МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ»**

КАФЕДРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ И ПРОГРАММИРОВАНИЯ

**И. Г. ГНИДЕНКО
Д. Ю. ФЕДОРОВ**

ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

**ИЗДАТЕЛЬСТВО
САНКТ-ПЕТЕРБУРГСКОГО ГОСУДАРСТВЕННОГО
ЭКОНОМИЧЕСКОГО УНИВЕРСИТЕТА
2017**

УДК 004.4, 004.7
ББК 32.973-018.2
Г56

Гниденко И. Г.

Г56 Технологии и методы программирования : учебное пособие / И. Г. Гниденко, Д. Ю. Федоров. – СПб. : Изд-во СПбГЭУ, 2017. – 58 с.

ISBN 978-5-7310-3850-8

В учебном пособии представлены основы технологий создания программного обеспечения, типы и структуры данных, используемые в повседневной практике программирования, приведены алгоритмы решения наиболее распространенных классов задач.

Предназначено для направлений подготовки бакалавров 10.03.01 – «Информационная безопасность», 01.03.02 – «Прикладная математика и информатика», 09.03.02 – «Информационные системы и технологии», 38.03.05 – «Бизнес-информатика»; может представлять интерес для преподавателей смежных дисциплин.

The manual presents the basics of technologies for creating software, types, and data structures used in the daily practice of programming, given algorithms for solving the most common classes of problems.

The manual is intended for the bachelor degree program 10.03.01 – «Information Security», 01.03.02 – «Applied Mathematics and Informatics», 09.03.02 – «Information Systems and Technologies», 38.03.05 – «Business Informatics»; It may be of interest to teachers of related disciplines.

УДК 004.4, 004.7
ББК 32.973-018.2

Рецензенты: канд. экон. наук, доц. **О. Д. Мердина**
канд. экон. наук, доц. **И. В. Егорова**

ISBN 978-5-7310-3850-8

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Информационные технологии проникли практически во все сферы жизни современного общества. Трудно найти область деятельности, где не было бы потребности в обработке информации. Наступившая цифровая революция привела к генерации огромных объемов данных, обработка которых превышает когнитивные возможности человека. В связи с этим все большую роль будут занимать люди, чья профессия связна с разработкой и кодированием алгоритмов, позволяющих работать с информацией. Важным направлениям в подготовке таких специалистов является дисциплина «Технологии и методы программирования», в которой раскрываются основные подходы к созданию программного обеспечения.

Существуют различные представления о методах преподавания данной дисциплины. Классический подход опирается на изучение языков программирования C и C++, таким образом, студенты переходят от процедурной к объектно-ориентированной парадигме программирования так, как это было исторически. В данном учебном пособии, исходя из собственного педагогического опыта, авторами предложен иной подход: от объектно-ориентированной парадигмы языка Python к процедурной парадигме языка C.

Учебное пособие состоит из трех разделов.

В разделе 1 представлен обзор основных этапов развития технологий и методов программирования, начиная с неструктурированного подхода и завершая объектно-ориентированным и функциональным.

В разделе 2 на примере языка программирования Python в среде Jupyter рассмотрены основные типы и структуры данных, используемые в повседневной практике программирования, приведены алгоритмы решения наиболее распространенных классов задач.

В разделе 3 рассматривается интеграция языков программирования на примере написания Python-модуля на языке C.

Учебное пособие предназначено для направлений подготовки бакалавров 10.03.01 – «Информационная безопасность», 01.03.02 – «Прикладная математика и информатика», 09.03.02 – «Информационные системы и технологии», 38.03.05 – «Бизнес-информатика»; может представлять интерес для преподавателей смежных дисциплин.

РАЗДЕЛ 1. ЭВОЛЮЦИЯ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

«*... руководители, не имеющие представления об ЭВМ и программировании, уйдут в небытие, профессиональные программисты станут системными аналитиками и системными программистами, а программировать сумеет каждый, что я и называю второй грамотностью*»[1]

(1981 год, академик А.П. Ершов)

«*Информатика не более наука о компьютерах, чем астрономия – наука о телескопах*».

(Эдсгер Дейкстра)

«*Есть два типа языков программирования – те, которые все ругают, и те на которых никто не пишет*».

(Б. Страуструп, разработчик языка программирования C++)

1.1. Неструктурированное программирование

Технологией программирования называется совокупность методов и средств, используемых в процессе разработки программного обеспечения. Технология программирования представляет собой последовательность технологических операций (этапов программирования) с описанием операций (исходные данные и результаты) и условиями их выполнения.

До середины 60-ых годов существовала *неструктурированная*, «стихийная» технология программирования. Структура первых простейших программ состояла собственно из программы, написанной на машинном языке (двоичных или шестнадцатеричных кодов) и обрабатываемых ею данных. Появление машинно-ориентированных языков (ассемблеров) позволило программистам вместо кодов использовать мнемонические обозначения и символические имена данных. Программы стали «читаемыми».

Затем появились *языки программирования высокого уровня* (FORTRAN, ALGOL), позволившие снизить уровень детализации операций. Большим достижение этих языков стало наличие средств использования подпрограмм. Теперь структура программы состояла из основной программы, области глобальных данных и набора подпрограмм. Недостаток такой структуры – возрастание вероятности искажения части глобальных данных какой-либо подпрограммой при увеличении количества под-

[1] Ершов А.П. Программирование – вторая грамотность [Электронный ресурс]. URL: http://www.computer-museum.ru/books/n_ershov/2_ershov_prog.htm (дата обращения: 05.03.2017)

программ. Для сокращения таких ошибок было предложено размещать в подпрограммах локальные данные. Появилась возможность осуществлять разработку программного обеспечения несколькими программистами параллельно.

В 60-ых годах 20-го века при разработке сложного программного обеспечения (например, операционных систем) стали срываться сроки разработки. Разразился «кризис программирования». Причина – несовершенство «стихийного» программирования. Процесс тестирования и отладки занимал 80% времени разработки программного обеспечения. Стоимость аппаратных средств снижалась, а стоимость разработки программного обеспечения все время росла из-за того, что создавались все более мощные и сложные прикладные программы при отставании технологии их разработки.

1.2. Процедурное и модульное программирование

В результате исследовательских работ 60-70-ых годов XX века была разработана технология процедурного (структурного, модульного программирования), внесшая ясность в написание программ. Технология процедурного программирования – это дисциплинированный подход к написанию программ по сравнению с неструктурированными программами. Процедурное программирование основано на модели построения программы как иерархии процедур, что и дало название данной технологии.

Для изучения процедурного программирования в 1971 году Н.Виртом был создан язык программирования Pascal, нашедший большое применение в университетах. На протяжении 70-ых годов создавался язык С на базе концепций предшествующих языков – BCPL и B, разработанных для написания компиляторов и операционных систем. Язык С получил широкую популярность в результате его использования в разработке операционной системы UNIX. В конце 70-ых был создан «классический» язык Б.Кернигана и Д.Ритчи (K&R).

Технология процедурного программирования основана на использовании следующих методов (приемов) программирования:

- Метод декомпозиции (нисходящего проектирования), т.е. разделение программы на процедуры простейшей структуры и представление программы в виде иерархии процедур.
- Метод модульной организации, т.е. группировка процедур и обрабатываемых ими данных в модули, которые программируются и компилируются отдельно.
- Метод структурного программирования процедур, который заключается в следующем:
    - разделение процедур на вложенные блоки;

- использование операторов ветвления и циклов;
- форматирование текста процедуры.

Наиболее известными процедурными языками программирования являются ALGOL-68, Pascal, C.

1.3. Объектно-ориентированное программирование

Проектирование и реализация информационных систем в экономике привели к появлению больших по объему программ. Программы не всегда объективно отражали объекты реального мира, и поэтому не могли быть повторно используемыми. В каждом новом проекте приходилось программировать все сначала и получать похожие программы, т.е. каждый новый проект трактовался как новая задача. А хотелось бы считать новый проект как расширение чего-то созданного. Обнаружились сложности сопровождения и модификации больших программ. Процедурное программирование не обеспечивало в достаточной степени абстракции данных.

В начале 80-ых годов Б.Страуструпом был разработан язык C++, обеспечивший возможность *объектно-ориентированного подхода* к программированию. Язык C++ был построен на базе двух языков – C и Simula 67, языке программного моделирования, разработанного в Европе. Имелись и другие объектно-ориентированные языки, и наиболее известный язык Smalltalk, который является чистым объектно-ориентированным языком.

В 1991 году нидерландским программистом Гвидо ван Россумом был разработан язык Python, включающий в себя как процедурные, так и объектно-ориентированные возможности.

В 1995 году фирмой Sun Microsystems был разработан на основе языков C и C++ язык Java, используемый для создания интерактивных Web-страниц и в разработке приложений на базе Internet и Intranet.

Основными методами (понятиями) объектно-ориентированного программирования являются механизмы *инкапсуляции*, *наследования* и *полиморфизма*. Объектно-ориентированное программирование принципиально отличается от процедурного программирования, основывается на понятиях прикладной области, являющихся классами. Программы строятся как иерархия классов.

На основе объектно-ориентированного подхода были созданы среды программирования, например Delphi, C++ Builder, Visual C++ и др., реализующие визуальное программирование.

1.4. Перспективы развития технологий программирования

Развитием событийно управляемой концепции объектно-ориентированного подхода стало появление в 90-х годах целого класса языков программирования, которые получили название языков сцена-

риев или скриптов. В рамках данного подхода программа представляет собой совокупность возможных сценариев обработки данных, выбор которых инициируется наступлением того или иного события (щелчок по кнопке мыши, попадание курсора в определенную позицию, изменение атрибутов того или иного объекта, переполнение буфера памяти и т.д.). События могут инициироваться как операционной системой, так и пользователем. Существенным преимуществом языков сценариев является их совместимость с передовыми инструментальными средствами автоматизированного проектирования и быстрой реализации программного обеспечения, или так называемыми CASE- (Computer-Aided Software Engineering) и RAD- (Rapid Application Development) средствами.

Одним из наиболее передовых инструментальных комплексов, предназначенных для быстрой разработки приложений, является Microsoft Visual Studio .NET. Характерные примеры сценарных языков программирования: VBScript, PowerScript, LotusScript, JavaScript.

Еще один важный этап в развитии технологий программирования – это применение *языков поддержки параллельных вычислений*. Программы, написанные на этих языках, представляют собой совокупность описаний процессов, которые могут выполняться как в действительности одновременно, так и в псевдопараллельном режиме. Языки параллельных вычислений позволяют достичь заметного выигрыша при обработке больших массивов информации, поступающих от одновременно работающих пользователей, либо имеющих высокую интенсивность (как, например, видеоинформация или звуковые данные высокого качества). Примерами языков программирования с поддержкой параллельных вычислений могут служить Erlang, Modula-3 и др.

Помимо процедурного и объектно-ориентированного подходов набирает популярность функциональный подход в программировании. По точному определению академика А.П. Ершова: «Функциональное программирование – это способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значение функции, а единственным правилом композиции – оператор суперпозиции функции. Никаких ячеек памяти, ни операторов присваивания, ни циклов, ни, тем более, блок-схем, ни передач управления» [1]. Функциональный подход представлен такими языками как Racket, Clojure, F#, Haskell и др.

1.5. Этапы разработки программ

«*Алгоритмы + структуры данных = программы*[2]»
(Н. Вирт, автор языка программирования Pascal)

На первом шаге программист обладает набором «сырых» данных (рис. 1): разрозненными бухгалтерскими отчетами, статистикой и пр.

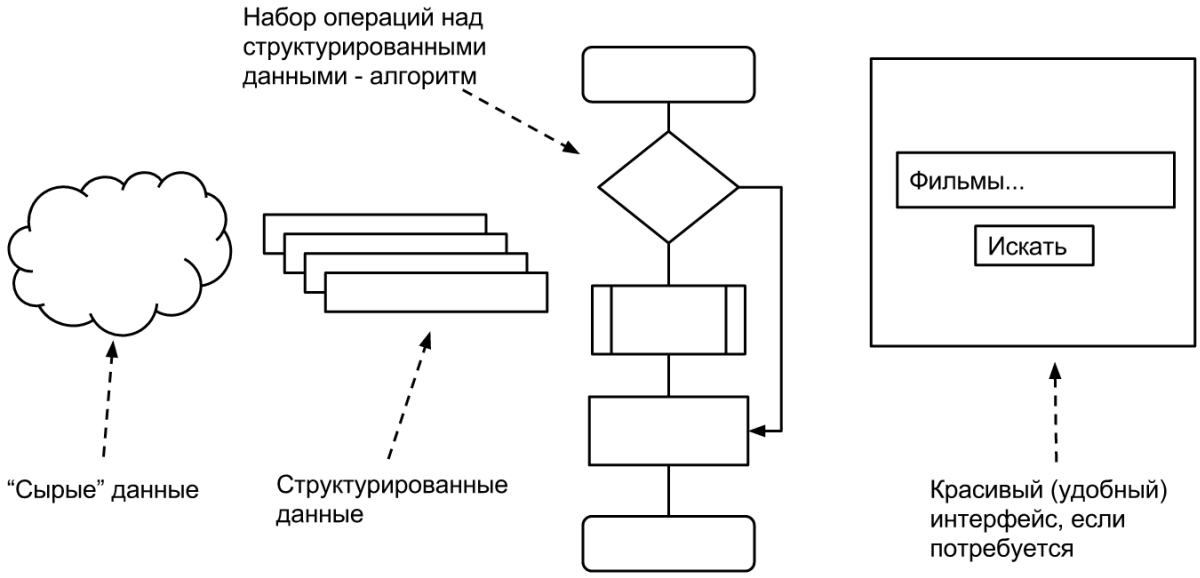


Рис. 1. Основные шаги при создании программы

Эти сведения необходимо структурировать и загрузить для обработки в компьютер. Далее, программист создает (выбирает) алгоритм, т.е. набор конечных действий для обработки структурированных данных, исходя из поставленной задачи. Отметим, что правильный выбор структуры данных влияет на создание (выбор) алгоритма. Сила языка программирования отчасти заключена в структурах данных, которое он предоставляет для работы. После того, как алгоритм разработан, и программа работает (в результате ее работы получается корректный ответ), можно переходить к созданию интерфейса для диалога с пользователем.

[2] Вирт Н. Алгоритмы + Структуры данных = Программы. – М. : Мир, 1985.

РАЗДЕЛ 2. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON

2.1. Выполнение программ на языке Python и среда разработки Jupyter

Выполнение программ осуществляется операционной системой (Microsoft Windows, GNU/Linux и пр.). В задачи операционной системы (ОС) входит распределение ресурсов (оперативной памяти и пр.) для программы, запрет или разрешение на доступ к устройствам ввода/вывода и. т. д. Для запуска программ, написанных на языке программирования Python, понадобится программа-интерпретатор[3] – виртуальная машина Python[4] (рис. 2).

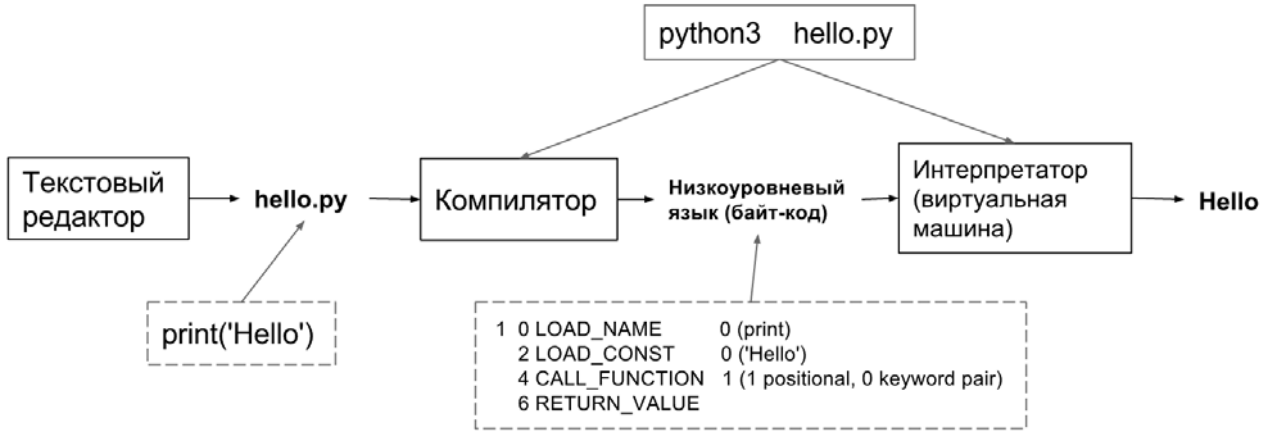


Рис. 2. Схема запуска программ

Виртуальная машина скрывает от Python-программиста особенности ОС, поэтому, написав программу в системе Windows, ее можно запустить, например, в GNU/Linux и получить схожий результат[5].

В качестве среды разработки будем использовать Jupyter[6] – развитие проекта IPython[7], который в интерактивном режиме по средствам веб-интерфейса позволяет запускать программы на языке Python 3. Jupyter (рис. 3) в отличие от IPython включает в себя не только интерпретатор языка Python, но и поддержку таких языков как Scala, Bash, Haskell, Julia, R, Ruby. Выполнить тестовый запуск полноценной версии Jupyter можно на сайте: https://try.jupyter.org.

---

[3] Python является интерпретируемым языком программирования (команды выполняются шаг за шагом).

[4] Далее рассматривается CPython – классическая реализация на языке C.

[5] При условии, что не производится обращение к специфическим возможностям отдельной операционной системы.

[6] Официальный сайт: http://jupyter.org.
Документация Jupyter: https://jupyter.readthedocs.io/en/latest/

[7] Проект, основанный в 2001 году, как усовершенствованный интерактивный интерпретатор Python.

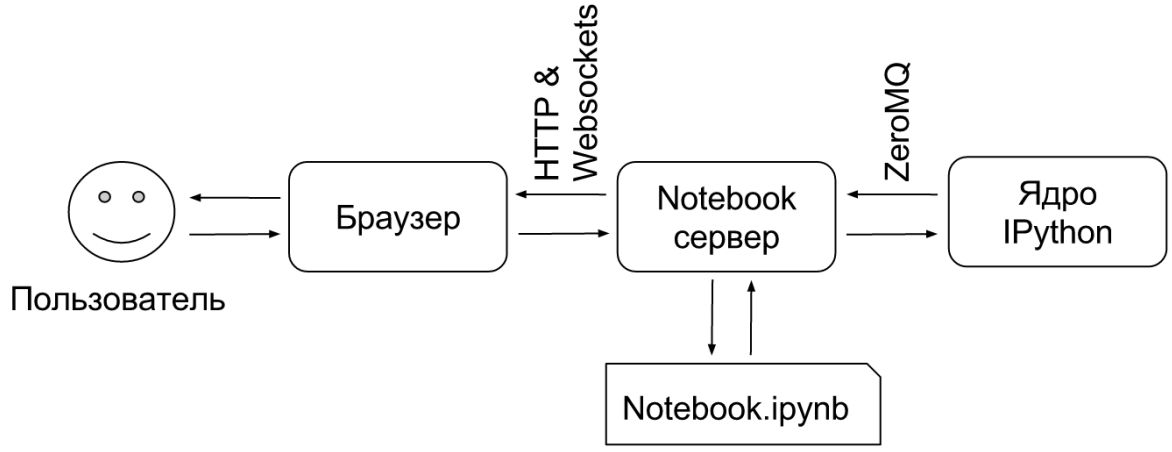


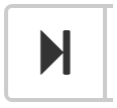
Рис. 3. Архитектура Jupyter на основе ядра IPython

Для установки Jupyter под ОС Windows[8] понадобится скачать и распаковать дистрибутив WinPython[9]. После установки запустите Jupyter Notebook: создастся локальный веб-сервер, прослушивающий сетевой порт с номером 8888, далее автоматически на странице http://localhost:8888/tree откроется веб-браузер. Создайте новый блокнот (Notebook) для дальнейшего запуска программ на языке Python 3: New → Python 3. Откроется веб-интерфейс с возможностью набора команд в строке In []. Переименуйте блокнот (File → Rename) в MyTest. Убедитесь, что в каталоге \notebooks\ появился файл с именем MyTest.ipynb[10].

2.2. Простой калькулятор

В самом начале обучения Python можно рассматривать как интерактивный калькулятор. Вычислим значения математических выражений. Для этого в ячейке In [] блокнота[11] Jupyter наберите:

```
3.6 + 8
```

Для выполнения ячейки и выделения следующей нажмите 

(сочетание клавиш <Shift> + <Enter> или в меню: Cell → Run Cells and Select Below). Результат отобразится в ячейке Out [] с индексом 1 (рис. 4).

---

[8] Для пользователей ОС GNU/Linux: pip3 install jupyter и jupyter notebook

[9] Официальный сайт: https://winpython.github.io. Еще один вариант установки – Anaconda: https://www.continuum.io

[10] Это файл в формате JSON.

[11] Пользователям программы Wolfram Mathematica знакомы пронумерованные строки.

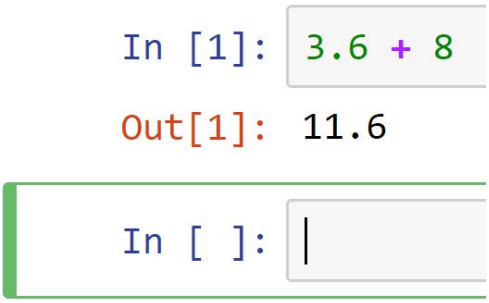


Рис. 4. Результат вычисления суммы чисел

Аналогично (каждое в отдельной ячейке) найдите значение следующих выражений:

```
4 + 9
1 - 5
_ + 6
```

Нижним подчеркиванием в предыдущем примере обозначается последний полученный результат. Если совершить ошибку при вводе команды:

```
In [5]:  a
```

интерпретатор сообщит:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-60b725f10c9c> in <module>()
----> 1 a

NameError: name 'a' is not defined
```

В математических выражениях в качестве операндов могут использоваться как *целые числа*[12] (например, 1, 4, -5) так и *вещественные*[13] (*числа с плавающей точкой*): 4.111, -9.3. Математические операции, доступные над числами в Python[14]:

---

[12] А также комплексные числа и логические значения: True, False.

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

```
In  [1]:  2/3 + 1
Out [1]:  1.6666666666666665
In  [2]:  5/3
Out [2]:  1.6666666666666667
```

[14] В Python выражение (b * (a // b) + a % b) эквивалентно a.

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

Таким образом, рассмотрели *числовой тип данных* (целочисленный тип int и вещественный тип float), т.е. множество числовых значений и множество математических операций, которые можно выполнять над данными значениям.

На языке Python вычислить значение y = x + 3 * 6 при x равном 1 можно следующим образом:

```
In  [6]: x = 1
         y = x + 3*6
         y
Out [6]: 19
```

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

Имена переменным задает программист, но есть несколько ограничений: нельзя начинать с цифры и использовать ключевые слова, которые для Python имеют определенный смысл (эти слова подсвечиваются в Jupyter зеленым цветом):

and	as	assert	break	class	Continue
def	del	elif	else	except	Exec
finally	for	from	global	if	Import
In	is	lambda	nonlocal	not	Or
pass	raise	return	try	while	With
yield	True	False	None		

Рассмотрим формулу перевода из шкалы в градусах по Цельсию (T_C) в шкалу в градусах по Фаренгейту (T_F):

$$T_F = \frac{9}{5} T_C + 32$$

Определим значение T_F при T_C равном 26:

```
In    [7]:  cel = 26
            cel
Out [7]:  26
In    [8]:  9/5 * cel + 32
Out [8]:  78.80000000000001
```

В момент выполнения инструкции присваивания cel = 26 в памяти компьютера создается *объект* (рис. 5), расположенный по некоторому *адресу* (условно обозначим его как *id*1), имеющий значение 26 целочисленного типа int. Затем создается переменная с именем cel, которой *присваивается адрес* объекта *id*1. Таким образом, переменные в Python содержат адреса объектов или *переменные ссылаются на объекты*. Тип переменной определяется типом объекта, на который он ссылается.

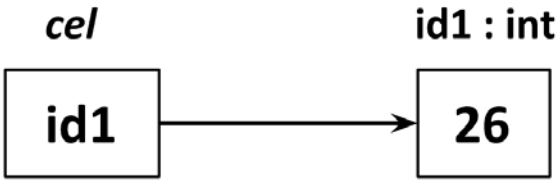


Рис. 5. Модель памяти для выражения cel = 26

Jupyter позволят проводить *интроспекцию объекта*, т.е. выводить общую информацию о нем:

```
In    [9]:  cel?
```

В дополнительном окне появится:

```
Type:        int
String form: 26
Docstring:
int(x=0) -> integer
int(x, base=10) -> integer
```

В Python числовые объекты являются *неизменяемыми*, поэтому при изменении содержимого переменной: создается новый числовой объект, а предыдущий удалятся[15] из памяти с помощью автоматической *сборки мусора*.

Функцией в программировании называется последовательность инструкций, которая выполняет вычисления. Рассмотрим встроенную

[15] Если на него не ссылаются другие переменные.

функцию с именем abs, принимающую на вход один аргумент – объект числового типа и возвращающую абсолютное значение для этого объекта:

```
In  [10]: abs(-9)
Out [10]: 9
```

На практике при написании программ требуется преобразовывать типы объектов. Функция int принимает значение и преобразует его в целое число, если это возможно (возвращает 0, если аргумент не передан). Может преобразовать число с плавающей точкой в целое через отсечение дробной части:

```
In  [11]: int(5.6)
Out [11]: 5
```

Функция float возвращает число с плавающей точкой, построенное из числа или строки[16] (возвращает 0.0, если аргументы не переданы).
Функция help отображает документацию[17]:

```
In [13]: help(abs)

Help on built-in function abs in module builtins:

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

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

$$T_C = \frac{5}{9}(T_F - 32)$$

Создадим собственную функцию, переводящую градусы по шкале Фаренгейта в градусы по шкале Цельсия:

```
In  [19]: def convert_co_cels(fahren):
              return (fahren-32)*5/9
```

Ключевое слово def означает, что далее последует определение функции. После def указывается имя функции convert_co_cels, затем в скобках указывается *параметр* (или параметры через запятую), которому будет присваиваться значение при вызове функции. Параметры функции – обычные переменные, которыми функция пользуется для

[16] О строках речь пойдет далее.
[17] Или воспользоваться возможностью интроспекции в Jupyter: abs?

внутренних вычислений. Переменные, объявленные внутри функции, называются *локальными* и не видны вне функции. После символа «:» начинается *тело функции* – блок команд, относящийся к функции. Тело функции может содержать любое количество инструкций. Jupyter самостоятельно расставит отступы[18] от края ячейки, тем самым обозначив, где начинается тело функции. Выражение, стоящее после инструкции return будет возвращаться в качестве результата вызова функции. После того как функция создана, ее можно вызвать, подставив в скобках фактические аргументы:

```
In  [20]: convert_co_cels(451)
Out [20]: 232.77777777777777
```

Имена функций в Python являются обычными переменными, содержащими адрес объекта[19] типа функция[20], поэтому этот адрес можно присвоить другой переменной и вызвать функцию уже с другим именем (в отдельной ячейке Jupyter):

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

2.3. Строки

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

```
In  [1]: 'hello'
Out [1]: 'hello'
```

Для работы со строками предусмотрено большое количество встроенных функций. Например, функция len определяет длину строки, переданной ей в качестве аргумента:

```
In  [2]: len('Привет!')
Out [2]: 7
```

---

[18] Отступы играют важную роль в Python, отделяя блок команд тела функции, цикла и пр. см. правила оформления исходного кода на Python в PEP8: https://www.python.org/dev/peps/pep-0008/

[19] В Python все является объектами.

[20] Да, да, это еще один тип данных.

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

```
In  [3]: 'Привет, ' + 'земляне!'
Out [3]: 'Привет, земляне!'
```

Попросим Python повторить (размножить) строку 3 раза:

```
In  [4]: "СПАМ" * 3
Out [4]: 'СПАМСПАМСПАМ'
```

Строки, по аналогии с числами, можно присваивать[21] переменным:

```
In  [5]: s = "Python"
In  [6]: s
Out [6]: 'Python'
In  [7]: s*4
Out [7]: 'PythonPythonPythonPython'
```

Попросим пользователя ввести значение с клавиатуры (через вызов функции input):

```
In  [8]: s = input()
```

После выполнения команды появится форма для ввода текста:

```
Земляне, мы прилетели с миром!
In   [9]: s
Out  [9]: 'Земляне, мы прилетели с миром!'
In  [10]: type(s)
Out [10]: str     # функция input возвращает строку!
```

В примере была вызвана функция input, результат ее выполнения присвоен переменной s. Содержимое переменной s вывели на экран – отобразилась фраза, введенная пользователем. Затем вызвали функцию type, которая позволяет определить тип объекта, ссылка на который содержится в переменной s.

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

---

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

```
In    [11]:  s = 'Я люблю писать программы!'
In    [12]:  s[0]
Out [12]:  'Я'
In    [13]:  s[-1]
Out [13]:  '!'
```

В квадратных скобках указывается индекс символа. Строки, как и числовые объекты, являются *неизменяемыми*. Прежде чем поймем, как модифицировать строку, познакомимся со *срезами*:

```
In    [14]:  s = 'Питоны водятся в Африке'
In    [15]:  s[1:3]
Out [15]:  'ит'
```

s[1:3] – срез (часть) строки s, начиная с индекса 1, заканчивая индексом 3 (не включительно). Теперь «изменим» первый символ в строке через создание новой строки:

```
In    [16]:  s = 'Я люблю писать программы!'
In    [16]:  'J' + s[1:]    # формируется новая строка
Out [17]:  'J люблю писать программы!'
```

2.4. Операторы сравнения и инструкция if

В Python для сравнения чисел предусмотрено несколько операторов сравнения:

>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
==	Равно
!=	Не равно

Произведем сравнение двух чисел:

```
In    [1]:  6 > 5
Out [1]:  True
In    [2]:  7 < 1
Out [2]:  False
In    [3]:  7 == 7      # не путайте == и = (присвоить)
Out [3]:  True
In    [4]:  7 != 7
Out [4]:  False
```

Python возвращает True[22] (истину[23]), когда сравнение верное и False (ложь) – в ином случае. True и False относятся к *логическому* (*булевому*) типу данных bool:

```
In  [1]: type(True)
Out [1]: bool
```

Для Python истинным или ложным может быть не только логическое высказывание, но и объект. Любое число, не равное нулю, или непустой объект интерпретируется как истина. Числа, равные нулю, пустые объекты и специальный объект None[24] интерпретируются как ложь.

Рассмотрим более подробно три логических оператора: and, or, not. Отрицание not (не) наиболее простой из них:

```
In  [1]: y = 6 > 8
In  [2]: y
Out [2]: False
In  [3]: not y
Out [3]: True
```

В результате применения логического оператора and (и) получим True или False, если его операндами являются логические высказывания:

```
In  [1]: 2 > 4 and 45 > 3  # комбинация False and True вернет False
Out [1]: False
```

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

```
In  [1]: 0 and 3 # вернет первый ложный объект-операнд
Out [1]: 0
In  [2]: 5 and 4 # вернет крайний правый объект-операнд
Out [2]: 4
```

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

Логический оператор or действует похожим образом, но для объектов-операндов Python возвращает первый объект, имеющий истинное зна-

---

[22] Обязательно с большой буквы.
[23] True интерпретируется Python как число 1, а False как число 0.
[24] Имеет специальный тип NoneType.

чение. Python прекратит дальнейшие вычисления, как только будет найден первый объект, имеющий истинное значение:

```
In  [1]:  2 or 3     # вернет первый истинный объект-операнд
Out [1]:  2
In  [2]:  None or 5
Out [2]:  5
In  [3]:  None or 0   # вернет оставшийся крайний объект-операнд
Out [3]:  0
```

Следующий полезный логический оператор – in. Он принимает две строки и возвращает True, если первая строка является подстрокой второй строки:

```
In  [1]:  'a' in 'abc'
Out [1]:  True
```

Наиболее часто логические выражения используются внутри условной инструкции if. В следующем листинге представлен пример программы, использующей условные инструкции[25]:

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

Контрольные задания

1. Напишите программу, определяющую максимальное из двух введенных чисел. Реализовать в виде вызова собственной функции, возвращающей большее из двух переданных ей чисел.
2. Напишите программу, проверяющую целое число на четность. Реализовать в виде вызова собственной функции.
3. Напишите программу, которая по коду города и длительности переговоров вычисляет их стоимость. Результат выводится на экран. Та-

[25] При наборе исходного текста следите за отступами

рифы: Екатеринбург – код 343, 15 руб/мин; Омск – код 381, 18 руб/мин; Воронеж – код 473, 13 руб/мин; Ярославль – код 485, 11 руб/мин.

2.5. Подключение модулей

Предположим, что имеется несколько полезных функций, которые часто используются в программах. Эти функции удобно поместить в отдельный файл и загружать (вызывать) оттуда. В Python такие файлы называются *модулями*. Для того чтобы воспользоваться функциями, находящимися в отдельном модуле, его необходимо *импортировать* с помощью инструкции import:

```
In [1]: import math
```

Команда загружает в память стандартный модуль math[26] (содержит набор математических функций), теперь можем обращаться к функциям, находящимся внутри этого модуля следующим образом (например, для нахождения квадратного корня[27] из 9):

```
In  [2]: math.sqrt(9)
Out [2]: 3.0
```

Указываем имя модуля, точку и имя функции с аргументами[28]. В момент импортирования модуля math создается переменная с именем math, которая является объектом типа module. Можно импортировать отдельную функцию из модуля:

```
In  [3]: from math import sqrt
In  [4]: sqrt(9)
Out [4]: 3.0
```

В этом случае переменная math создана не будет, а в память загрузится только функция sqrt.

Ранее для нахождения модуля числа вызывали функцию abs. Эта функция находится в модуле __builtins__ (два нижних подчеркивания до и после имени модуля[29]), который загружается в память в момент начала работы интерпретатора.

---

[26] см. help(math)
[27] см. help(math.sqrt)
[28] Jupyter позволяет подсказывать пользователю список методов: math.**<Tab>**
[29] см. help(__builtins__) и dir(__builtins__)

2.6. Строковые методы

Тип данных в Python описывается *классом*. Для упрощения представим класс, как аналог модуля, т.е. набор функций и переменных, содержащихся внутри класса. Функции, которые находятся внутри класса, называются *методами*. Их главное отличие от вызова функций из модуля заключается в том, что в качестве первого аргумента метод принимает, например, строковый объект, если это метод строкового класса:

```
In  [1]: str.capitalize('hello')
Out [1]: 'Hello'
```

По аналогии с вызовом функции из модуля указывается имя класса str, затем через точку – имя строкового метода capitalize, который принимает один строковый аргумент.

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

```
In  [2]: 'hello'.capitalize()
Out [2]: 'Hello'
```

Первый аргумент метода поместили на место имени класса[30]. Получение справки для метода производится путем вызова функции help, но вместо имени модуля указывается имя класса: help (**str**.capitalize)

Python содержит строковые методы, которые возвращают True или False. Например, строковый метод startswith проверяет, начинается ли строка с символа, переданного в качестве аргумента методу[31]:

```
In  [3]: 'spec'.startswith('a')
Out [3]: False
```

Предположим, что переменная s содержит некоторую строку, тогда к ней применимы следующие методы[32]:

```
s.upper() – возвращает строку в верхнем регистре
s.lower() – возвращает строку в нижнем регистре
s.title() – возвращает строку, первый символ которой в верхнем регистре
s.find('вет', 2, 3) – возвращает позицию подстроки в интервале либо -1
s.count('e', 1, 5) – возвращает количество подстрок в интервале либо -1
s.isalpha() – проверяет, состоит ли строка только из букв
```

[30] Python всегда преобразует сокращенную форму в аналогичную ей полную форму.
[31] см. https://docs.python.org/3.6/library/stdtypes.html#str.startswith
[32] см. https://docs.python.org/3/library/stdtypes.html#string-methods

s.isdigit() – проверяет, состоит ли строка только из чисел
s.isupper() – проверяет, написаны ли все символы в верхнем регистре
s.islower() – проверяет, написаны ли все символы в нижнем регистре
s.istitle() – проверяет, начинается ли строка с большой буквы
s.isspace() – проверяет, состоит ли строка только из пробелов

2.7. Списки

Списки (list) являются аналогом массива в других языках программирования, за исключением важной особенности – списки в качестве своих элементов могут содержать объекты различных типов. Список может быть присвоен переменной (переменной присваивается адрес объекта типа список):

```
In  [1]: e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
In  [2]: e
Out [2]: [56.806, 57.1578, 57.4093, 56.1843, 57.2207]
In  [3]: e?
Type:          list
String form: [56.806, 57.1578, 57.4093, 56.1843, 57.2207]
Length:        5
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Список позволяет хранить разнородные данные, обращаться к которым можно через имя списка (в данном случае переменную e). Рассмотрим схематично работу Python со списками (рис. 6).

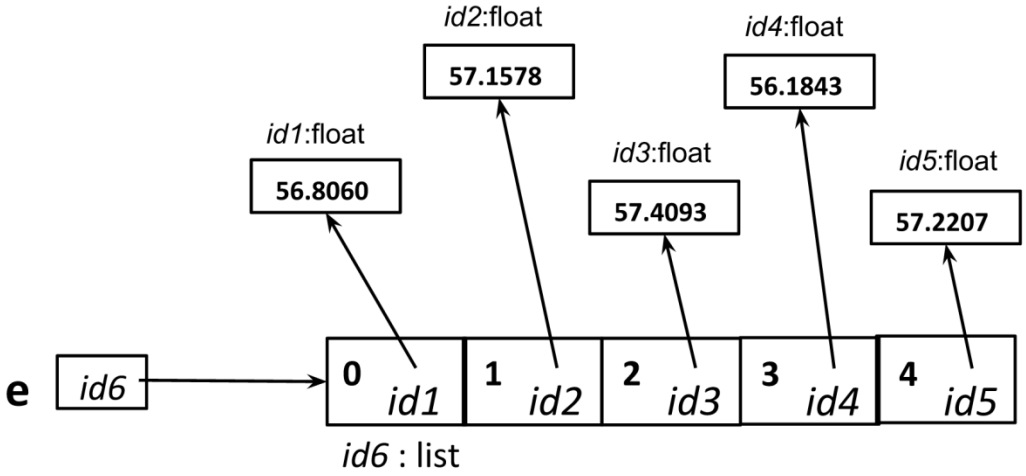


Рис. 6. Схема памяти Python при работе со списком

Переменная e содержит адрес списка (*id*6), каждый элемент списка содержит адрес соответствующего объекта.

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

```
In    [4]:  e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
In    [5]:  e[0]
Out [5]:  56.806
```

Список относится к *изменяемому типу данных*:

```
In    [6]: h = ['Hi', 27 , -8.1, [1,2]]   # создание списка
In    [7]: h[1] = 'hello' # изменение элемента списка в позиции 1
In    [8]: h                # результирующий список
Out [8]: ['Hi', 'hello', -8.1, [1, 2]]
```

Python содержит большое число встроенных функций, позволяющих легко и быстро обрабатывать списки:

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

Инструкция del удаляет элементы из списка по указанному индексу:

```
In    [9]:   h = ['Hi', 27 , -8.1, [1,2]]
In    [10]: del h[0]
In    [11]: h
Out [11]: ['hello', -8.1, [1, 2]]
```

Работа с методами списка[33] похожа на работу со строковыми методами[34]:

```
In    [1]: colors = ['red', 'orange', 'green']
In    [2]: colors.extend(['black','blue'])   # расширяет список
In    [3]: colors
Out [3]: ['red', 'orange', 'green', 'black', 'blue']
In    [4]: colors.append('purple')      # добавляет элемент в список
In    [5]: colors
Out [5]: ['red', 'orange', 'green', 'black', 'blue', 'purple']
```

На практике довольно часто возникает необходимость в изменении строк. Решением может стать преобразование строки в список, изменение списка и обратное преобразование списка в строку:

---

[33] Подробнее: https://docs.python.org/3/tutorial/datastructures.html#more-on-lists
[34] Только вместо класса str в полной форме используется list, а в качестве первого аргумента метод принимает объект типа список.

```
In    [1]: s = 'Строка для изменения'
In    [2]: list(s)  # функция list пытается преобразовать аргумент в список
Out [2]: ['С', 'т', 'р', 'о', 'к', 'а', ' ', 'д', 'л', 'я', ' ', 'и', 'з', 'м', 'е', 'н', 'е', 'н', 'и', 'я']
In    [3]: lst = list(s)
In    [4]: lst[0] = 'М'  # изменяем список, полученный из строки
In    [5]: lst
Out [5]: ['М', 'т', 'р', 'о', 'к', 'а', ' ', 'д', 'л', 'я', ' ', 'и', 'з', 'м', 'е', 'н', 'е', 'н', 'и', 'я']
In    [6]: s = ''.join(lst)  # преобразуем список в строку
In    [7]: s
Out [7]: 'Мтрока для изменения'
```

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

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

```
In    [8]: n = 73485384753846538465
In    [9]: list(str(n)) # число преобразуем в строку, затем строку в список
Out [9]: ['7', '3', '4', '8', '5', '3', '8', '4', '7', '5', '3', '8', '4', '6', '5', '3', '8', '4', '6', '5']
```

Если строка содержит разделитель (пробел), то ее легко преобразовать к списку с помощью строкового метода split, который по умолчанию в качестве разделителя использует пробел:

```
In    [10]: s = 'd a dd dd gg rr tt yy rr ee'.split()
In    [11]: s
Out [11]: ['d', 'a', 'dd', 'dd', 'gg', 'rr', 'tt', 'yy', 'rr', 'ee']
```

2.8. Итерации

Язык программирования Python позволяет в кратчайшие сроки создавать прототипы реальных приложений благодаря тому, что в него заложены конструкции для решения типовых задач. К примеру, вывести на экран каждый из элементов списка можно с помощью инструкции *цикла for:*

```
In [1]: num = [0.8, 7.0, 6.8, -6]
In [2]: for i in num:
            print(i, '- number')

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

Инструкция цикла for позволяет обратиться к каждому из элементов указанного списка. Имя переменной, в которую на каждом шаге будет помещаться элемент списка, выбирает программист. На первом шаге переменной i присваивается первый элемент списка num, равный 0.8. Затем программа переходит в тело цикла for, отделенное отступами (четыре пробела). В теле цикла содержится вызов функции print, которой передается на вход переменная i. На следующем шаге (итерации цикла) переменной i присвоится второй элемент списка, равный 7.0. Произойдет вызов функции print для отображения текущего содержимого переменной i на экране и т.д. тело цикла выполняется до тех пор, пока не достигнет конца списка.

Инструкция for схожим образом работает для строк: происходит обращение к каждому из символов, пока не достигнет конца строки.

Достаточно часто на практике при разработке программ необходимо получить *диапазон* целых чисел, для этого предусмотрена функция range[35]. В качестве аргументов она принимает начальное значение диапазона (по умолчанию 0), конечное значение (*не включительно*) и шаг (по умолчанию 1). Если вызвать функцию range, то диапазон чисел не увидим[36]:

```
In  [3]: range(0, 10, 1)
Out [3]: range(0, 10)
In  [4]: range(10)
Out [4]: range(0, 10)
```

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

```
In [5]: for i in range(2, 20, 2):
            print (i, end=' ')

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

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

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

```
In  [6]:  sum(list(range(1, 101)))
Out [6]:  5050
```

---

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

[36] Функция range вернет объект типа range, т.к. для экономии места она хранит в памяти только начало, окончания и шаг диапазона.

Рассмотрим пример создания списка с помощью метода append:

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

Инструкция for последовательно из диапазона от 1 до 14 выбирает числа и с помощью метода append, находящегося в теле цикла, добавляет их к списку a. Следующий пример – создание списка из диапазона с помощью функции list:

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

Далее показано использование «*спискового включения*» для создания списка:

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

Правила работы со списковым включением представлены на рис. 7.

Рис. 7. Схема использования спискового включения

Функция map[37] позволяет создавать новый список на основе существующего:

```
In  [1]: def f(x):
             return x + 5

In  [2]: list(map(f, [1, 3, 4]))
Out [2]: [6, 8, 9]
```

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

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

Функциональные возможности Python позволяют определять небольшие однострочные lambda-функции там, где требуется вызов функции:

```
In  [1]: list(map(lambda s: s*2, "hello"))
Out [1]: ['hh', 'ee', 'll', 'll', 'oo']
```

Рассмотрим генерацию списка, состоящего из случайных целых чисел:

```
In  [1]: from random import randint
In  [2]: A = [randint(1, 9) for i in range(5)]
In  [3]: A
Out [3]: [5, 6, 3, 3, 6]
```

В данном примере функция range выступает как счетчик количества элементов создаваемого списка. В результате пять раз будет произведен вызов функции randint[38], которая сгенерирует целое псевдослучайное число из интервала от 1 до 9, и уже это число добавится в новый список.

Инструкция for используется, если заранее известно, сколько итераций необходимо выполнить (указывается через аргумент функции range или пока не закончится список/строка). В ином случае применяется инструкция цикла while. Тело цикла while выполняется до тех пор, пока выражение является истинным или не произошел выход из цикла с помощью инструкции break. Пример программы подсчета числа кроликов с использованием инструкции цикла while:

```
rabbits = 3
while rabbits > 0:
    print(rabbits)
    rabbits = rabbits - 1
```

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

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

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

```
3
2
1
```

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

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

Результат выполнения программы:

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

Тело цикла while в данном примере выполняется бесконечное число раз, т.к. True всегда является истиной. Выход из цикла осуществляется по инструкции break, если пользователь введет с клавиатуры строку "стоп".

Контрольные задания

1. Напишите программу-игру: компьютер загадывает случайное число, пользователь пытается его угадать. Пользователь вводит число до тех пор, пока не угадает или не введет слово «Выход». Компьютер сообщает пользователю больше или меньше введенное число относительно загаданного.
2. Напишите программу. На вход программы поступает произвольный текст. Найдите номер первого самого длинного слова в нем.
3. Напишите программу. На вход программы поступает произвольный текст. Напечатайте все имеющиеся в нем цифры, определите их количество и сумму.

2.9. Дополнительные встроенные типы данных в Python

Множество (set) – неупорядоченная коллекция неизменяемых, уникальных элементов:

```
In  [1]: v = {'A', 'C', 4, '5', 'B'}
In  [2]: v
Out [2]: {'C', 'B', '5', 4, 'A'}
```

Повторяющиеся элементы, которые добавились при создании, удаляются из результирующего множества:

```
In  [3]: v = {'A', 'C', 4, '5', 'B', 4}
In  [4]: v
Out [4]: {'C', 'B', '5', 4, 'A'}
```

Создание множества из списка:

```
In  [5]:  set([3, 6, 3, 5])
Out [5]:  {3, 5, 6}
```

Обратите внимание, что в момент создания множества из списка удалены повторяющиеся элементы. Это легкий способ очистить список от повторов:

```
In  [6]:  list(set([3, 6, 3, 5]))
Out [6]:  [3, 5, 6]
```

Рассмотрим некоторые операции над множествами[39]:

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

```
In   [1]:  s1 = set(range(5))
In   [2]:  s2 = set(range(2))
In   [3]:  s1
Out [3]:  {0, 1, 2, 3, 4}
In   [4]:  s2
Out [4]:  {0, 1}
In   [5]:  s1.add('5')  # добавление элемента
In   [6]:  s1
Out [6]:  {0, 1, 2, 3, 4, '5'}
```

Кортеж (tuple) схож по своим свойствам со списком. Он используется, когда необходимо быть уверенным, что элементы структуры данных не будут изменены в процессе работы программы. Рассмотрим некоторые популярные операции над кортежами[40]:

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

С помощью кортежей можно присваивать значения одновременно двум переменным:

```
In   [1]:  (x, y) = (10, 5)
In   [2]:  x
Out [2]:  10
In   [3]:  y
Out [3]:  5
In   [4]:  x, y = 1, 3     # убрали круглые скобки
In   [5]:  x
Out [5]:  1
In   [6]:  y
Out [6]:  3
```

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

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

Рассмотрим пример создания словаря, который каждому слову на английском языке (*множество ключей словаря*) ставит в соответствие слово на испанском языке (*множество значений словаря*). Схематично такой словарь представлен на рис. 8.

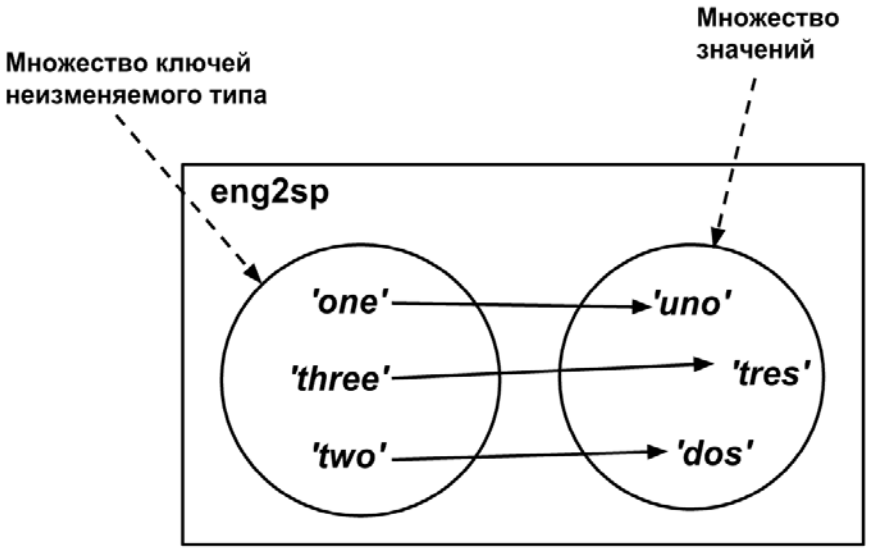


Рис. 8. Схема соответствия множеству ключей множества значений словаря

Создадим словарь-переводчик с английского языка на испанский язык:

```
In  [1]: eng2sp = dict()  # создаем пустой словарь
In  [2]: eng2sp
Out [2]: {}
In  [3]: eng2sp['one']='uno'
In  [4]: eng2sp
Out [4]: {'one': 'uno'}
In  [5]: eng2sp['one']
Out [5]: 'uno'
In  [6]: eng2sp['two'] = 'dos'
In  [7]: eng2sp['three'] = 'tres'
In  [8]: eng2sp
Out [8]: {'three': 'tres', 'one': 'uno', 'two': 'dos'}
```

2.10. Обработка исключений

В Python реализован[41] перехват ошибок, основанный на обработке исключительных ситуаций:

```
try:
   x = int(input("Введите число: "))
   print(5/x)
```

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

```
except:
    print("Возникла ошибка деления на нуль")
```

Выполним программу и попытаемся разделить на нуль:

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

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

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

except без указания типа исключительной ситуации перехватывает все виды возникающих ошибок. Как нам отделить ошибку деления на нуль от ошибки преобразования типов? Выполним несколько ошибочных команд, приводящих к исключениям[42]:

```
In [1]: 4/0

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-39-6de94738d89d> in <module>()
----> 1 4/0

ZeroDivisionError: division by zero

In [2]:  int("r")

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-40-991c836bf0cf> in <module>()
----> 1 int("r")

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

При делении на нуль возникает ошибка ZeroDivisionError, при преобразовании типов – ValueError:

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

```
try:
    x = int(input("Введите число: "))
    print(5/x)
except ZeroDivisionError:  # указываем тип исключения
    print("Возникла ошибка деления на нуль")
except ValueError:
    print("Возникла ошибка преобразования типов")
```

Теперь срабатывают различные блоки except в зависимости от типа возникающего исключения.

2.11. Работа с файлами

На практике данные для обработки часто поступают из внешних источников – файлов. Существуют различные форматы файлов, наиболее простой и универсальный – текстовый. Помимо текстовых есть другие типы файлов (музыкальные, видео, .doc, .ppt и пр.), которые открываются в специальных программах (музыкальных или видео проигрывателях, MS Word и пр.). Остановимся на текстовых файлах, хотя возможности Python ими не ограничиваются.

Выполните следующие шаги.

1. Создайте директорию file_examples в notebooks;
2. Создайте в директории file_examples текстовый файл example_text.txt, содержащий следующий текст:

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

3. В отдельной ячейке Jupyter выполните код:

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

В результате получится:

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

Описание программы представлено на рис. 9.

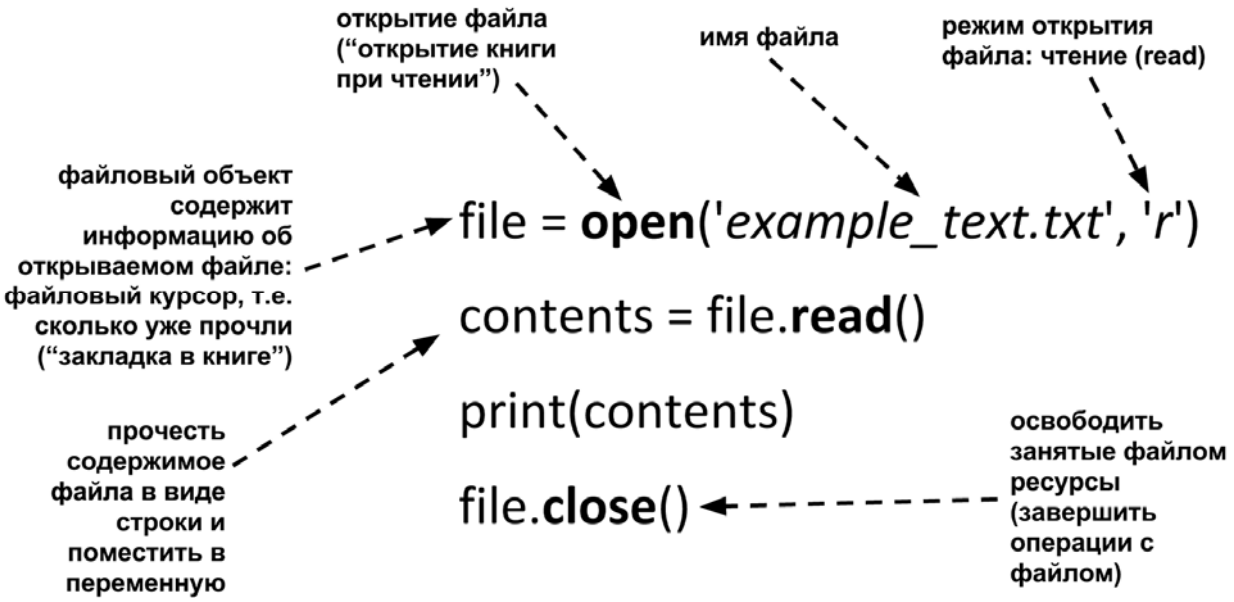


Рис. 9. Описание программы для работы с файлами в Python

Рассмотренный подход для работы с файлами[43] Python унаследовал из языка C. По умолчанию, если не указывать режим открытия, то используется открытие на «чтение» (можно открыть на «запись» или «добавление»). В дальнейшем для работы с файлами будем использовать *менеджер контекста* (инструкцию with), который не требует ручного освобождения ресурсов, т.е. вызова метода close. В следующем примере представлен исходный код программы с использованием менеджера контекста:

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

Выполним программу:

```
Ошибка открытия файла
```

Далее представлен пример чтения содержимого всего файла, начиная с текущей позиции курсора (перемещает курсор в конец файла):

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

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

Результат выполнения программы:

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

Jupyter обладает тесной интеграцией с оболочкой ОС. Существует набор магических функций[44], позволяющих выполнять команды, связанные с ОС:

```
In  [1]:  %pwd  # возвращает текущую рабочую директорию
Out [2]:  'C:\\WinPython-64bit-3.5.2.1Qt5\\notebooks'
```

Восклицательный знак ! в начале командной строки Jupyter означает, что все следующее за ним необходимо выполнить в оболочке ОС [2].

Контрольные задания

1. Напишите программу, которая создает (генерирует) полноценный HTML-документ, содержащий текст, приведенный по ссылке: http://dfedorov.spb.ru/python/files/tutchev.txt и под текстом размещает картинку: http://dfedorov.spb.ru/python/files/tutchev.jpg. Пример итогового HTML-документа: http://dfedorov.spb.ru/python/files/p.html (обратите внимание на код страницы, содержащей HTML-теги). Выполните обработку ошибок. В момент чтения и записи используйте параметр функции open – encoding='utf-8'.
2. Очистите файл от HTML-тегов: http://dfedorov.spb.ru/python/files/p.html. Выведите на экран «чистый» текст. В задании можно использовать только стандартные модули Python[45].

2.12. Создание собственных типов данных

Собственные типы данных в Python создаются через создание классов. Например, создадим тип данных (класс) Address:

```
class Address():    # имя класса выбирает программист
   name = ""        # строковое поле класса
   line = ""
   city = ""
```

[44] см. https://ipython.readthedocs.io/en/stable/interactive/magics.html
[45] см. регулярные выражения: https://docs.python.org/3/library/re.html

Определили шаблон для хранения места проживания человека. Превратить шаблон в конкретный адрес можно через создание *объекта* (*экземпляра*)[46] класса Address[47]:

```
home = Address()
```

Заполним через точку поля объекта:

```
# заполняем поле name объекта home:
home.name = "Вася Пупкин"
home.line1 = "'Невский пр. 22"
home.city = "СПб"
```

Возможности классов и объектов не ограничиваются только хранением состояния объекта. Классы также позволяют задавать функции внутри себя (методы) для работы с полями класса, т.е. влиять на поведение объекта. Для демонстрации создадим класс Dog:

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

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

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

# Вызываем метод bark объекта myDog,
# т.е. попросим собаку подать голос:
myDog.bark()

# Полная форма для вызова метода myDog.bark()
# будет: Dog.bark(myDog), где myDog – сам объект (self)
```

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

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

Результат работы программы:

```
Лайка  говорит гав
```

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

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

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

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

# Выводим имя собаки:
print(myDog.name)
```

Теперь имя собаки присваивается в момент ее создания («рождения»). В конструкторе указали self.name, т.к. в момент вызова конструктора вместо self подставится конкретный объект, т.е. myDog. Результат выполнения программы:

```
Лайка
```

В предыдущем примере для обращения к имени собаки выводили на экран поле myDog.name, т.е. залезали во «внутренности» Собаки. Для более «гуманной» работы с объектом добавим в класс Dog два метода setName и getName:

```
class Dog():
```

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

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

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

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

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

Выполним программу:

```
Лайка
Шарик
```

Контрольные задания

1. Создайте класс StringVar для работы со строковым типом данных, содержащий методы set и get. Метод set служит для изменения содержимого строки, get – для получения содержимого строки. Создайте объект типа StringVar и протестируйте его методы.
2. Создайте класс точка Point, позволяющий работать с координатами (x, y). Добавьте необходимые методы класса.

2.13. Иерархия наследования в Python

В Python все создаваемые классы наследуются от класса object. Создадим класс Point, в котором определим (переопределим методы базового класса object) специальные методы __init__, __eq__, __str__:

```
class Point:
    def __init__(self, x=0, y=0): # устанавливаем координаты точки
        self.x = x
        self.y = y
```

```
    def __eq__(self, other): # выполняет сравнение двух точек
        return self.x == other.x and self.y == other.y
    def __str__(self): # выполняет строковый вывод информации
        return "({0.x}, {0.y})".format(self)

a = Point()    # создаем объект, по умолчанию координаты: x=0, y=0
print(str(a))   # вызывается метод __str__ класса Point,
                # полная форма вызова метода: Point.__str__(a)
b = Point(3, 4)
print(str(b))
b.x = -19
print(str(b))
print(a == b, a != b)   # вызывается метод __eq__
# полная форма для сравнения a == b имеет вид: Point.__eq__(a, b)
```

Результат выполнения программы:

```
(0, 0)
(3, 4)
(-19, 4)
False True
```

В Python за каждой операцией над объектами закреплен специальный метод[48], например в момент сложения (+) вызывается метод __add__. Заметим, что в предыдущем примере не переопределялся метод __ne__ для неравенства a != b, но Python смог выполнить сравнение, т.к. принял его результат за обратный к равенству (вернул противоположный результат вызова метода __eq__).

Наследуем от класса Point класс Circle (рис. 10).

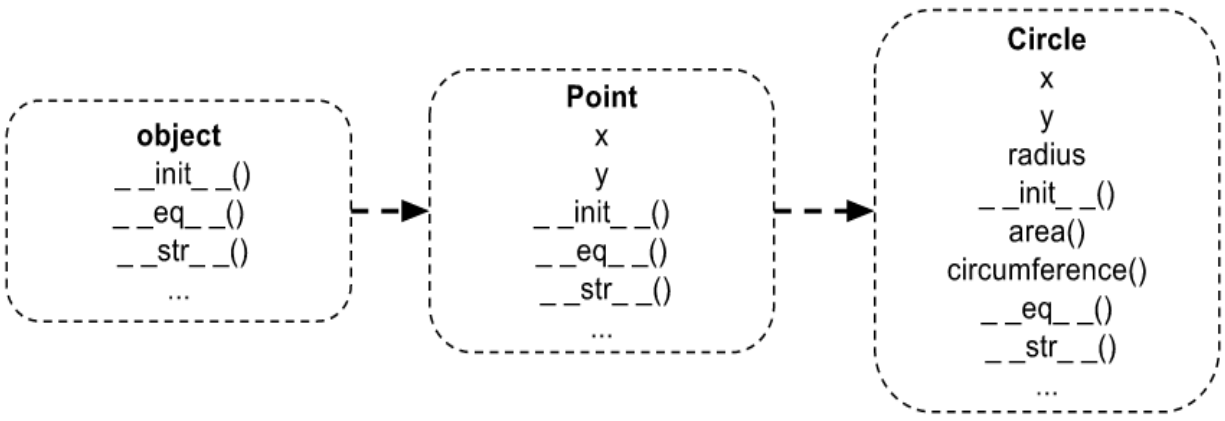


Рис. 10. Пример иерархии наследования классов

[48] см. https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types

Исходный код класса Circle:

```
class Circle(Point):
    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y) # вызов конструктора базового класса
        self.radius = radius
    def area(self): # вычисляет площадь окружности
        return math.pi * (self.radius ** 2)
    def circumference(self):  # вычисляет длину окружности
        return 2 * math.pi * self.radius
    def __eq__(self, other): # сравнивает две окружности
        return self.radius == other.radius and super().__eq__(other)
    def __str__(self):  # выводит информацию в виде строки
        return "({0.radius}, {0.x}, {0.y})".format(self)

circle = Circle(2) # создаем объект, radius=2, x=0, y=0
circle.radius = 3
circle.x = 12
a = Circle(4, 5, 6)
b = Circle(4, 5, 6)
print(str(a))    # вызывается специальный метод __str__
print(str(b))
print(a == b)  # вызывается специальный метод __eq__
# полная форма вызова метода для a == b: Circle. __eq__(a, b)
print(a == circle)
print(a != circle) # отрицание результата вызова метода __eq__
```

Результат выполнения программы:

```
(4, 5, 6)
(4, 5, 6)
True
False
True
```

2.14. Документирование и тестирование функций на языке Python

Вспомним, что для получения документации ранее вызывалась функция help. Данная функция считывает строку (многострочный комментарий в тройных кавычках), содержащуюся внутри тела функции:

```
def my_function():
    """Эта функция ничего не делает.
    """
    pass
help(my_function)
```

Результат выполнения программы:

```
Help on function my_function in module __main__:

my_function()
    Эта функция ничего не делает.
```

Помимо справочной информации, в комментариях к функции могут содержаться тесты для автоматизированной проверки правильности вычислений[49]:

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

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

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

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

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

В результате запуска программы на экране отобразится:

```
TestResults(failed=0, attempted=2)
```

Если в первом тесте заменить аргумент 70 на 71, то произойдет ошибка:

```
**********************************************************
File "__main__", line 4, in __main__.func_m
Failed example:
    func_m(20, 30, 71)
Expected:
    40.0
Got:
    40.333
**********************************************************
1 items had failures:
   1 of   2 in __main__.func_m
```

[49] см. https://docs.python.org/3.6/library/doctest.html

```
***Test Failed*** 1 failures.

TestResults(failed=1, attempted=2)
```

2.15. Сравнение времени работы алгоритмов поиска

Определим несколькими способами позиции двух наименьших элементов в неотсортированном списке, для этого на языке Python создадим три функции [3, 4].

1. Поиск индекса минимального элемента в списке, удаление найденного элемента, повторный поиск минимального элемента, возвращение удаленного элемента в список:

```
def find_two_smallest_1(L):
    smallest = min(L)
    min1 = L.index(smallest)
    L.remove(smallest)   # удаляем первый минимальный элемент

    next_smallest = min(L)
    min2 = L.index(next_smallest)
    L.insert(min1, smallest) # возвращаем первый минимальный обратно
    if min1 <= min2: # проверяем индекс второго минимального
        min2 += 1     # min2 = min2 + 1

    return (min1, min2)  # возвращаем кортеж
```

2. Сортировка списка, поиск минимальных элементов, определение их индексов:

```
def find_two_smallest_2(L):
    temp_list = sorted(L)  # возвращаем копию отсортированного списка
    smallest = temp_list[0]
    next_smallest = temp_list[1]
    min1 = L.index(smallest)
    min2 = L.index(next_smallest)
    return (min1, min2)
```

3. Сравнение каждого элемента по порядку:

```
def find_two_smallest_3(L):
    if L[0] < L[1]:
        min1, min2 = 0, 1  # устанавливаем начальные значения
    else:
        min1, min2 = 1, 0
    for i in range(2, len(L)):
        if L[i] < L[min1]:
            min2 = min1
```

```
            min1 = i
        elif L[i] < L[min2]:
            min2 = i
    return (min1, min2)
```

Время работы алгоритмов измерим с помощью встроенной в Jupyter команды:

```
In [1]: %time?

Docstring:
Time execution of a Python statement or expression.
…
```

Создадим три списка A1, A2 и A3, состоящих из случайных элементов:

```
In [2]: from random import randint
In [3]: A1 = [randint(1, 1000) for i in range(50000)]
In [4]: A2 = [randint(1, 1000) for i in range(100000)]
In [5]: A3 = [randint(1, 1000) for i in range(300000)]
```

Пример вычисления времени работы алгоритма find_two_smallest_1 для списка A1:

```
In   [6]: %time find_two_smallest_1(A1)
          Wall time: 4.01 ms
Out [6]: (250, 2158)
```

Результаты расчета времени представлены в табл. 1.

Таблица 1

Расчет времени работы алгоритмов поиска

	A1	**A2**	**A3**
find_two_smallest_1	4.01 ms	7.01 ms	22 ms
find_two_smallest_2	21 ms	41 ms	154 ms
find_two_smallest_3	15 ms	28 ms	89.1 ms

Контрольные задания

1. Напишите функцию, которая возвращает разность между наибольшим и наименьшим значениями из списка целых случайных чисел. Определите время работы алгоритма для различных входных данных.
2. Напишите программу, которая для целочисленного списка из 1000 случайных элементов определяет, сколько отрицательных элементов

располагается между его максимальным и минимальным элементами. Определите время работы алгоритма для различных входных данных.

2.16. Построение графиков с помощью matplotlib

Matplotlib[50] является пакетом для визуализации двумерной графики[51] на Python. Для демонстрации примера работы воспользуемся модулем pyplot[52], обеспечивающим MATLAB-подобный интерфейс:

```
# Подключение возможностей пакета NumPy
# для работы с однородными многомерными массивами:
import numpy as np
# Команда для вывода интерактивного графика в окне браузера:
%matplotlib notebook
# Подключение возможностей пакета matplotlib:
import matplotlib.pyplot as plt
# Генерация последовательности чисел от -10 до 10 с шагом 100,
# возвращает NumPy массив (значения по оси X):
x = np.linspace(-10, 10, 100)
# Создаем второй массив (значения по оси Y):
y = np.sin(x)
#  Создание линейного графика на основе двух массивов:
plt.plot(x, y)
```

Результат выполнения программы представлен на рис. 11.

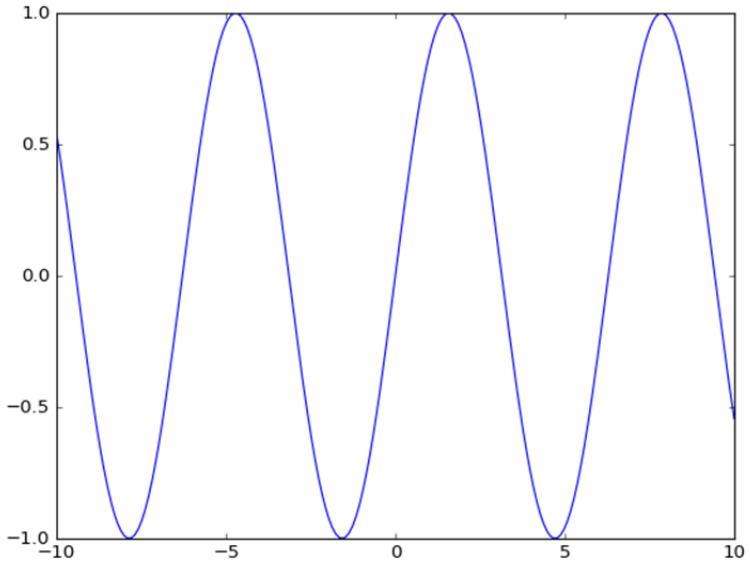


Рис. 11. График функции sin(x)

[50] см. http://matplotlib.org
[51] см. http://matplotlib.org/gallery.html
[52] см. http://matplotlib.org/users/pyplot_tutorial.html

Построим графики зависимости времени работы алгоритмов поиска (см. п. 2.15) от числа входных элементов[53]:

```
# Команда для вывода графика в окне браузера:
%matplotlib notebook
# Подключение возможностей пакета matplotlib:
import matplotlib.pyplot as plt
plt.plot([4.01, 7.01, 22], color="blue",
        linewidth=2.5, linestyle="-",
        label="find_two_smallest_1")
plt.plot([21, 41, 154], color="red",
        linewidth=2.5, linestyle="-",
        label="find_two_smallest_2")
plt.plot([15, 28, 89.1], color="green",
        linewidth=2.5, linestyle="-",
        label="find_two_smallest_3")
plt.legend(loc='upper left', frameon=False)
```

Результат выполнения программы представлен на рис. 12.

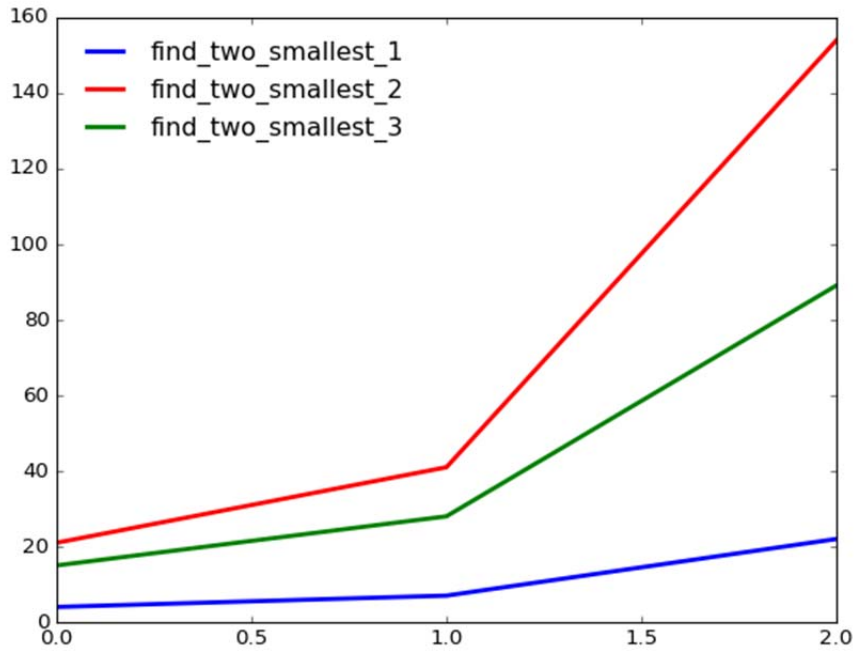


Рис. 12. Графики зависимости времени работы алгоритмов поиска от числа входных элементов

2.17. Создание графического интерфейса с помощью tkinter

Язык Python позволяет создавать приложения с графическим интерфейсом, для этого применяются различные графические библиотеки[54].

[53] см. http://matplotlib.org/api/pyplot_api.html
[54] см. https://wiki.python.org/moin/GuiProgramming

Остановимся на рассмотрении стандартной графической библиотеки tkinter[55].

Первым делом при работе с tkinter необходимо создать главное (корневое) окно (рис. 13), в котором размещаются остальные графические элементы – *виджеты*. Существует большой набор виджетов[56] на все случаи жизни: для ввода текста, вывода текста, выпадающие меню и т.д. Среди виджетов есть кнопка, при нажатии на которую происходит заданное событие. Некоторые виджеты (фреймы) используются для группировки других виджетов внутри себя.

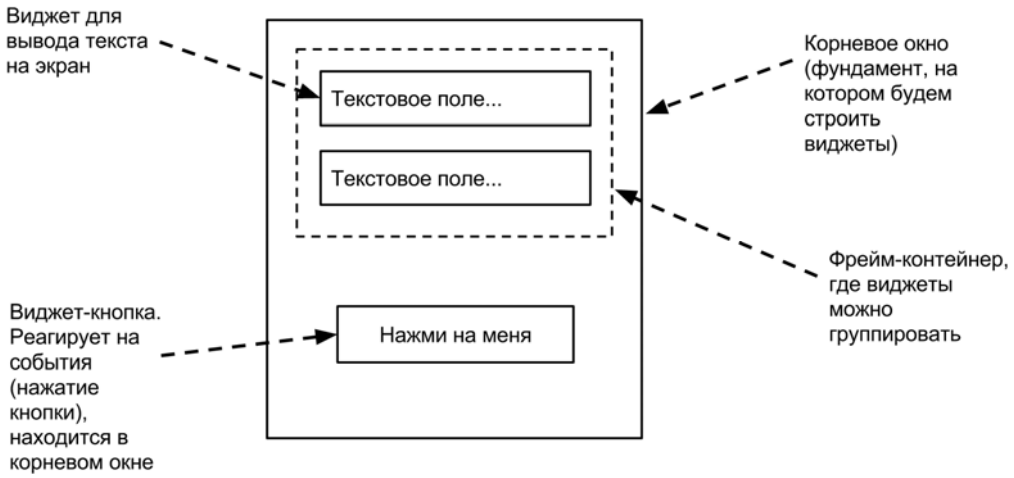
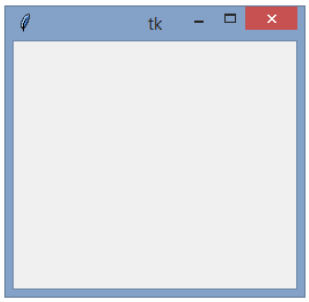


Рис. 13. Схема главного окна в tkinter

Приведем пример простейшей программы для отображения главного окна:

```
# Подключаем модуль, содержащий методы для работы с графикой
import tkinter
# Создаем главное (корневое) окно,
# в переменную window записываем ссылку на объект класса Tk
window = tkinter.Tk()
# Задаем обработчик событий для корневого окна
window.mainloop()
```

Результат выполнения программы:

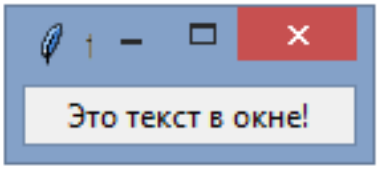


[55] см. https://docs.python.org/3/library/tkinter.html
[56] см. http://effbot.org/tkinterbook/tkinter-index.htm

Следующий пример демонстрирует создание виджета Label:

```
import tkinter
window = tkinter.Tk()
# Создаем объект-виджет класса Label в корневом окне window
# text – параметр для задания отображаемого текста
label = tkinter.Label(window, text = "Это текст в окне!")
# Отображаем виджет с помощью менеджера pack
label.pack()
window.mainloop()
```

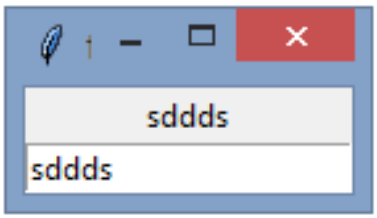
В результате работы программы отображается графическое окно с текстом внутри:



Следующий пример показывает, каким образом используется виджет Entry для ввода данных:

```
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
var = tkinter.StringVar()
# Обновление содержимого переменной в момент ввода текста
label = tkinter.Label(frame, textvariable=var)
label.pack()
# Пробуем набрать текст в появившемся поле для ввода
entry = tkinter.Entry(frame, textvariable=var)
entry.pack()
window.mainloop()
```

Запустим программу и попробуем набрать произвольный текст:



Видим, что текст, который мы набираем, сразу отображается в окне. Дело в том, что виджеты Label и Entry используют для вывода и ввода текста

соответственно одну и ту же переменную data класса StringVar[57]. Подобная схема работы оконного приложения укладывается в универсальный шаблон (паттерн), который называется «Модель-вид-контроллер» (Model-View-Controller или MVC[58]). В общем виде под *моделью* (Model) понимают способ хранения данных, т.е. как данные хранятся (например, в переменной какого класса). *Вид* (View) служит для отображения данных. *Контроллер* (Controller) отвечает за обработку данных (рис. 14).

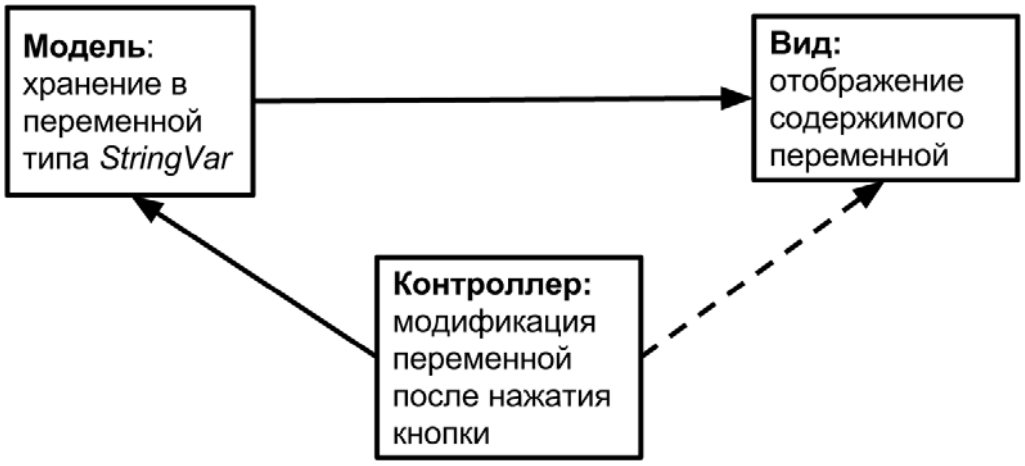


Рис. 14. Пример шаблона MVC

Интересная особенность MVC в том, что в случае изменения контроллером данных (как это было в предыдущем примере с изменением переменной var), «посылается сигнал» *виду* с просьбой обновить отображаемое содержимое (перерисовать окно), отсюда получается обновление текста в режиме реального времени.

В следующем примере введенный текст (виджет Entry) отображается на экране (виджет Label) только в момент нажатия кнопки (виджет Button):

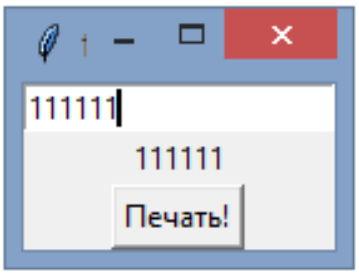
```
import tkinter
# Вызывается в момент нажатия на кнопку:
def click():
   # Получаем строковое содержимое поля ввода и
   # с помощью config изменяем отображаемый текст
   label.config(text=entry.get())

window = tkinter.Tk()
frame = tkinter.Frame(window)
```

[57] Tkinter поддерживает работу с переменными классов: BooleanVar, DoubleVar, IntVar, StringVar.

[58] Паттерн MVC получил широкое распространение при разработке веб-приложений.

```
frame.pack()
entry = tkinter.Entry(frame)
entry.pack()
label = tkinter.Label(frame)
label.pack()
# Привязываем обработчик нажатия на кнопку к функции click
button = tkinter.Button(frame, text='Печать!', command=click)
button.pack()
window.mainloop()
```

Результат выполнения программы:



Контрольные задания

1. Напишите программу с графическим интерфейсом, переводящую градусы по шкале Фаренгейта в градусы по шкале Цельсия.
2. Напишите программу с графическим интерфейсом, которая позволяет произвольный текст, введенный с клавиатуры, сохранить в обычный текстовый файл либо в файл HTML-формата.
3. Напишите программу-генератор паролей с графическим интерфейсом. В качестве входных параметров генератора можно указать:
   - наличие цифр;
   - наличие прописных букв;
   - наличие строчных букв;
   - наличие специальных символов %, *,),?, @, #, $, ~;
   - длину пароля.

РАЗДЕЛ 3. ИНТЕГРАЦИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

«*И хотя C используется все реже и реже, он остается лингва-франка для работающих программистов. Это тот язык, который используется, чтобы общаться друг с другом и, что еще более важно, он гораздо ближе к машине, чем «современные» языки, которым вас учат в колледже: ML, Java, Python, какому бы новомодному мусору не учили сегодня. Вам нужно, как минимум, семестр чтобы стать ближе к машине, иначе вы никогда не сможете создавать эффективный код на языках более высокого уровня. Вы никогда не сможете работать над компиляторами и операционными системами, а это одни из самых лучших рабочих мест. Вам никогда не доверят создавать архитектуру больших проектов. Меня не интересует, сколько вы знаете о последовательностях, замыканиях и обработке исключений, если вы не можете объяснить, почему* ***while (*s++ = *t++);*** *копирует строку, или это для вас не одна из самых естественных вещей в мире, ну, тогда вы программируете, основываясь на суевериях, подобно доктору, который, не зная анатомии, отпускает рецепт, основываясь на том, что говорит аптекарша*».

(Джоэл Спольски, программист, писатель [5])

3.1. Введение в язык C

Язык программирования C был создан Деннисом Ритчи в 1971 г. как язык системного программирования для PDP-11-варианта UNIX системы и основывался на языке B, разработанном Кеном Томпсоном (с недавнего времени в Google занимается разработкой языка Go).

В 1973 году вышла четвертая версия UNIX, полностью переписанная на C.

В 1978 г. опубликована книга Брайана Кернигана и Денниса Ритчи «Язык программирования Си», ставшая на тот момент стандартом языка (K&R) [6].

1988 г. – проект стандарта ANSI (C89 или C90).

1999 г. – стандарт ISO/IEC 9899.

8 декабря 2011 опубликован новый стандарт для языка Си (ISO/IEC 9899:2011)

Напишем первую программу на языке C [7, 8]:

```
/* hello.c */
#include <stdio.h>
int main() {
        printf("Hello, world\n");
        return 0;
}
```

Для компиляции воспользуемся компилятором GCC (GNU Compiler Collection)[59]:

```
gcc -Wall -g -o hello hello.c
```

hello.c – исходный файл (с исходным текстом программы), hello – исполняемый объектный файл, -o опция (флаг), задающая имя исполняемого объектного файла (по умолчанию a.out). Флаг -Wall отображает все возможные предупреждения, -g – добавляет отладочную информацию. Запуск программы в ОС GNU/Linux производится командой: ./hello

Компиляция программы включает несколько стадий (рис. 14) [9].

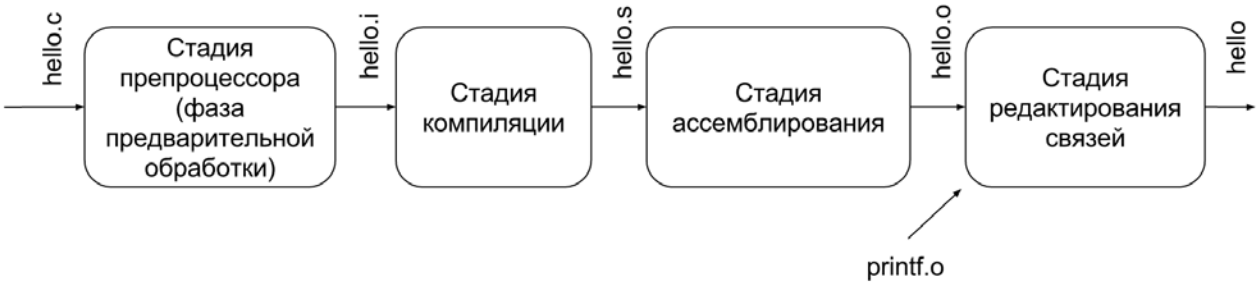


Рис. 14. Система компиляции GCC

- *Стадия препроцессора* (фаза предварительной обработки). Препроцессор cpp считывает содержимое системного файла заголовков stdio.h и добавляет его в текст программы. На выходе стадии препроцессора файл hello.i:

```
gcc -E hello.c -o hello.i
```

- *Стадия компиляции*. Компилятор cc транслирует hello.i в hello.s, содержащий программу на языке ассемблера. Каждый оператор на языке ассемблера точно описывает одну из машинных инструкций в текстовой форме:

```
gcc -S hello.c
```

- *Стадия ассемблирования*. Ассемблер as транслирует файл в инструкции машинного языка, перемещаемую объектную программу (hello.o).

[59] В ОС Windows потребуется установить «Minimalist GNU for Windows»: http://www.mingw.org. В качестве альтернативы GCC рекомендуется Clang: http://clang.llvm.org

- *Стадия редактирования связей*. Программа обращается к функции printf. Редактор связей ld объединяет объектные файлы в исполняемую программу hello.

3.2. Создание модуля Python на языке C

Рассмотрим процесс написания расширения (модуля) Python на языке программирования C[60].

На самом деле, Python – это спецификация того, как должен выглядеть язык программирования. Реализовывать эту спецификацию можно на любом другом языке программирования: C, Java, C#, Ruby и т.д. В пособии используется CPython – реализация на языке C, которая поддерживается официальными разработчиками языка. CPython предоставляет API (Application Programming Interface) для разработчиков[61], желающих написать собственный модуль на языке C.

Рассмотрим пример реализации модуля (plus.c), который содержит только одну функцию сложения двух чисел, переданных ей в качестве аргумента:

```
#include <Python.h>

static PyObject *py_plus(PyObject *self, PyObject *args) {
  double x, y, result;

  if (!PyArg_ParseTuple(args,"dd", &x, &y)) {
    return NULL;
  }
  result = x + y;
  return Py_BuildValue("d", result);
}

static PyMethodDef ownmod_methods[] = {
{
        "plus",         // название функции внутри Python
         py_plus,      // функция C
         METH_VARARGS, /* макрос, поясняющий, что функция у нас с аргументами */
         "plus function" // документация для функции внутри Python
    },
    { NULL, NULL, 0, NULL } /* так называемый sentiel. Сколько бы элементов структуры у вас не было, этот нулевой элемент должен быть всегда, и при этом быть последним */
};
```

---

[60] Данный раздел написан при активном участии студентов СПбГЭУ направления «Прикладная математика и информатика» Ю.С. Макарова и Д.Ю. Митюры.

[61] см. https://docs.python.org/3.6/extending/index.html#extending-index
см. https://docs.python.org/3.6/c-api/

```
static PyModuleDef simple_module = {
    /* Описываем наш модуль */
    PyModuleDef_HEAD_INIT, // обязательный макрос
    "my_plus", // my_plus.__name__
    "amazing documentation", // my_plus.__doc__
    -1,
    ownmod_methods // сюда передаем методы модуля
};

// В названии функции обязательно должен быть префикс PyInit
PyMODINIT_FUNC PyInit_my_plus(void) {
   PyObject* m;
   // создаем модуль
   m = PyModule_Create(&simple_module);
   // если все корректно, то эта проверка не проходит
   if (m == NULL)
      return NULL;
   return m;
}
```

В первой строке импортируется заголовочный файл Python.h. Он содержит API CPython. Его необходимо подключать перед другими модулями, т.к. он содержит в себе директивы препроцессора, которые могут оказать влияние на стандартные заголовочные файлы. Далее объявляется функция, складывающая два числа. Стоит обратить внимание на тип возвращаемого значения, а также аргументы функции. PyObject[62] – это C-структура, являющаяся базовой для всех остальных объектов. Все функции в будущем модуле, должны иметь данный тип возвращаемого значения.

self и args – обязательные аргументы функции. self – указатель на объект самого модуля, если данная функция принадлежит модулю, или указатель на объект, если это функция какого-либо класса. args – это кортеж аргументов функции, переданный прямиком Python. Например, если вызвать simple.plus(1, 3), то в этом случае args будет состоять из чисел (1, 3).

В теле функции объявляются три переменные, которые будут в дальнейшем использованы в функции PyArg_ParseTuple. Остановимся на ней подробнее. Она используется для обработки аргументов функции и позволяет трансформировать типы данных Python в типы данных C. В качестве первого аргумента она принимает PyObject, который, по сути, является кортежем переданных функции аргументов, вторым – типы переменных, в которые мы хотим преобразовать наши аргументы. Далее идут

[62] см. https://docs.python.org/3.6/c-api/structures.html#c.PyObject

переменные, в которые сохраняем преобразованные значения. Несколько примеров использования этой функции:

```
PyArg_ParseTuple(args, "s", &some_string) /* проверить, содержит ли массив args один элемент строкового типа, если да – сохранить это значение в переменную some_string */
PyArg_ParseTuple(args, "i", &some_int) // аналогично для числа
```

Далее вычисляем значение переменной result, и используем функцию Py_BuildValue[63], которая позволяет конвертировать переменные C в переменные соответствующих Python-типов. Примеры работы данной функции представлены в табл. 2.

Таблица 2

Соответствие между вызовами функции Py_BuildValue и возвращаемыми типами данных Python

Пример вызова функции	**Результат конвертирования**
Py_BuildValue("")	None
Py_BuildValue("i", 123)	123
Py_BuildValue("iii", 123, 456, 789)	(123, 456, 789)
Py_BuildValue("s", "hello")	'hello'
Py_BuildValue("y", "hello")	b'hello'
Py_BuildValue("ss", "hello", "world")	('hello', 'world')

Для того чтобы преобразовать C-программу в Python-модуль потребуется создать файл setup.py в той же директории, где лежит C-программа. Файл должен иметь следующее содержание:

```
from distutils.core import setup, Extension

module1 = Extension(
    'my_plus',              # название модуля внутри Python
    sources = ['plus.c'] # исходные файлы модуля
)

setup(
    name = 'my_plus',       # название модуля (my_plus.__name__)
    version = '1.1',            # версия модуля
    description = 'Simple module',     # описание модуля
    ext_modules= [module1]     # объекты типа Extension
)
```

Далее, необходимо последовательно в GNU/Linux выполнить несколько команд:

---

[63] см. https://docs.python.org/3.6/c-api/arg.html#c.Py_BuildValue

```
sudo python3 setup.py build
sudo python3 setup.py install
```

После этого можно запустить python3, импортировать написанный модуль, и вызвать метод plus сложения двух чисел:

```
>>> import my_plus
>>> my_plus.plus(1, 100)

101.0
```

ЗАКЛЮЧЕНИЕ

Область разработки программного обеспечения стремительно меняется. На взгляд авторов, данная дисциплина носит, в больше степени, практический характер, поэтому в учебном пособии отражены основные, но далеко не все темы, входящие в дисциплину «Технологии и методы программирования». За рамками пособия остались: компонентное программирование, аспектно-ориентированное программирование, автоматное программирование и др.

Исходя из практической направленности, рекомендуем в учебном процессе дополнительно задействовать бесплатные online-платформы[64], интерактивные задания[65], включающие игровые механизмы. Для развития командной деятельности можно создавать учебные проекты с последующей их защитой в конце семестра.

[64] см. https://stepik.org
[65] см. https://checkio.org

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Хендерсон П.* Функциональное программирование. Применение и реализация : Пер. с англ. – М.: Мир, 1983.

2. *Маккинни Уэс*. Python и анализ данных. – М.: ДМК Пресс, 2015.

3. *Томас Х. Кормен*. Алгоритмы. Вводный курс : Пер. с англ. – М. : Вильямс. 2014. – 208 с.

4. *Роберт Седжвик, Кевин Уэйн, Роберт Дондеро*. Программирование на языке Python. Учебный курс. курс : Пер. с англ. – М. : Вильямс. 2017. – 736 с.

5. *Джоель Спольски*. Совет студентам, изучающим вычислительную технику [электронный ресурс]. URL: http://russian.joelonsoftware.com/Articles/AdviceforComputerScienceC.html (дата обращения: 24.02.2017).

6. *Керниган Б., Ритчи Д.* Язык программирования Си: Пер. с англ. – 3-е изд. – СПб.: Невский Диалект, 2001.

7. *Столяров А.В*. Программирование: введение в профессию. 1: Азы программирования. – М.: МАКС Пресс, 2016.

8. *Столяров А.В*. Программирование: введение в профессию. II: Низкоуровневое программирование. – М.: МАКС Пресс, 2016.

9. *Рэндал Э. Брайант, Дэвид Р.* О`Халларон. Компьютерные системы: архитектура и программирование. – СПб.: БХВ-Петербург, 2005.

Учебное издание

Гниденко Ирина Геннадьевна
Федоров Дмитрий Юрьевич

ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Издано в авторской редакции

Подписано в печать 11.04.17. Формат 60×84 1/16.
Усл. печ. л. 3,75. Тираж 50 экз. Заказ 284.

Издательство СПбГЭУ. 191023, Санкт-Петербург, Садовая ул., д. 21.

Отпечатано на полиграфической базе СПбГЭУ