процесс, чтобы генерировать случайные числа. Числа называются *псевдослучайные числа* потому, что происходят из математического процесса, они не совсем случайные, хотя для всех намерений и целей они кажутся случайными. Любой, кто знает процесс, может воссоздать числа. Это нормально для научных и математических приложений, где Вы захотите иметь возможность восстановить те же самые случайные числа для целей тестирования, но это не пригодно для криптографии, где Вы должны генерировать по настоящему случайные числа.

Seeds Для математических и научных приложений, Вы может захотите воспроизвести те же самые случайные числа каждый раз когда Ваша программа выполняется, для того чтобы протестировать Вашу работу с теми же данными. Вы можете сделать это определением *seed*. Примеры показаны ниже:

```
random.seed(1)
print("Seed 1: ", [random.randint(1,10) for i in range(5))
random.seed(2)
print("Seed 2: ", [random.randint(1,10) for i in range(5))
random.seed(1)
print("Seed 1: ", [random.randint(1,10) for i in range(5))
```

```
Seed 1: [ 2, 9, 8, 3, 5 ]
Seed 2: [ 10, 10, 1, 1, 9 ]
Seed 1: [ 2, 9, 8, 3, 5 ]
```

seed может быть любым целым числом. Если мы просто используем *random.seed*(), тогда *seed* будет более или менее случайно выбрано основываясь на системных часах.

Функция random Большинство функций в модуле *random* основаны на функции *random* , которая применяет Вихрь Мерсенна, чтобы сгенерировать случайные числа между 0 и 1. Затем математические преобразования используются на результате *random*, чтобы получить некоторые

из наиболее интересных функций случайных чисел .

Другие функции в модуле random Модуль *random* содержит функции, которые возвращают случайные числа из различных распределений, например Гауссовское или экспоненциальное распределения. Например, чтобы сгенерировать Гауссовскую(обычную) случайную величину, примените функцию *gauss*. Далее примеры:

```
random.gauss(64,3.5)
[ round(random.gauss(64,3.5), 1)  for  i  in  range(10)]
```

```
61.37965975173485
[58.4, 61.0, 67.0, 67.9, 63.2, 65.0, 64.5, 63.4, 65.5, 67.3]
```

Первым аргументом *gauss* является среднее значение, а второй - среднеквадратичное отклонение. Если Вы не знакомы с обычными случайными величинами, то они соответствуют стандартной колоколообразной кривой. Вещи подобные росту и баллам SAT и многим другим реальным показателям приблизительно соответствуют этому распределению. В примере выше, сгенерированные случайные числа сосредоточены вокруг 64. Числа, близкие к 64 будут сгенерированы с большей вероятностью, чем числа ррасположенные дальше.

Существует группа других распределений, которые мы можем использовать. Самым распространенным является равномерное распределение, в котором все значения в ряду примерно одинаковы. Например:

```
random.uniform(3,8)
```

```
7.535110252245726
```

Смотрите документацию Python [1] для информации по другим распределениям.

Более случайная функция randint Одним из способов сгенерировать криптографически стойкие случайные числа является использование достаточно случайного физического процесса. Примеры включают радиоактивный распад, атмосферные явления и поведение определенных электрических цепей. Модуль *os* содержит функцию *urandom*, которая генерирует случайные числа из физического процесса, чей точный характер зависит от Вашей системы. Функция *urandom* принимает один аргумент говорящий ему сколько байт случайных данных произвести. Вызов *urandom*(1) создает 1 байт информации, который мы можем перевести в целое число между 0 и 255. Вызов *urandom*(2)производит 2 байта данных, переводящиеся в целые числа между 0 и 65535. Вот функция, которая действует подобно Вызов *randint*, но использует Вызов *urandom*, чтобы предоставить нам неопределяемые случайные числа:

```
from  os  import  urandom
from  math  import  log

def urandint(a,b):
    x = urandom(int(log(b-a+1)/log(256)) +1)
    total = 0
    for  (i,y)  in  enumerate(x):
          total += y*(2**i)
    return total%(b-a+1) + 1
```

Способ её работы следующий: сначала мы должны определить сколько байт случайной информации сгенерировать. Так как один байт дает 256 возможных значений, а два байта предоставляют 265^2 потенциальных величин и так далее, мы вычисляем логарифм основания 256 размера диапазона *b-a+1*, чтобы определить сколько байт сгенерировать. Затем мы передаем сгенерированные

байты в цикл и преобразуем их в целое число. Наконец, изменяя то целое число *b-a+1*, уменьшая это целое число к числу между 0 и *b-a+1*, и добавляя *a* к тому созданному целому числу в желаемом диапазоне.

22.9 Разнообразные темы

Шестнадцатеричная, восьмеричная и двоичная В Python имеются функции **hex**, **oct** и **bin** для преобразования целых чисел в шестнадцатеричные, восьмеричные и двоичные. Функция **int** переводит эти основания в основание 10. Ниже представлены примеры:

```
hex(250)
oct(250)
bin(250)
int(0xfa)
```

```
'0xfa '
'0o372'
'0b11111010 '
250
```

Шестнадцатеричным значениям предшествует 0x, восьмеричным 0o, а двоичные предваряются префиксом 0b.

Функция int Функция **int** содержит второй опциональный аргумент, который позволяет Вам определить основание из которого Вы преобразуете. Далее несколько примеров:

```
int('101101', 2) # перевести из основания 2
int('121212', 3) # перевести из основания 3
int('12A04', 11 # перевести из основания 11
int('12K04', 23) # перевести из основания 23
```

```
45
455
18517
314759
```

Функция pow В Python есть встроенная функция, называемая **pow**, которая возводит числа в степень. Она работает аналогично оператору **, за исключением того, что принимает третий аргумент, который определяет модуль. Таким образом **pow**(x,y,n) возвращает (x**y)%n. Причина по которой Вы захотите применять это заключается в том, что **pow** способ является намного быстрее, когда вовлекаются очень большие числа, которые часто случаются в криптографических приложениях.

22.10 Использование оболочки Python как калькулятора

Я часто использую оболочку Python в качестве калькулятора. Этот раздел содержит несколько приемов для работы в оболочке.

Импортирование математических функций Хорошим способом начать работу в оболочке является импортирование некоторых математических функций:

```
from math import *
```

Специальная переменная Существует специальная переменная _, которая хранит значение предыдущего вычисления. Ниже пример:

```
>>> 23**2
529
>>> _+1
530
```

Логарифмы Я много использую натуральный логарифм и для меня более привычно вводить *ln* вместо *log*. Если Вам захочется сделать то же, просто сделайте следующее:

```
ln = log
```

Суммирование рядов Вот способ получить среднюю сумму рядов в этом случае $\sum_{n=0}^{\infty} \frac{1}{n^2-1}$:

```
>>> sum( [1/(n**2) for n in range(2, 1000)])
0.7489994994995
```

Другой пример: Скажем Вам потребовался синус каждого из углов 0, 15, 30, 45, 60, 75 и 90. Вот быстрый способ сделать то:

```
>>> [round(sin(radians(i)), 4) for  i  in  range(0, 91, 15)]
[ 0.0, 0.2588, 0.5, 0.7071, 0.866, 0.9659, 1.0 ]
```

Сторонние модули Существует ряд других сторонних модулей, которые Вы могли найти полезными, когда работаете в оболочке Python. Например, имеются Numpy и SciPy, о которых мы упомянули в разделе 22.7. Имеется также Matplotlib, универсальная библиотека для вычерчивания объектов, и существует Sympy, которая выполняет символьные вычисления.

Глава 23

Работа с функциями

Эта глава рассматривает ряд тем для работы с функциями, включая некоторые темы функционального программирования.

23.1 Функции первого класса

Считается, функции Python являются функциями первого класса, что означает, что они могут быть присвоены переменным, скопированы, использованы как аргументы других функций и так далее, как любой другой объект.

Копирование функций Вот пример, где мы делаем копию функции:

```
def f(x):
    return x*x
g = f
print('f(3) =', f(3), 'g(3) =', g(3), sep = '\n')
```

```
f(3) = 9
g(3) = 9
```

Списки функций Затем у нас имеется список функций:

```
def f(x):
    return x**2
def g(x):
    return x**3
funcs = [ f, g ]
print(funcs[0] (5), funcs[1] (5), sep = '\n' )
```

```
25
125
```

Вот другой пример. Скажем, у нас имеется программа с десятью различными функциями и программа должна решить с каким временем выполнения какую функцию использовать. Одним из решений является использование десяти инструкций if. Короткое решение заключается в использовании списка функций. Пример ниже предполагает, что мы уже создали функции *f1*, *f2*, . . . , *f10*, каждая которая принимает два аргумента.

```
funcs = [ f1, f2, f3, f4, f5, f6, f7, f8, f9, f10 ]
num = eval(input('Введите число: '))
funcs[num] ((3,5))
```

Функции как аргументы к функциям Скажем, у нас имеется список 2 кортежей. Если мы сортируем список, то сортирование выполняется на основании первого элемента, как ниже:

```
L = [ (5, 4), (3, 2), (1, 7), (8, 1) ]
L.sort()
```

```
[ (1, 7), (3, 2), (5, 4), (8, 1) ]
```

Предположим, нам нужно, чтобы сортировка выполнялась на основе второго элемента. Метод *sort* принимает опциональный аргумент, называемый *key*, который является функцией и определяет как сортировка должна быть сделана. Вот как производится сортировка основываясь на втором элементе:

```
def comp(x):
    return x[1]
L = [ (5, 4), (3, 2), (1, 7), (8, 1) ]
L.sort(key=comp)
```

```
[ (8, 1), (3, 2), (5, 4), (1, 7) ]
```

Здесь другой пример, где мы сортируем список строк по длине, а не по алфавиту.

```
L = [ 'это', 'тест', 'разделения', 'списка' ]
L.sort(key=len)
```

```
[ 'это ', 'тест ', 'списка ', 'разделения ']
```

Ещё одним местом, где мы видели функции как аргументы к другим функциям, являются функции обратных вызовов кнопок Tkinter.

23.2 Анонимные функции

В одном из примеров выше, мы передали функцию сравнения методу *sort*. Далее код снова:

```
def comp(x):
    return x[1]
L = [ (5, 4), (3, 2), (1, 7), (8, 1) ]
L.sort(key=comp)
```

Если у нас имеется действительно короткая функция, которую мы собираемся применять только один раз, тогда мы можем использовать то, что называется анонимной функцией, наподобие внизу:

```
L.sort(key=lambda x: x[1])
```

Ключевое слово lambda указывает то, что следует будет анонимной функцией. Затем у нас имеются аргументы к функции, за которыми следует двоеточие и далее код функции. Код функции не может быть длиннее чем одна строка.

Мы применяли анонимные функции раньше, когда работали с GUI, чтобы передать информацию о том какая кнопка была нажата в функцию обратного вызова. Далее снова код:

```
for i in range(3):
    for j in range(3):
        b[i][j] = Button(command = lambda x=i, y=j: function(x, y))
```

23.3 Рекурсия

Рекурсия - это процесс, в котором функция вызывает саму себя. Одним из стандартных примеров рекурсии является факториальная функция. Факториал, $n!$, продукт всех чисел от 1 до n. Например, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Также, по соглашению, $0! = 1$. Рекурсия включает в себя определение функции в терминах самой себя. Обратите внимание, что например, $5! = 5 \cdot 4!$ и в общем, $n! = n \cdot (n\text{-}1)!$. Поэтому факториальная функция может быть определена в терминах самой себя. Далее рекурсивная версия факториальной функции:

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Мы должны определить случай n=0 или иначе функция будет продолжать вызывать себя вечно(или по крайней мере пока Python не сгенерирует ошибку о слишком большом уровне рекурсии).

Обратите внимание, что модуль *math* содержит функцию, называемую *factorial*, поэтому эта версия здесь представлена просто для демонстрации. Заметьте также, что существует не рекурсивный способ сделать факториал, используя цикл for. Это примерно так же просто, как рекурсивный способ, но быстрее. Однако, для некоторых задач рекурсивное решение является более простым решением. Далее следует программа, которая раскладывает число на простые множители.

```
def factor(num, L=[]):
    for  i  in  range(2, num//2 + 1):
        if num%==0:
            return L+[i]+factor(num//i)
    return L+[num]
```

Функция *factor* принимает два аргумента: число для разложения и список предыдущих найденных множителей. Она проверяет наличие множителя и если она находит, то добавляет его в список. Рекурсивная часть заключается в том, что она делит число на множитель, который она нашла, а затем добавляет в список все множители того значения. С другой стороны, если функция не находит каких-либо множителей, то она добавляет число в список, поскольку оно должно быть простым и возвращает новый список.

23.4 map, filter, reduce и генераторы списков

map и filter В Python находятся встроенные функции, называемые **map** и **filter**, которые используются для применения функций к содержанию списка. Они появились прежде чем генераторы списков были частью Python, но сейчас генераторы списков могут выполнять все, что могут эти функции. До сих пор, Вы могли случайно видеть код использующий эти функции, поэтому хорошо о них знать.

Функция **map** принимает два аргумента - функцию и итерируемый объект - и она применяет функцию к каждому элементу объекта, создавая новый итерируемый объект. Вот пример, который принимает список строк и возвращает список длин строк. Первая строка выполняет это с **map**, в то время как вторая использует генератор списка:

```
L = list(map(len, [ 'это', 'простой', 'тест' ]))
L = [ len(word)  for  word  in [ 'это', 'простой', 'тест' ]]
```

Функция **filter** принимает функцию и итерируемый объект, а возвращает итерируемые объекты из всех элементов списка для которых функция верна. Далее пример, который возвращает все слова в списке, имеющие длину больше 2. Первая строка применяет **filter**, чтобы сделать это, а вторая делает это с генератором списка:

```
L = list(filter(lambda, x: len(x)>2, [ 'это', 'простой', 'тест' ]))

L = [word  for  word  in  [ 'это', 'простой', 'тест' ]  if  len(word)>2 ]
```

Далее один способ для нахождения количества элементов в списке L, которые больше чем 60:

```
count = 0
for  i  in  L:
    if i>60:
        count = count + 1
```

Ниже второй метод использующий генератор списка подобный функции **filter**:

```
len([ i  for  i  in  L  if i>60])
```

Второй способ и короткий и легкий для понимания.

reduce Существует другая функция, **reduce**, которая применяет функцию к содержанию списка. Она использовалась как встроенная функция, но в Python 3 она была перемещена в модуль *functools*. Эта функция не может быть легко заменена генераторами списков. Чтобы понять её, сначала рассмотрим простой пример, который суммирует числа от 1 до 100.

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

Функция **reduce** может быть применена, чтобы сделать это в одну строку:

```
total = reduce( lambda x, y: x+y, range(1, 101))
```

В общем, **reduce** принимает функцию и итерируемый объект, и применяет функцию к элементам слева направо, накапливая результат. Как другой простой пример, факториальная функция могла быть улучшена применяя reduce:

```
def fact(n):
    return reduce(lambda x, y: x*y, range(1, n+1))
```

23.5 Оператор module

В предыдущем разделе, когда нам требовалась функция, для представления оператора Python, например сложение или умножение, мы использовали анонимную функцию, как внизу:

```
total = reduce( lambda x, y: x+y, range(1, 101))
```

В Python имеется модуль, называемый *operator*, который содержит функции, выполняющие то же самое, что и операторы Python. Они выполняются быстрее чем анонимные функции. Мы можем переписать пример выше как:

```
from operator import add
total = reduce(add, range(1, 101))
```

Модуль *operator* содержит функции аналогичные арифметическим операторам, логическим операторам и даже вещи типа срезов и оператора in

23.6 Больше об аргументах функций

Вы можете захотеть написать функцию, для которой Вы не знаете сколько аргументов ей будет передано. Примером является функция **print**, куда Вы можете ввести столько объектов, сколько Вы хотите вывести, каждую отделяя запятой.

Python позволяет нам объявить специальный аргумент, который собирает несколько других аргументов в кортеж. Этот синтаксис продемонстрирован ниже:

```
def product(*nums):
    prod = 1
    for i in nums:
        prod*=i
    return prod
print(product(3,4), product(2, 3, 4), sep='\n')
```

```
12
24
```

Существует аналогичное обозначение, **, для набора произвольного числа аргументов ключевых слов в словаре. Ниже простой пример:

```
def f(**keyargs):
    for k in keyargs:
        print(k, '**2 : ', keyargs[k] ** 2, sep= ' ')
f(x=3, y=4, z=5)
```

```
y**2 : 16
x**2 : 9
z**2 : 25
```

Вы можете также использовать эти обозначения вместе с обычными аргументами. Порядок имеет значение — аргументы собранные при помощи * должны следовать после всех позиционных аргументов, а аргументы собранные при помощи ** всегда идут последними. Два примера объявления функций показаны ниже:

```
def func(a, b, c=5, *d, **e):
def func(a, b, *c, d=5, **e):
```

Вызов функций Обозначения * и ** могут быть использованы также когда вызывается функция. Далее пример:

```
def f(a,b):
    print(a+b)
x=(3,5)
f(*x)
```

Она выведет 8. В этом случае мы могли более просто вызвать *f*(3,5), но возникают ситуации когда это невозможно. Например, возможно у Вас есть несколько разных наборов аргументов, которые Ваша программа могла бы применить для функции. У Вас могло быть несколько инструкций if, но если имеется множество наборов аргументов, тогда запись * будет намного проще. Далее

следует простой пример:

```
def f(a,b):
    (a+b)
args = [ (1,2), (3,4), (5,6), (7,8), (9,10) ]
i = eval(input('Введите число от 0 до 4:'))
f(*args[i])
```

Одно из применений записи ** заключается в упрощении обявлений Tkinter. Предположим, у нас имеются несколько виджетов, у всех из которых одинаковые свойства, скажем один и тот же шрифт, цвет символов и фон. Вместо повторения этих свойств в каждом объявлении, мы можем сохранить их в словаре, а потом использовать запись ** в объявлении, как ниже:

```
args = { 'fg': 'blue', 'bg': 'white', 'font': ('Verdana', 16, 'bold') }
label1 = Label(text = 'Label 1', **args)
label2 = Label(text = 'Label 2', **args)
```

apply В Python 2 имеется функция, называемая **apply**, которая является больее или менее, эквивалентом * и ** для вызова функций. Вы можете увидеть её в старом коде.

Переменные функции, которые сохраняют свои значения между вызовами Иногда полезно иметь переменные, которые являются локальными для функции, и сохраняют свои значения между вызовами функции. Так как функции являются объектами, мы можем выполнить это добавлением переменной в функцию, так как это был бы более обычный вид объекта. В примере ниже переменная *f.count* отслеживает сколько раз функция вызывалась.

```
def f():
    f.count = f.count + 1
    print(f.count)
f.count=0
```

Глава 24

Модули itertools и collections

Модули *itertools* и *collection* содержат функции, которые могут существенно упростить некоторые обычные задачи программирования. Мы начнем с некоторых функций в *itertools* .

24.1 Permutations и combinations

Permutations(Перестановки) Перестановки последовательности - это изменение порядка элементов этой последовательности. Например, перестановками [1,2,3] являются [3,2,1] и [1,3,2]. Ниже пример, который показывает все возможные варианты:

```
list(permutations([1,2,3]))
```

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

Мы можем найти перестановки любого итерируемого объекта. Вот перестановки строки '123':

```
[ ''.join(p) for p in permutations('123') ]
```

```
[ '123', '132', '213', '231', '312', '321']
```

Функция *permutations* принимает опциональный аргумент, который позволяет нам определять размер перестановок. Например, если нам просто требуются все возможные двухэлементные подстроки из '123', то мы можем сделать следующее:

```
[ ''.join(p) for p in permutations('123', 2) ]
```

```
[ '12',  '13',  '21',  '23',  '31',  '32']
```

Обратите внимание, что *permutations* и большинство других функций в модуле *itertools* возвращают итератор. Вы можете перебрать в цикле элементы в итераторе и применить **list**, чтобы преобразовать его в список.

Combinations(Комбинации) Если нам нужны все возможные k-элементы подмножеств из последовательности, где все что имеет значение является элементами, но не порядок в котором они появляются, тогда то что нам нужно — это комбинации(сочетания). Например, 2-элементное подмножество, которое может быть сделано из 1,2,3 будет 1,2, 1,3 и 2,3. Мы рассматриваем 1,2 и 2,1 как то же самое, потому что они содержать одинаковые элементы. Вот пример, показывающий возможные комбинации двухэлементных подстрок из '123':

```
[ ''.join(c) for c in combinations('123', 2) ]
```

```
[ '12',  '13',  '23']
```

Комбинации с заменой Для сочетаний с повторяющимися элементами используйте функцию *combinations_with_replacement*.

```
[ ' '.join(c)  for  c  in  combinations_with_replacement('123', 2) ]
```

```
[ '11',  '12',  '13',  '22',  '23',  '33']
```

24.2 Декартово произведение

Функция *product* создает итератор из Декартова произведения итерируемых объектов. Декартово произведение двух множеств X и Y состоит из всех пар (x, y), где x находится в X, а y в Y. Ниже короткий пример:

```
[ ' '.join(p)  for  p  in  product('abc',  '123') ]
```

```
[ 'a1',  'a2',  'a3',  'b1',  'b2',  'b3',  'c1',  'c2',  'c3']
```

Пример Для чтобы продемонстрировать применение *product*, здесь находятся три постепенно сокращающихся и более понятных способов нахождения всех одно- или двухцифровых Пифагоровых троек(значения (x,y,z) удовлетворяющих $x^2 + y^2 = z^2$). Первый способ использует вложенные циклы for:

```
for  x  in  range(1, 101):
    for  y  in  range(1, 101):
        for  z  in  range(1, 101):
            if x**2 + y**2 == z**2
                print(x,y,z)
```

Ниже то же самое, переписанное с *product*:

```
X = range(1, 101)
for  (x,y,z)  in  product (X,X,X):
    if x**2+y**2==z**2:
        print(x,y,z)
```

Будет еще короче если мы применим генераторы списков:

```
X = range(1, 101)
[ (x,y,z)  for (x,y,z)  in  product(X,X,X)  if x**2+y**2==z**2 ]
```

24.3 Группирование объектов

Функция *groupby* удобна для группирования объектов. Она разбивает список на группы путем прохождения списка и всякий раз когда происходит изменение значений создаётся новая группа. Функция *groupby* возвращает упорядоченные пары, которые состоят из элемента списка и объекта итератора *groupby*, который состоит изгруппы элементов.

```
L = [ 0,  0,  1,  1,  1,  2,  0,  4,  4,  4,  4,  4 ]
for key, group  in  groupby(L):
    print(key, ':' , list(group))
```

```
0 : [ 0, 0 ]
1 : [1, 1, 1 ]
2 : [ 2 ]
0 : [ 0 ]
4 : [ 4, 4, 4, 4, 4 ]
```

Обратите внимание, что у нас получились две группы нулей. Это из-за того, что *groupby* возвращает новую группу каждый раз когда происходит изменение в списке. В примере выше, если мы просто, вместо этого захотели узнать количество обнаруженных чисел, то мы можем сначала отсортировать список, а затем вызвать *groupby*.

```
L = [ 0,  0,  1,  1,  1,  2,  0,  4,  4,  4,  4,  4 ]
L.sort()
for key, group  in  groupby(L):
    print(key, ':' , len)(list(group)))
```

```
0 : 3
1 : 3
2 : 1
4 : 5
```

Чаще всего, Вам будет нужно сортировать Ваши данные перед вызовом *groupby*.

Опциональный аргумент Функция *groupby* принимает опциональный аргумент, который является функций, говорящей ей как группировать объекты. При использовании этого, Вы обычно должны сначала отсортировать список с той функцией, как ключом сортировки. Ниже представлен пример, который группирует список слов по длине:

```
L = [ 'это',  'тест',  'функции',  'groupby' ]
L.sort(key = len)
for  key, group  in  groupby(L, len):
    print(key, ':', list(group))
```

```
3 : [ 'это ']
4 : [ 'тест ']
7 : [ 'функции ', 'groupby ']
```

Примеры Мы завершаем этот раздел двумя примерами.

Первый, предположим, что L является списком нулей и единиц, и нам нужно узнать какой длины самый длинный ряд единичек имеется. Мы можем выполнить это одной строкой используя *groupby*:

```
max([ len(list(group)) for  key, group  in, groupby(L)  if  key==1 ])
```

Второй, предположим, у нас имеется функция, называемая *easter*, которая возвращает дату Пасхи в заданный год. Следующий код создаст гистограмму того, какие даты встречаются чаще всего с 1900 по 2099 год.

```
L = [ easter(Y)  for  Y  in  range ]
L.sort()
for  key, group  in  groupby(L):
    print(key,  ':', ' * ' * (len(list(group))))
```

24.4 Разные вещи из *itertools*

chain Функция *chain* сцепляет итераторы вместе в один большой итератор. Например, если у Вас имеется три списка *L*, *M* и *N*, и нужно вывести все элементы каждого, один за другим, Вы можете сделать следующее:

```
for i in chain(L,M,N):
    print(i)
```

Как другой пример, в разделе 8.6, мы использовали генераторы списков для выравнивания списка списков, который заключается в возвращении списка со всеми элементами списков. Вот другой способ сделать это используя *chain*:

```
L = [ [ 1,2,3 ], [ 2,5,5 ], [ 7,8,3 ] ]
list(chain(*tuple(L)))
```

```
[ 1,  2,  3,  2,  5,  5,  7,  8,  3 ]
```

count Функция *count* ведет себя подобно **range**(∞). Она принимает опциональный аргумент, поэтому *count*(x) аналогично **range**(x, ∞).

cycle Функция *cycle* непрерывно выполняет циклические действия по элементам итератора. Когда она завершает цикл, то начинает его с начала и продолжает делать это непрерывно. Следуюший простой пример выводит числа от 0 до 4 постоянно, пока пользователь не введет 'n ':

```
for x in cycle(range(5)):
    z = input('Продолжать выполнять? да или нет : ')
    if z == 'да ':
        break
    print(x)
```

Больше о итераторах Существует ряд других функций в модуле *itertools*. Больше смотри документацию Python [1]. В ней содержится хорошая таблица, обобщающая, что выполняют различные функции.

24.5 Подсчет объектов

Модуль *collections* содержит полезный класс, называемый *Counter*. Вы снабжаете его итерируемым объектом и объект *Counter*, который создается, является чем-то очень сильно похожим на словарь, чьи ключи - это элементы из последовательности, а значениями - количество вхождений ключей в последовательности. В действительности, *Counter* является подклассом **dict**, класса словаря Python. Ниже пример:

```
Counter('ааббавабвгабвгд ')
```

```
Counter({ 'а ': 5,  'б ':4,  'в ': 3,  'г ': 2 ,  'д ':1 })
```

Так как *Counter* является подклассом **dict**, Вы можете получить доступ к элементам просто, как в словаре, и большинство из обычных методов словаря работают. Например:

```
c = Counter('ааббавабвгабвгд ')
c['а']
list(c.keys())
list(c.values())
```

```
5
[ 'а',  'б',  'в',  'г',  'д']
[ 5, 4, 3, 2, 1 ]
```

Получение самых часто встречаемых элементов Метод *most_common* принимает целое число *n* и возвращает список *n* самых часто встречаемых элементов, расположенных как (ключ, значение) кортежи. Например:

```
c = Counter('аабабвабвгабвгд ')
c.most_common(2)
```

```
[ ('а ', 5),  ('б ', 4) ]
```

Если мы пропустим аргумент, то она возвратит кортежи для каждого элемента в порядке с уменьшающей частотой. Чтобы получить элементы с наименьшим количеством, мы можем использовать срезы с конца списка возвращаемого *most_common*. Ниже несколько примеров:

```
c = Counter('аабабвабвгабвгд ')
c.most_common()
c.most_common()[ -2: ] c.most_common()[ -2::-1 ]
```

```
[ ('а', 5),  ('б',4),  ('в',3),  ('г',2),  ('д',1) ]
[ ('г',2),  ('д',1) ]
[ ('г',2),  ('в',3),  ('б',4),  ('а',5) ]
```

Последний пример использует отрицательный индекс среза, чтобы изменить порядок на обратный, от меньшей до наибольшей распространенности.

Пример Вот реально короткая программа, которая просканирует текстовой файл и создаст объект *Counter* повторяемости слов.

```
from collections import Counter
import re

s = open('filename.txt').read()
words = re.findall('\w+', s.lower())
c = Counter(words)
```

Чтобы вывести десять самых частых слов, мы можем сделать следующее:

```
for word, freq in c.most_common(10):
    print(word, ': ', freq)
```

Чтобы выбрать только те слова, которые встречаются более чем пять раз, мы можем сделать следующее:

```
[ word for word in c if c[ word]>5 ]
```

Математика со счетчиками Мы можем применить некоторые операторы на объекты *Counter*. Ниже представлены примеры:

```
c = Counter('ааббб')
d = Counter('абввв')
c + d
c - d
c & d
c | d
```

```
Counter({ 'б': 4,  'а': 3,  'в'})
Counter({ 'б': 2,  'а': 1})
Counter({ 'а': 1,  'б': 1})
Counter({ 'б': 3,  'в': 3,  'а': 2})
```

Выполняя c + d объединяем счетчики из *c* и *d*, в то время как c - d вычитает счетчики *d* из соответствующих счетчиков *c*. Обратите внимание, что *Counter*, возвращаемое c - d, не включает 0 или отрицательные счетчики. % обозначает пересечение(intersection) и возвращает минимум двух значений для кажлого элемента, а | отвечает за объединение и возвращает максимум двух значений.

24.6 defaultdict

Модуль *collections* содержит другой подобный словарю класс, называемый *defaultdict*. Он почти точно соответствует обычному словарю, за исключением, что когда Вы создаете новый ключ, то ему присваивается значение по умолчанию. Вот пример, который имитирует, то что делает класс *Counter*.

```
s = 'аабабвабвгабвг '
dd = defaultdict(int)
for  c  in  s:
    dd[ c ] += 1
```

```
defaultdict(<class 'int '>, { 'а':5,  'б':4,  'в':3,  'г':2 })
```

Если бы мы попробовали это с *dd* просто обычный словарь, то получили бы ошибку, когда программа первый раз достигает dd[c] += 1, потому что dd[c] ещё не существует. Но так как мы объявили *dd* равной *defaultdict*(**int**), каждому значению автоматически назначается значение 0 после создания и поэтому мы избегаем ошибок. Обратите внимание, что мы могли применить обычный словарь, если бы добавили инструкцию if в цикл, и имеется также функция обычных словарей, которая позволяет Вам установить значение по умолчанию, но *defaultdict* работает быстрее.

Мы можем использовать другие типы данных отличных от целых чисел. Далее пример со строками:

```
s = 'аабабвабвгабвг '
dd = defaultdict(str)
for  c  in  s:
    dd[ c ] += '* '
```

```
defaultdict(<class 'str '>, { 'а': *****,  'б':****,  'в':***,  'г':** })
```

Используйте **list** для списков, **set** для множеств, **dict** для словарей и **float** для чисел с плавающей точкой. Вы можете использовать различные другие классы также. Значение по умолчанию для целых чисел равняется 0, для списков [ ],для множеств **set**(), для словарей { } и для чисел с плавающей точкой 0.0. Если бы Вы захотели различные значения по умолчанию, то можете применить анонимную функцию, как внизу:

```
dd = defaultdict(lambda:100)
```

Выполнив с кодом из примера выше это создаст:

```
defaultdict(<class 'int '>, { 'a':105,  'б':104,  'в':103,  'г':102 })
```

Глава 25

Исключения

Эта глава предоставляет краткое введение в исключения.

25.1 Основы

Если Вы пишите программу, которую кто-то еще собирается использовать, и не хотите, чтобы произошел сбой, если случится ошибка. Скажем, Ваша программа выполняет группу вычислений, и в какой-то момент у Вас появляется строка c = a/b. Если b случится быть равной 0, то Вы получите ошибку деления на ноль и в программе будет сбой. Вот пример:

```
a = 3
b = 0
c = a/b
print('Всем привет ')
```

```
ZeroDivisionError: int division or modulo by zero
```

Как только произойдет ошибка, ничего из кода ниже c = a/b не будет выполняться. В действительности, если пользователь не запускает программу в IDLE или каком-либо другом редакторе, он даже не увидит ошибку. Программа просто остановится и вероятно закроется.

Когда ошибка случается, генерируется *исключение*. Вы можете *перехватить* это исключение и позволить Вашей программе восстановиться от ошибки без сбоя. Ниже пример:

```
a = 3
b = 0
try:
    c = a/b
except ZeroDivisionError:
    print('Ошибка вычисления ')
print'Привет всем '
```

```
Ошибка вычисления
Привет всем
```

Различные варианты У нас могут быть разные инструкции в блоке try, а также и различные блоки except, как внизу:

```
try:
    a = eval(input('Введите число :'))
    print(3/a)
except NameError:
    print('Пожалуйста введите число.')
except ZeroDivisionError:
    print("Не вводите 0.")
```

Не определяя исключение Вы можете пропустить имя исключения, как ниже:

```
try:
    a = eval(input('Введите число :'))
    print(3/a)
except:
    print('Возникла проблема.')
```

Как правило Вам не рекомендуется, однако, делать это, так как приведет к перехвату каждого исключения, включая те, которые Вы возможно не ожидаете, когда пишете код. Это приведет к трудностям при отладке Вашей программы.

Использование исключения Когда Вы перехватываете исключение, информация о нем хранится в объекте Exception. Ниже пример, который передает имя исключения пользователю:

```
try:
    c = a/0
exception Exception as e:
    print(e)
```

```
int division or modulo by zero
```

25.2 *try/except/else*

Вы можете использовать условие else вместе с try/except. Вот пример:

```
try:
    file = open('filename.txt ', 'r ')
except IOError:
     print('Не могу открыть файл')
else:
    s = file.read()
    print(s)
```

В этом примере, если *filename.txt* не существует, исключение input/output, называемое **IOError** сгенерируется. С другой стороны, если же файл имеется, то может быть определенное намерение, которое нам захочется сделать с ним. Мы можем вложить это в блок else.

25.3 *try/finally* и *with/as*

Существует еще один блок, который мы можем использовать, называемый finally. Код в блоке finally - это код, который должен быть выполнен, независимо, произошло исключение или нет, Поэтому если даже исключение произошло и Ваша программа дала сбой, инструкции в блоке finally выполнятся. Одной из них является закрытие файла. Если Вы просто размещаете закрытие файла после блока try, а не в блоке finally, то программа бы дала сбой прежде закрытия файла.

```
f = open('filename.txt ', 'w ')
s = 'привет'
try:
    # какой-либо код, который возможно может дать сбой здесь при выполнении
finally:
    f.close()
```

Блок **finally** может быть применен совместно с блоками **except** и **else**. Такая вещь с файлами являются достаточно обычной, чтобы она имела свой собственный синтаксис:

```
s = 'привет'
with open('filename.txt ')  as  f:
    print(s, file = f)
```

Это пример того, что называется *context manager (менеджером контекста)*. Менеджеры контекста и **try**/**finally** являются самыми используемыми в более сложных приложениях, как сетевое программирование.

25.4 Больше с исключениями

Имеется много больше, что может быть сделано с исключениями. Смотри документацию Python [1] о всех различных типах исключений. В действительности не все исключения выходят из ошибок. Существует даже инструкция, называемая **raise**, которую Вы можете использовать, чтобы поднять свои собственные исключения. Это полезно если Вы пишите свой собственный класс, который будет использован в других программах и Вы захотите послать сообщения, аналогично сообщениям об ошибке, к людям использующих Ваш класс.

ЧИТАТЬ ГРАММАТИКУ МАЛЕНЬКАЯ КНИЖКА

Литература

[1] Python документация. Доступно на www.python.org.
Потрясающая документация Python. Она красиво отформатирована, обширна, и ее легко найти.

[2] Lundh, Frederick. An Introduction to Tkinter. Доступно на www.effbot.org.
Это огромный справочник по Tkinter.