Питон
Курс лекций

© Зайцев Вадим Валерьевич, 2002–2010,
spider.vz@gmail.com

I
Лекция первая

Какое бы название ни имел тот или иной курс, первая лекция обычно не содержит ничего (или почти ничего) из той основной и самой важной части курса, ради которой он и был задуман. Не отступая от этой традиции, мы сегодня расскажем непосвящённым о том, что такое компьютер и зачем он нам нужен и введём несколько основных определений, которые позже неминуемо перейдут в класс очевидных и интуитивно понятных.

На самом деле первая лекция курса нужна лишь для того, чтобы дать почувствовать, что предмет из себя представляет, для чего он нужен и что он может дать лично вам.

Весь материал будет делиться на несколько больших тем, каждая из которых вправе содержать какое-то ненулевое количество малых тем.

1 Введение

1.1 ЭВМ и её обеспечение

Обеспечение компьютера, как известно, делится на две неравные части: аппаратное и программное. Это деление сродни делению человека на душу и тело. Аппаратное обеспечение — это тело, то есть всё, существующее в качестве деталей: корпус, монитор, платы, устройства ввода/вывода информации, различные провода, шлейфы и порты. Программное обеспечение — душа компьютера — куда богаче и разнообразнее: от содержимого микросхем `BIOS` и загрузочных секторов дисков до новой версии Windows, которая может занимать многие гигабайты. Программное обеспечение часто делят на системное и прикладное.

Системное программное обеспечение — это то, которым вы пользуетесь, сами того не замечая, либо к которому прибегаете в самых тяжёлых моментах жизни компьютера. То есть: операционные системы, драйверы и всевозможные утилиты.

Прикладное программное обеспечение является, вообще говоря, предметом роскоши, посему чрезвычайно разнообразно: *файловые менеджеры*, *редакторы* и *просмотрщики* многочисленных форматов файлов, *проигрыватели* музыки и видео, *архиваторы* (числившиеся не так давно в системном), *вычислительные системы*, *сетевые приложения* и *средства подготовки программ*, которые заслуживают того, чтобы остановиться на них подробнее. В эту категорию относят все программы, служащие для производства новых программ.

Спектр средств подготовки программ содержит *редакторы* исходных текстов (обычно обеспечивающих подсветку (выделение некоторых элементов текста, имеющих значение для пользователя: скобок, служебных слов и т.д.) и некоторую поверхностную проверку синтаксиса вводимых конструкций), *трансляторы* (позволяющие собственно запускать программы), *отладчики* (призванные служить благородному делу поиска ошибок в программах, но не всегда помогающие программисту) и в некоторых случаях ещё и *тесты* (профайлеры), позволяющие, например, определить наиболее медленный или наиболее требовательный к ресурсам блок программы.

В последнее время чётко выделились две тенденции, употребляющиеся соответственно в двух выдержавших жестокую конкуренцию семействах операционных систем: **UNIX** и **Windows** (операционные системы некогда переживали бум, теория их строения была сформулирована очень подробно, но, увы, многие разработки так и остались в теоретической области). В юниксах, вообще говоря, всего два редактора: *vi* и *emacs*, и каждый юниксоид, подчас великолепно владея одним из них, с трудом догадывается о том, как выйти из другого. *Emacs*, например, определяет по расширению открываемого файла, какую подсветку ему применять, и в стандартной поставке содержит до сотни подсветок разных языков программирования. Отладчик в юниксе также один, работает на очень низком уровне и редко помогает на практике при использовании языка сколь-нибудь высокого уровня. Таким образом, в поставку языка под юниксовую платформу обычно включается только транслятор (и лишь в редких случаях высокоуровневый отладчик).

Под **Windows** дело обстоит несколько иначе — все вышеперечисленные компоненты спаяны воедино и результат называется *интегрированной средой разработки*. Выглядит это, как вы, вероятно, знаете, как редактор, из которого различными комбинациями клавиш можно вызвать такие действия, как компиляцию исходного текста, выполнение программы, запуск отладчика, и т.д.

1.2 Трансляторы языков программирования

Трансляторы бывают трёх типов: *ассемблеры*, *компиляторы* и *интерпретаторы*. Ассемблеры переводят программу на языке ассемблера в машин-

ные коды. При этом каждой строчке исходного текста ставится в соответствие одна команда процессора (от одного до дюжины байт кода). Компиляторы переводят текст программы на языке высокого уровня в машинные коды. При этом одной строчке исходного текста (которая в языке высокого уровня может иметь невероятно сложную структуру) может соответствовать много тысяч команд процессора. Интерпретаторы исполняют программу на языке высокого уровня немедленно, строчка за строчкой. Естественно, для этого они должны перевести исходный текст в другое представление, которое не обязано быть машинным кодом. Например, транслятор языка Ява переводит исходный текст в команды так называемой виртуальной машины.

Вообще, справедливо следующее:

II
Лекция вторая

Раз уж зашла речь о языках, это достойно того, чтобы поговорить подробнее:

1.3 Типы языков программирования и их эволюция

По этой теме написано книг чуть ли не больше, чем по каждому языку отдельно. Но мы попытаемся ограничиться лишь общим обзором.

1. Ассемблеры — это вербализованые машинные коды. Сколько машинных архитектур, столько и ассемблеров. Даже самая малая программа занимает много страниц на этом языке, абстракции никакой, уровень сверхнизкий. Сейчас эти языки используются только в мелких, но очень важных частях систем, которым необходимо быстродействие.

2. Процедурные языки — языки среднего и высокого уровня, ориентированые на деление основной проблемы на несколько более мелких и решение каждой мелкой с помощью своей подпрограммы. Основные представители этого направления: `Фортран` (в настоящее время используется версия Фортран-99, и та только в программировании больших численных проектов, откуда постепенно вытесняется готовыми математическими вычислительными системами вроде Мэпл, Матлаб и других), `Кобол` (применяется в области экономики), `Алгол` (не

применяется нигде, но в 60х годах имел большое теоретическое влияние на развитию теории языков программирования), Си (уже почти не используется), Ада (широко использовался Департаментом Защиты США, сейчас заменён) и Паскаль (пока что используется в системе Дельфи, но постепенно умирает).

Большинство используемых процедурных языков имеют ограниченные возможности работы с объектами, но не дотягивают до языков следующей категории.

3. Объектно-ориентированные языки — языки высокого уровня, ориентированные только на работу с различными объектами. Наиболее используемая в наше время группа. Основные представители: Си++ (очень широко используется во многих областях), Ада-95 (опять-таки, используется в основном Департаментом Защиты США), Ява (потомок Си++, используется всё шире с каждым днём, удобен для интернет-программирования), Смоллток (один из первых объектно-ориентированных языков программирования, живой и по сей день), КЛОС (о котором ниже) и Эйфель (программирование, ориентированное на ограничения — инварианты).

4. Языки, ориентированные на данные — языки, созданные специально для работы с одним определённым типом данных. Например, АПЛ настроен на работу с матрицами и векторами без циклов, Снобол и его преемник Икон работают со строками как с базовой структурой, СЕТЛ позволяет описывать множества почти математическим языком, Форт полностью ориентирован на стек.

5. Функциональные языки — практически разросшийся подтип языков, ориентированных на данные. Основная структура данных — связный список. Функциональными языками они названы засчет того, что программирование на них принципиально отличается от процедурного. Функциональные языки — это ЛИСП и его потомки: более объектно-ориентированный — КЛОС и более чисто реализующий функциональную парадигму — МЛ.

6. Логические языки — языки, ориентированные на решение проблем без описания алгоритмов. Действительно используется только один язык — Пролог. Где? Конечно, в области искусственного интеллекта.

7. Сценарные языки, ещё называемые скриптами — это языки, для которых не существует отдельной от какого-либо программного продукта реализации, либо используемые только в связке с одной программой или типом программ. Это, конечно, прежде всего Яваскрипт, простейшее и одновременно наиболее широко используемое средство интернет-программирования. Этот язык позволяет управлять браузером — программой просмотра интернет-документов — причём его

возможностей хватает подчас для реализации больших серьёзных проектов. Ещё два типично сценарных языка (выросших, однако, и приобретших отдельные реализации) — это `Перл` и `Питон` — две большие противоположности. Перл — очень сложный и мощный сиподобный язык, питон — попроще и полегче, к тому же более архаично построенный, паскале- или даже фортраноподобный. Хотя простота, конечно, не всегда означает меньшие возможности.

На этом можно наш обзор завершить. Конечно, языков программирования существуют многие тысячи, к тому же есть ещё широко используемые языки разметки, такие как `HTML`, `XML` или TEX.

Мы недаром перескочили через определение языка программирования. «Некорректный вопрос», как написано в одной уважаемой книге. Вообще, формальное определение существует.

Определение. Программы суть последовательности символов, определяющие вычисление.

Определение. Языки программирования суть наборы правил, определяющих, какие последовательности символов составляют программу и какое именно вычисление описывается этой программой.

Как видно, можно дать определение, даже не используя слово *компьютер*. На деле же язык программирования используется как механизм абстрагирования, позволяя программисту описать вычисления абстрактно и перекладывая большую часть работы на транслятор.

2 Введение в язык питон

2.1 Краткая история языка

Питон — молодой сценарный язык, история которого началась только в 1990 году, когда сотрудник голландского института CWI, тогда ещё мало кому известный Гвидо ван Россум участвовал в проекте создания языка ABC. Этот язык был предназначен для замены языка Бейсик в обучении студентов основным концепциям программирования. (Язык Бейсик как-то странно и надолго закрепился в сфере обучения, хотя многие понимали, что к добру это привести не может. Например, одно из светил теории программирования Эдсгер Дейкстра говорил, что «преподавателей, которые начинают обучение программированию с бейсика, следует привлекать к уголовной ответственности»).

Параллельно с работой над основным проектом Гвидо ван Россум дома на своём Макинтоше написал интерпретатор другого простого языка; он, конечно, позаимствовал некоторое количество идей из ABC. Он назвал его «Питон» и стал распространять через Интернет.

Язык стал быстро развиваться, поскольку появилось большое количество заинтересованных и понимающих в развитии языков программирования людей. Сначала это был совсем простой язык, просто небольшой интерпретатор, некоторое количество функций, не было объектно-

ориентированного программирования, но всё это быстро появилось. Уже в 1991 году появились первые средства объектно-ориентированного программирования.

Позже Гвидо ван Россум переехал из Голландии в Америку, перешёл из CWI в CNRI, потом в фирму BeOpen Labs, а сейчас работает в Digital Creations. Всё это время он продолжает развитие языка, выпуская новые версии. Причём каждая следующая версия имеет несколько серьёзных отличий от предыдущей, меняющих подчас саму философию программирования и подходы к решению различных задач.

Интерпретаторы питона существуют под все мыслимые платформы: **Windows**, **UNIX**, **Macintosh**, **QNX** и пр. Все они распространяются бесплатно, что обеспечивает дополнительную привлекательность использования этого языка как в коммерческих, так и в свободно-распространяемых проектах.

Последняя версия питона — 2.1 — уже пятнадцатая, откомпилированная под **Windows** 16 апреля 2001 в `18:25:49`. Спектр разработанного программного обеспечения (как в форме отдельных программ, так и в форме подключаемых модулей) очень разнообразен:

- `Zope` — сервер интернет-приложений, позволяющий создавать и поддерживать интернет-сайты со сложной структурой не только профессионалам, но и простым редакторам и наборщикам.

- `Jython` — реализация питона, позволяющая компилировать программы на нём в коды виртуальной ява-машины (универсального воображаемого компьютера, в команды которого компилируются программы на языке ява). Установка явы на персональный компьютер означает установку программы, позволяющей выполнять команды виртуальной ява-машины, поэтому откомпилированный программы на яве остаются машинно-независимыми (если говорить о реальных машинах, конечно). Сейчас возможность запускать мелкие программы на яве (так называемые *апплеты*) встроена почти в каждый браузер, и Jython — это начало наступления питона на яву.

- `Blender` — пакет работы с трёхмерной графикой и создания сложных фильмов, использующий питон по прямому назначению — в качестве сценарного языка. Питон позволяет легко в паре десятков строк кода сформулировать сложное движение трёхмерной фигуры.

- `Mailman` — программа поддержки списков рассылки. Имеет поддержку всех необходимых возможностей: работа со шлюзом групп новостей, формирование дайджестов, ведение архивов и т.п.

- Два математических расширения питона: `Numeric` и `Scientific`. Первое помогает работать с матрицами различными численными методами, по возможностям сравнимо с системой Матлаб. Второе представляет из себя набор модулей, реализующих тензорное исчисление, статические процедуры, трёхмерную визуализацию и пр.

- `PyXML` и `4Suite` позволяют работать на питоне с такими современными технологиями, как `XML`, XPath, XSLT, SAX, DOM, RDF и ODS.
- `Sketch` и `PIL` — ещё два пакета работы с графикой. Первый — это просто векторный графический редактор, написанный на питоне, а второй — пакет для работы с различными растровыми форматами.

Кроме того, питон сильно теснит остальные языки: он широко используется как сценарный язык CGI, отвоёвывая место у перла; в стандартной поставке питона есть платформонезависимый модуль `Tk` для лёгкого построения графического интерфейса (раньше он использовался только в связке с языком тикль (TCL, Tool Command Language, язык командования инструментами)).

2.2 Работа с интерпретатором питона

Питон — интерпретируемый язык. Это значит, что термин *программа* эквивалентен термину *исходный тест программы*. Питон может работать в двух различных режимах: интерактивном и неинтерактивном.

В интерактивном режиме питон ведёт диалог с пользователем. Реплики самого питона не блещут разнообразием — они включают `>>>`, `...` и результаты введённых выражений. Первые две реплики — это приглашения, если последнее, что написано на экране — это `>>>`, можно смело начинать набирать новую команду. `...` означает, что набор команды ещё не кончен несмотря на переход на новую строку (возможно, не закрыта скобка или действительно выражение ещё не закончено).

Можно начинать вводить выражения питона или последовать выводимому на экран при запуске совету посмотреть права, благодарности или лицензию:

```
Python 2.1 (#15, Apr 16 2001, 18:25:49) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>> copyright
Copyright (c) 2001 Python Software Foundation.
All Rights Reserved.
Copyright (c) 2000 BeOpen.com.
All Rights Reserved.
Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.
Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>> credits
Thanks to CWI, CNRI, BeOpen.com, Digital Creations and a cast of thousands
for supporting Python development. See www.python.org for more information.
>>>
```

Лицензию смотреть не рекомендуем — ценной информации там нет, а занимает она несколько страниц. А вот советом заглянуть на http://

www.python.org пренебрегать не стоит, по этому адресу можно найти много полезной информации.

Неинтерактивный режим подразумевает существование программы, записанной в отдельный файл. Питон переходит в этот режим автоматически, если при запуске дать ему первым же параметром имя файла с программой (после выполнения программы управление вернётся операционной системе, а не интерпретатору!).

III
Лекция третья

Итак, узнав всё необходимое об обеспечении ЭВМ, о проектировании программ, об эволюции языков программирования и о том, как работать с интерпретатором питона, попробуем перейти к чему-нибудь более конкретному, а именно: разобрать простенькую программу на языке питон.

3 Типы данных и простейшие конструкции питона

3.1 Понятие переменной. Оператор присваивания

Вот типичный пример программы на питоне:

```
a = 1
b = 2
print "a + b = a+b
```

На первый взгляд программа проста и, хотя она на самом деле, возможно, ещё более проста, чем кажется, здесь есть о чём поговорить.

Первая строчка. У переменной по имени `a` появляется значение, равное единице.

Определение. Переменная есть имя, присвоенное одной или нескольким ячейкам памяти, содержащим некое значение.

У некоторых сразу могут возникнуть два вопроса:

1. Может ли одно и то же имя указывать на разные ячейки одновременно?

2. Может ли одна и та же ячейка памяти иметь одновременно несколько имён?

Для питона ответы соответственно: *нет* и *да*. Имя связано только с одним набором ячеек, а вот один и тот же набор ячеек может иметь сразу несколько имён. Позже станет понятно, что это справедливо только для сложных переменных: последовательностей и объектов, а строки и числа имеют только по одному имени.

Множество допустимых значений переменной — это её тип. Так, например, 25 — это целое число, π — вещественное, а «мехмат» — это строка. В питоне переменные могут в процессе жизни легко менять свой тип, поэтому он называется нетипизированнным языком. Конечно же, эта бестиповость не означает, что в питоне нет данных различных типов. Вовсе нет! Просто программисту не обязательно об этом задумываться. Таким образом, данные имеют тип, а переменные — нет.

Теперь поговорим о знаке равенства, стоящем между именем переменной и её будущим значением. Так обозначается оператор присваивания, один из важнейших операторов в большинстве языков. В одной книге по языкам программирования автор утверждал, что <есть только один оператор, который фактически что-то делает, — оператор присваивания. Все другие операторы... существуют только для того, чтобы управлять последовательностью выполнения операторов присваивания>. Мнение это спорное, но правильное.

Определения оператора присваивания мы давать не будем, а ограничимся описанием того, что происходит при его выполнении:

1. Вычисление значения выражения в правой части оператора (справа от знака равенства до конца строки).

2. Вычисление выражения в левой части оператора (выражение это должно однозначно определить адрес ячеек памяти).

3. Копирование значения из шага 1 в ячейки из шага 2.

На практике чаще всего слева стоит имя переменной, хотя имён может быть и несколько. Например, первые две строки нашей программы можно было объединить в одну:

```
a,b=1,2
```

Строгую теоретическую базу под возможность использования оператора присваивания таким образом мы подведём позже.

Кроме того, оператор присваивания позволяет присваивать *одно и то же* значение сразу нескольким переменным:

```
c=d=e=0
```

Третья строка программы содержит оператор вывода на экран.

3.2 Вывод данных

В данном случае будут напечатаны две вещи: строка `a + b =`, не претерпевшая изменений, и вычисленное значение выражения a+b. После этого курсор будет переведён на новую строку. Таким образом, на экране мы увидим:

```
a + b = 3
```

Пробел между двумя выведенными объектами оператор вывода вставляет автоматически, а на новую строку переходит только после вывода всех значений. Если это необходимо сделать в другом месте, програм-

мист должен либо использовать несколько операторов вывода, либо явно указать переход в виде "\n— в этом месте и будет разорвана строка. Так,

```
print "a +\nb = 5
```

выдаст:

```
a␣+
b␣=␣5
```

Обратный слэш вместе с последующей буквой называется управляющей последовательностью. Некоторые буквы не порождают такой последовательности и выводятся как есть, но во избежание сюрпризов стоит каждый обратный слэш набирать как "\\— эта простейшая управляющая последовательность используется для обозначение слэша как такового. Использование обратного слэша для ввода обычных символов называется маскировкой. Подробнее об этом мы поговорим, когда дойдём до символьного типа данных.

Если переход на новую строку не нужен даже в конце оператора `print`, следует после списка всех значений поставить дополнительную запятую:

```
print "one
print "two
print "three"
one␣two␣three
```

Иногда бывает необходимым сделать так, чтобы вывод был более красивым: сделать выравнивание по какой-то стороне текста, добавить пробелов и т.д. В питоне это можно делать, не выходя за пределы оператора `print`, причём нужно задать только поле, которое должно занимать значение переменной, а нужное количество пробелов оператор вывода вставит автоматически. Делается это так:

```
print '%-5d = %5d' % (25, 34)
```

Первым параметром идёт заключённая в кавычки строка, содержимое которой и определяет выводимый формат. Затем следуют все выводимые переменные или значения, перечисленные в скобках. Все символы форматирующей строки, за исключением символа процента (и следующих за ним числа и буквы), будут выведены как обычно. Сам процент называется форматирующим оператором. На место каждого из форматирующих операторов будет вставлено соответствующее значение следующим образом: число определяет количество экранных знакомест (для текстового режима) или пробелов (для графического), отведённых для значения. Если длина выводимого значения больше этого числа, пробелы не добавляются. Если же меньше, дописываются пробелы справа (если число отрицательное) или слева (если положительное) так, чтобы длина выведенной строки была равна заданному числу. Буква после числа означает формат вывода и может иметь значение `d` для целых чисел, `f` для вещественных или `s` для строк (или вывода чисел как строк). Таким образом, наше выражение будет напечатано так:

```
25␣␣␣␣=␣␣␣␣34
```

Каждое число отделяет от знака равенства не три, а четыре пробела — ещё один пробел мы сами вписали в форматирующую строку.

Для вещественных чисел имеется возможность задать нужное количество символов после запятой, округление будет произведено автоматически:

```
print 'Число пи примерно равно %5.3f' % 3.1415926535897931
Число␣пи␣примерно␣равно␣3.142
```

3.3 Ввод данных

Логично было бы предположить наряду с оператором вывода существование оператора ввода. И он действительно есть и называется `input`. Используется он следующим простейшим образом:

```
x=input()
```

При этом у пользователя спрашивается питоновское выражение, значение которого и заносится в переменную `x`. Это именно значение выражения, поэтому, например, если ввести `25+59`, в `x` будет передано `84`, а если попытаться ввести строку, питон выдаст ошибку — строки надо заключать в кавычки явным образом. Естественно, в этом выражении, как и в любом другом, можно использовать имена уже определённых переменных, на место которых будут подставлены их текущие значения.

Второй способ использования оператора ввода такой:

```
N=input('N=')
```

Строка, передаваемая оператору `input`, называется приглашением, она выдаётся на экран перед запросом выражения пользователя (который происходит совершенно точно так же).

IV
Лекция четвёртая

3.4 Целые числа и операции над ними

Понятно, что при объяснении даже нетипизированного языка приходится уделять некоторое внимание типам данных, в нём используемых. Основной простейший тип данных — это, конечно, целые числа.

Целые числа в питоне бывают двух типов: обычные и длинные. Обычные занимают только 32 бита и могут иметь знак, т§о есть каждое целое число есть число от `-2147483648` до `2147483647` включительно. Задаются они следующим естественным образом:

```
a=65535
```

Конечно, существует возможность задания целых чисел и менее естественными способами, например, в восьмеричном или шестнадцатиричном виде. В первом случае число должно начинаться с нуля (и быть не более

017777777777), во втором — с `0x` (ограничено сверху 0x7fffffff). Недесятичные числа также могут иметь знак, но отрицательные числа могут быть также заданы явным представлением (например, -20 — это 037777777754 или же 0xffffffec). Получить строки с восьмеричным или шестнадцатиричным представлением числа можно функциями `oct` и `hex` соответственно.

Длинные целые числа имеют неограниченную длину (точнее, ограниченую объёмом всей оперативной памяти, а это очень и очень много). Отличаются длинные целые от обычных целых буквой `L` после последней цифры. Например:

```
b=99999999999999999999999999999999999999999999999L
b=0xffffffffffffffffffffffffffffffffffffffffffffffffL
```

Все операции над целыми числами можно разделить на арифметические и логические. Арифметические — это с детства знакомые нам сложение (`+`), вычитание (`-`), деление (`/`, с округлением вниз), умножение (`*`) и возведение в степень (`**`), логические — это всевозможные побитовые операции НЕ (∼), И (`&`), ИЛИ (`|`), исключающее или (∧) и так далее. Для того, чтобы избежать путаницы между этими побитовыми операциями и логическими булевыми, последние были названы словами, а побитовым досталось по символу, их обозначающему.

```
c=2+3
c*=2
```

В последней сточке используется так называемое *присваивание умножением*, когда операция проводится напрямую с содержимым переменной `c`. Это эквивалентно:

```
c=c*2
```

Считается, что использование таких приёмов (которые существуют для всех операций: есть и присваивание делением, и возведением в степень, и исключающим или) экономит машинное время, позволяя более эффективно проводить вычисления, но очевидно, что это экономит как минимум несколько символов в исходном тексте программы.

3.5 Вещественные числа

Вещественные числа — это числа, имеющие наряду с целой ещё и дробную часть, которая приписывается справа от целой после точки. Они могут записываться тремя различными способами: в обычной, усечённой и экпоненциальной форме. Обычная форма подразумевает следующее:

```
d=-324.8451
e=2.7183
```

Усечённая форма позволяет записывать целые числа как вещественные (возможно, для того, чтобы к ним можно было применять вещественные функции). Индикатором «вещественности» числа в этом случае будет служить точка:

```
f=10.
```

При проверке на равенство такое число будет равно обычной, целой десятке, но при попытке разделить его на 100 мы получим не ноль, а одну

десятую.

Аналогичным образом можно опускать и целую часть, записывая только точку и следующие за ней цифры.

Экспоненциальная форма позволяет дописывать в конце множитель. Дело в том, что вещественные числа хранятся в памяти ЭВМ не точно, а приближённо только несколько первых знаков после запятой (обычно 16), и для того, чтобы сделать такое представление не совсем уж плохим, используется множитель. Например, число 123456789123456789 будет иметь такой вещественный вид:

$$123456789123456789 \approx 1.2345678912345678 \cdot 10^{17}$$

Его можно будет записать в питоне так:

```
g=123456789123456789.
```

или:

```
g=1.2345678912345678e17
```

Понятно, что если число имеет немного значащих цифр, но большой порядок, его запись в экспоненциальной форме наиболее легка. Сравните эти два присваивания:

```
h=100000000000000000000000000000000000000000000000000.
h=1e50
```

Они абсолютно эквивалентны, в чём легко убедиться, посчитав нули.

Побитовых действий над вещественными числами не производится также ввиду сложности их машинного представления. Арифметические действия выполняются без округления, хотя погрешность может возникнуть, если, например, результат содержит много разреженых чисел. Например:

```
k=1.
l=1e-20
m=k+l
```

После этого `m`, как это ни печально, будет равно `k`, а вся добавочная часть в лице `l`, способная на отдельное существование, канет в небытие.

В добавление к обычным операциям, для работы над вещественными числами написан специальный модуль под названием `math`. Подключив его, можно будет считать всевозможные синусы, косинусы и тангенсы с арктангенсами. Ещё он позволяет правильно округлять числа и брать логарифмы. Помимо этого, модуль содержит две константы: e и π. Модуль подключается ключевым словом `import`, и далее при использовании любого оператора и любой константы оттуда нужно к имени добавлять имя модуля. Список всего содержимого модуля можно получить удобной функцией `dir`:

```
>>> import math
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2',
'ceil', 'cos', 'cosh', 'e', 'exp', 'fabs', 'floor', 'fmod',
'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow',
'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

```
>>> math.cos(math.pi)
-1.0
>>>
```

3.6 Комплексные числа

Питон имеет встроенную поддержку комплексных чисел. Их можно задавать двумя способами: прямым или же функцией `complex`. Делается это так:

```
n=2+1j
p=complex(2,1)
```

Эти две строчки эквивалентны. Обратите внимание на то, что в питоне мнимая единица обозначается маленькой латинской `j` и никак иначе. Причём эта буква вопринимается таким образом только если написана непосредственно после числа, поэтому даже в нашем случае нужно писать `1j`, а не `j`.

Обе части комплексного числа (как реальная, так и мнимая) считаются вещественными, даже если заданы в виде целого числа. Комплексное число без мнимой части не перестаёт быть комплексным числом.

Все приёмы работы с комплексным числом можно узнать той же функцией `dir`, дав ей комплексное число в качестве аргумента. Они включают: `real` для получения реальной части числа, `imag` соответственно для мнимой и `conjugate()` для получения комплексного числа, сопряжённого данному. Модуль комплексного числа (как и модуль любого другого) можно получить функцией `abs()`.

3.7 Связь между числами, связь между операциями

Если в одном и том же выражении встречаются несколько типов чисел, то результат имеет самый сильный тип из всех встречающихся в выражении. Самым сильным числовым типом считается комплексный, за ним идёт вещественный, затем длинный целый и только потом целый. Таким образом, присутствие дробной части считается более важным, чем точность большого числа. Стоит быть осторожным, если для вас это не так, и округлять все вещественные числа перед добавлением к длинным целым.

Если в одном и том же выражении встречаются несколько операций, то они, конечно, выполняются все, причём в определённой последовательности. Она жёстко определяется приоритетами операций и порядком их следования в выражении. Приоритеры таковы: самый высокий у возведения в степень, затем идёт логическое отрицание, затем вместе умножение и деление, и только потом сложение и вычитание, после них коньюнкция, потом исключающая дизъюнкция, и лишь после неё обычная дизъюнкция. В случае равных приоритетов вычисление идёт слева направо. Для изменения этого порядка выполнения используются скобки. Их следует использовать во всех сомнительных ситуациях, не перекладывая основной смысл выражения на приоритеты.

3.8 Строки

Вообще говоря, мы уже закончили рассмотрение всех тех типов, которыми ограничивался набор типов данных в автокодах и даже первых языках программирования. Но вскоре стало очевидным, что не менее важна в прикладных программах возможность обработки нечисловой информации. Сегодня, конечно, количество текстовых редакторов и процессоров, систем управления базами данных, программ-переводчиков и прочих пакетов символьной обработки существенно превосходит количество сугубо математических пакетов.

С точки зрения разработчика программного обеспечения обработка текста чрезвычайно сложна из-за разнообразия естественных языков и способов их записи. С точки зрения языков программирования обработка текста куда проще, так как подразумевается, что в языке набор символов представляет собой короткую и упорядоченную последовательность значений. Фактически, за исключением языков с большим числом букв (восточных: китайского, японского) и языков с большим числом начертаний одной и той же буквы (семитских: арабского, мальтийского) хватает 256 значений одного байта. Последнее время всё чаще используется Уникод (Unicode) — двухбайтовая кодировка, содержащая сразу все мыслимые символы.

Строки в питоне не сильно напоминают строки в других языках программирования засчет отсутствиея типа *символ*. Обычный способ задания строкового типа состоит во введении символа и представления строк как последовательностей (массивов) символов. В питоне же символ — это строка длины 1. Таким образом, нет смысла во введении двух разных символов: обычного и уникодового, символ мы можем рассматривать как элемент соответствующей строки и не более того.

Итак, раз уж мы не можем сказать, что строка есть последовательность символова, придётся признать такое **определение**: строка есть нечто, заключённое в кавычки. Обычно используют двойные кавычки, если строка содержит одинарные внутри себя и одинарные в противном случае. Вот примеры присваивания строк:

```
a="строка"
b='ещё␣одна␣строка'
c='Он␣сказал:␣"Да"'
d="O'Хара"
```

Конечно, рано или поздно должна будет встретиться строка, содержащая оба типа кавычек. Что же делать в этом случае? Есть ещё обратные кавычки, но они уже нагружены другим смыслом, о котором чуть ниже. Поэтому используется так называемая маскировка — перед запретным символом ставится обратный слэш:

```
e='"Isnt\'t␣it?"␣-␣she␣asked.'
f="\"It␣is\"␣-␣he␣replied."
```

Видно, что маскировка — это не только средство разрешения конфликтов между кавычками, но и просто удобный в некоторых ситуациях механизм. Вообще, программист не сильно связан в этом вопросе и может

по своему усмотрению решать, когда и какие кавычки использовать.

Маскировка — одна из возможностей использования управляющих последовательностей, о которых мы уже говорили раньше, при обсуждении оператора вывода и перехода на новую строку.

Если одиночный обратный слэш завершает строчку, на следующей строчке будет ожидаться продолжение строки, но сам переход записан не будет. В этой фразе ярко проявляется несовершенство терминологии. Строки (strings) — это переменные, содержащие символьные данные, которые мы обсуждаем в этой теме. Строчки (lines) — это строки экрана, по которым идут знакоместа в текстовом режиме и символы текущим шрифтом в графическом. Мы надеемся, что в каждом конкретном случае смысл будет ясен из контекста.

Переход на новую строчку можно включать в строки так же, как мы это делали для оператора `print`: как \n. Но есть ещё один способ включить переход на новую строчку в явном виде. Для этого есть ещё два варианта заключения в кавычки. Нет, это не широко известные обратные, до которых мы всё ещё не дошли, а особые, существующие только в питоне: тройные. Они бывают двух типов: тройные двойные и тройные одинарные (тройные одинарные были введены сугубо ради симметрии и почти не используются). Выглядят они так:

```
g="""Строка,
␣явным␣образом
разбитая␣на
строчки"""
```

Ну и, наконец, перейдём к обратным кавычкам, потому как это последний их тип.

```
h=53
k=‘h‘
```

Что после этого находится в `k`? Строка, конечно. А что в строке? Значение переменной `h`, то есть `"53"`. Итак, обратные кавычки *возвращают преобразованное в строку значение переменной, имя которой записано между ними.* Другой, менее экзотичный способ совершения того же действия выглядит так:

```
k=repr(h)
```

Согласитесь, длиннее, труднее для запоминания и менее красиво. Другая от частого употребления ставшая стандартной операция со строками — это взятие её длины:

```
l=len(k)
```

Упражнение. Попробуйте угадать, что будет в `m`, `n` и `p` после выполнения следующего:

```
m,n=3.1415,"""\\\\\\\"\"␣’’’␣"""␣’’’␣"""␣’’’
p=len(‘n‘+’m’+’n’+‘m‘)
```

Правильный ответ: 3.1415, `\\\""␣’’’␣␣"""` и 43 соответственно.

V
Лекция пятая

Разобравшись с кавычками, мы можем перейти к скобкам. Их существует четыре вида: круглые, квадратные, фигурные и угловые. Три из них служат для определения трёх важнейших сложных типов данных питона.

3.9 Композитные типы данных

Существует два сложных, или композитных, типов данных в питоне: последовательности и объекты. Сегодня мы разберёмся с тремя разновидностями последовательностей.

Определение. Последовательность есть нечто, заключённое в скобки.

1. Кортеж *есть неоднородный неизменяемый массив. Задаётся круглыми скобками или же их отсутствием.* Ну, неоднородный — это понятно, значит, может содержать разнотиповые данные, например:

   `A=(2,3.14,"aaa")`

   `B=((((((1),0),0),0),0),0)`

   Неизменяемый — это сложнее. Это значит, что структура кортежа не может быть изменена после того, как он был создан. (Как будет выяснено далее, кое-что можно всё же сделать в обход ограничений). В питоне только строки и кортежи являются неизменяемыми типами данных. Так, нельзя заменить одну букву в строке, оставив саму строку той же, но можно создать новую строку с одной изменённой буквой.

   Для доступа к элементам кортежа используются квадратные скобки с указанием номера нужного элемента:

   `C=A[1]`

   В этом случае в `C` будет занесена не двойка, а 3.14, потому что *нумерация элементов всегда идёт с нуля.* Также можно из кортежа взять часть с несколькими элементами, называемую сечением. Сечения бывают трёх видов: начальные, центральные и конечные. Рассмотрим различия между ними на примерах:

   `D=(1,2,3,4,5,6,7,8,9)` даёт (1,2,3,4,5,6,7,8,9)

   `E=D[3:8]` даёт (4,5,6,7,8)

   `F=D[:4]` даёт (1,2,3,4)

   `G=D[7:]` даёт (8,9)

   В первой строчке мы занесли в `C` какой-то произвольный кортеж, удобный для демонстрации различных сечений. Во второй строчке берётся

центральное сечение — с третьего элемента включительно по восьмой невключительно. Следует отметить, что это обычный для питона метод обхождения с границами чего бы то ни было — нижняя граница всегда входит в диапазон, а верхняя — нет. Это не обусловлено никакими теоретическими выкладками, а только практическим удобством использования. Ну и, конечно, ни на минуту нельзя забывать, что нумерация элементов идёт с нуля! В третьей строчке мы опустили первое число, и оно по умолчанию приняло значение 0 — номер первого элемента, что дало нам начальное сечение. Ясно, что конечное сечение получается при опускании последнего индекса, принимающего номер на один больший номера последнего элемента (то есть так, чтобы последний элемент вошёл в сечение, а не остался непонятно где).

У некоторых логично мыслящих может возникнуть вопрос: а что, если опустить оба числа? Правильный ответ таков: результатом будет полное сечение или копия исходного кортежа. Такой ответ ожидаем, но не вносит ясности, появления которой мы так жаждали при формулировке вопроса, и даже наоборот, он запутывает ситуацию, порождая новый вопрос: в чём разница между `H=D` и `H=D[:]`? Ответ: **в семантике!**

Дело в том, что в питоне для сложных типов данных (то есть не строк и не чисел) оператор присваивания работает совсем по-другому. Вместо пересылки содержимого одних ячеек памяти в другие происходит дополнительное именование *тех же самых* ячеек. Таким образом, разные для нас имена трактуются как один и тот же набор ячеек питоном. Это называется семантика указателей.

Количество имён объекта[1] называется его мощностью. Для уменьшения мощности используется оператор `del`. Когда мощность объекта опускается до нуля, объект потерян, мы больше не имеем к нему доступа, и в ближайшее время он будет уничтожен интерпретатором питона. Все строки и числа имеют мощность 1 и уничтожаются сразу по вызову оператора `del` или при получении именем нового значения. Запись `H=D[:]` олицетворяет семантику копирования. Создаётся новый объект, полностью копирующий структуру и содержимое старого, и мы получаем два одинаковых (точнее, разных, но равных) объекта мощности 1 каждый.

Для преобразования строки или последовательности в кортеж используется функция `tuple()`:

```
>>> tuple('123')
('1','2','3')
```

Если аргумент этой функции — кортеж, она вернёт именно его (а не его копию).

[1] Объектом мы пока что называем множество ячеек памяти

2. Список *есть изменяемый неоднородный массив. Задаётся квадратными скобками.*

   Список — это более обычный для опытных (испорченных другими языками) программистов, встречающийся во многих языках программирования высокого уровня. Сечения берутся подобным же образом:

   `K=range(1,10)` даёт [1,2,3,4,5,6,7,8,9]

   `L=K[2:]` даёт [3,4,5,6,7,8,9]

   В первой строчке стандартной функцией питона мы создали список последовательных целых чисел от 1 до 10 (как обычно, первое число вошло в результат, а второе — нет). Эта функция чрезвычайно полезна и, как сказал бы на нашем месте философ, если бы её не было, её стоило бы придумать. Полную её мощь мы сможем вкусить на следующей лекции, когда доберёмся до операторов циклов. Пока же вернёмся к последовательностям.

   Во второй строчке берётся конечное сечение — с третьего элемента по последний включительно. Прочие типы сечений берутся аналогично.

   Кроме того, мы можем изменять значения элементов списка:

   `K[5]=5` даёт в K [3,4,5,6,7,5,9]

   Здесь следует быть осторожным и различать семантику копирования и семантику указателей. Например, если мы напишем:

   ```
   M=[3,4,5]
   N=M
   N[0]=333
   ```

   То какое значение окажется в `M`? Правильно, `[333,4,5]`, потому что какое бы имя мы не использовали: `M` или `N`, обращение идёт к одним и тем же ячейкам оперативной памяти.

   Воспользуемся уже известным методом для определения операций над списком — функцией `dir`:

   ```
   >>> dir([])
   ['append', 'count', 'extend', 'index', 'insert', 'pop',
   'remove', 'reverse', 'sort']
   ```

   Эти функции служат соответственно для добавления элемента в конец готового списка; для подсчёта количества элементов списка, равных данному; для расширения этого списка другой последовательностью (в т.ч. и кортежем) добавлением всех его элементов в конец этого; для действия, обратного взятию элемента — определения индекса элемента по его значению; `N.insert(2,3)` позволяет вставить новый элемент не в конец списка, а на определённое место, номер места идёт первым параметром, значение — вторым; `pop` позволяет *вытолкнуть* последний элемент из списка (при этом сам список становится короче);

`remove` удаляет элемент, значение которого ему дано; `reverse` обращает список (последний элемент теперь идёт первым); `sort` сортирует список по возрастанию. Для сортировки по убыванию, конечно, можно использовать комбинацию `sort` и `reverse`.

Упражнение. Запустите функцию `dir` для кортежа и объясните результат.

Для преобразования строки или последовательности в список используется функция `list()`:

```
>>> list('123')
['1','2','3']
>>> list((1,2,3))
[1,2,3]
```

Если аргумент этой функции — список, то она создаст копию и вернёт именно её (а не его исходный список). Таким образом, если `P` — список, то `P=list(Q)` эквивалентно `P=Q[:]`.

Почему для кортежа копия не создавалась, а для списка создаётся? Да просто никто не будет изменять кортеж, и нет смысла хранить два одинаковых неизменяемых объекта.

Вдумчивому слушателю (читателю) не даст покоя очередной вопрос: имеют ли кортежи право на существование, раз всё их отличие от списков заключается в неудобстве, связанном с невозможностью изменения структуры? Разница между кортежем и списком философская, её можно уподобить разнице между джазом и блюзом. И то, и другое — музыка, которую могут исполнять одни и те же музыканты на тех же инструментах, но разницу можно уловить невооружённым ухом. Суть этих стилей принципиально разная: джаз — это импровизация, полёт фантазии, экспромт, а блюз — это крик души испытываемого судьбой человека. Список — это прежде всего нумерованая последовательность, кортеж — прежде всего упорядоченная. Координаты точки в пространстве — это кортеж; перечень фамилий студентов — список. Цветовые составляющие пикселя — кортеж; строчки, введённые из файла — список. Конечно, удобства и полноты ради и список упорядочен, и кортеж пронумерован, но это уже вторично. Безусловно, никто (и даже сам Гвидо ван Россум) не в силах вам помешать использовать списки вместо кортежей везде, где они только встречаются — начинающие программисты, замученные паскалем, бейсиком и явой, так и делают, но это (кроме замедления работы программы) есть ни что иное, как преступление против философии — самое тяжкое из всех преступлений, а если это и что-то другое, то демонстрация собственного невежества, некомпетентности и неумения грамотно пользоваться предоставляемыми средствами. Тем не менее, даже дав зарок никогда не пользоваться кортежами, вы рискуете вскоре его нарушить, даже не подозревая об этом. Например, в следующем случае:

`R,S=2,3`

В этом случае на лету создаётся кортеж, прозрачный для программиста, и тут же уничтожается за ненадобностью. Нет необходимости ни нумеровать элементы (кроме как для того, чтобы знать, в какой последовательности они идут), ни реализовывать возможность последующего изменения элементов, их добавления, сортировки и т.д. — легко, просто, быстро, доступно. Аналогично используется так называемая декомпозиция кортежа:

`T=1,2,3`

`U,V,W=T`

Теперь понятно, как реализуется предыдущий пример: кортеж сначала создаётся, а потом декомпозиционируется.

А о каких возможностях мы говорили, заявляя, что как-то можно обойти ограничения, наложеные изобретателем кортежей? Есть одна лазейка в его **определении**, позволяющая кое-что сделать. Сказано — нельзя менять структуру. Изменение самих элементов кортежа может существенно изменить структуру, потому и запрещено. Но представьте, что один из элементов кортежа — список. Можно ли менять его элементы? Да, конечно!

`X=[2,3,4],5,6`

`X[1]=4` — нельзя

`X[0][1]=33` — можно

3. Словарь *есть ассоциативный изменяемый неоднородный массив. Задаётся парами ключ: значение, перечисленными в фигурных скобках.*

   Словарь — это квинтэссенция программистской мысли, направленной на изучение последовательностей, это массив, в котором элементы пронумерованы не подряд идущими целыми положительными числами, а чем угодно: строками, вещественными числами, кортежами. Конечно, это не может быть сделано списками (которые меняют свою структуру) и словарями. Индекс элемента словаря называется ключом.

   Сечений словарь не поддерживает, а для создания копии нужно написать в явном виде:

   `Y={"one":1,"two":2}`

   `Z=A.copy()`

   Полное очищение словаря производится функцией `clear()`, добавление новой пары — простым присваиванием:

   `Y["three"]=3`

   При попытке узнать значение по несуществующему ключу выдаётся ошибка и работа программы останавливается, поэтому нужно почаще пользоваться функцией `has_key()`, определяющей, есть ли такой

ключ в данном словаре. Кроме получения списка ключей (`keys()`) и списка значений (`values()`) весьма полезно **представление словаря как списка кортежей** функцией `items()`. Понятно, что здесь философия соблюдена — легко что-то добавить в полученный *псевдокортеж* или отсортировать его, а каждой паре это ни к чему — всё, что нам нужно знать, это где ключ и где значение, ему соответствующее. Это с лихвой обеспечивается кортежем.

Прочие возможности и приёмы работы со словарём можно узнать у функции `dir({})`.

Ясно, что уже данное нами определение типа при нынешнем уровне знаний не выдерживает никакой критики, и нужно давать его заново и подругому:

Определение. Тип есть совокупность множества значений и методов для работы с ними.

Упражнение. Множество допустимых значений типа — это список или кортеж? А сам тип?

Существует большое разнообразие типов, необходимых в самых разных областях применения программирования. Среди наиболее интересных можно вспомнить множества — когда важно присутствие элемента, но не важен его порядок, и один элемент может присутствовать только в единственном экземпляре. Для графов существует много разных машинных представлений: матрицы инцидентности и смежности, объединение множества дуг и множества вершин и т.п. Деками называют массивы, в которых доступ может производиться только к крайним элементам, по одному с каждой стороны (такая же, но одностороняя структура называется стеком). В некоторых задачах удобно пользоваться кольцами — замкнутыми массивами, в которых доступ осуществляется только к одному элементу кольца и для перехода к другому кольцо нужно *прокрутить*. Всё это — сильно специализированные вещи, не входящие в стандартную поставку питона, но их можно реализовать с помощью объектной модели, о чём мы и узнаем через несколько лекций.

VI
Лекция шестая

Несмотря на то, что питон — язык нетипизированный, мы и в этой лекции рассмотрим ещё один тип данных и операторы, им порождённые.

На практике иногда оказывается, что типы, кажущиеся на первый взгляд менее важными, да и по определению скромнее уже рассмотренных, во много крат мощнее и нужнее. Таков, например, так называемый логический тип, называемый иногда булевым в честь ирландского математика Джорджа Буля.

3.10 Логический тип

Логический тип, как можно догадаться, состоит всего из двух возможных значений: истины и лжи (англоязычные источники пользуются терминами *true* и *false* соответственно). В питоне нет самостоятельного булева типа даже на том уровне, где можно признать существование целого и вещественного типов данных (ведь, вообще говоря, нет никаких типов, так, теория одна). В качестве булева типа возможно использование любого другого по следующим правилам:

- Операции отношения, такие, как `>`, `==` или `!=`, возвращают `1` (значение целого типа), если отношение выполняется и `0` в противном случае.
- При использовании значения какого-либо типа в качестве булева только всевозможные нули и пустые списки (то есть `0, 0.0, 0L, 0j, (), '', , '""', , [], {}`, а также особый пустой объект `None`, с которым мы ознакомимся позже) считаются ложью, все прочие — истиной.
- Выражение `A or B` (A или B) возвращает B, если A ложно и A в противном случае.
- Выражение `A and B` (A и B) возвращает B, если A истинно и A в противном случае.
- Выражение `not A` (не A) возвращает 1, если A ложно и 0 в противном случае.

Изменить это (если кому вдруг захочется) нельзя, а создать новый тип с похожими свойствами можно только пользуясь объектно-ориентированным подходом, о чём мы расскажем позже.

Булев тип чаще всего используется при различного рода проверках, в операторах ветвления, которые мы сейчас рассмотри. Когда мы говорили об операторе присваивания, было упомянуто существование операторов, управляющих последовательностью их выполнения. Теперь же мы обсудим их подробно.

В языке питон существует три вида таких операторов: ветвление, повтор и перебор. Оператор ветвления записывается так:

```
if <условие>: <оператор>
```

или так:

```
if <условие>:
␣<оператор>
```

После ключевого слова `if` записывается условие (для наглядного отделения обычно используют круглые скобки, но можно обходиться без них). Вообще, скобок можно ставить сколько угодно, питон не спутает число в скобках с кортежем из одного элемента, ведь такой кортеж должен в задании иметь ещё и запятую: `a=(0,)`.

После двоеточия указывается оператор, который будет выполнен в случае истинности условия. Если в случае истинности нужно выполнить не одну, а несколько строк кода, используется так называемый составной оператор, обозначаемый отступом:

```
if␣(A):
␣B=input()
␣print␣A+B
```

Отступом может служить как пробел, так и символ табуляции. Составной оператор (а с ним и оператор ветвления) кончается перед следующей строкой без отступа.

Для сравнения величин многих типов (чисел, строк, ...) используются привычные математические символы: `<` для *меньше*, `>` для *больше*, `>=` для *больше или равно*, `<=` для *меньше или равно*, `==` для *равно*, `<>` или `!=` для *не равно*. Их можно группировать по всем правилам арифметики:

```
if 0<x<10 and -10<=y<=100:
 print y%x
```

Есть ещё две инфиксных (то есть записывающихся между операндами) операции сравнения: `is` и `in`. Первая используется в основном для сравнения объектов на эквивалентность, для более простых же типов данных она аналогична `==`. Вторая проверяет элемент на принадлежность последовательности (кортежу, списку или строке). Есть краткая запись для `not (e in L)`:

```
e not in L
```

Оператор ветвления можно продлить, добавив секцию, срабатывающую при **ложном** условии. Повторно указывать условие не нужно:

```
if (<условие>):
␣<операторы>
else:
␣<операторы>
```

Развивая заложенную в этом маленьком усовершенствовании большую идею, можно придти к так называемому множественному ветвлению, когда в случае неудачи одного условия проверяется другое, при его неудаче — третье, четвёртое, и так далее. В питоне это записывается следующим образом:

```
if (<условие>):
␣<операторы>
elif (<условие>):
␣<операторы>
elif (<условие>):
␣<операторы>
else:
␣<операторы>
```

Логические операции, которым мы дали определение в начале лекции, помогают существенно сократить или даже полностью избежать появления одинаковых блоков программы или одинаковых условий:

```
if (A):
 print "!"
elif (B):
 print "!"
else
 print "?"
```

$\Longrightarrow$

```
if (A or B):
 print "!"
else
 print "?"
```

Совет, данный нами при описании приоритетов различных операций, остаётся в силе и здесь: не жалейте скобок для того, чтобы сделать выражение более удобным и читабельным для вас — питон всё поймёт и всё простит, но простите ли вы себя сами через месяц, пытаясь разобраться в мудрёных условиях?

3.11 Комментарии

Комментариями называют части программы, не интересующие интерпретатор. В питоне есть два варианта комментариев: однострочные естественные и многострочные синтаксические. Комментарии первого типа начинаются символом # и завершаются переходом на новую строку.

```
i = 1 #этого питон уже не видит
```

Комментарии второго типа представляют собой строку, записанную без всякого присваивания. В случае прямой работы с интерпретатором в диалоговом режиме эта строка будет выдана на экран, но при выполнении программы из файла она не попадёт никуда:

```
j = 1+i
"
Комментарий, поясняющий,
что в этом месте программы
переменная j
получила инкрементированное значение
переменной i
"
```

В новых версиях питона этот возникший чисто синтаксический механизм обмана интерпретатора получил более оправданное применение. Например, при определении функции комментарий записывается в специальную связанную с ней переменную `func_doc`.

4 Циклы и функции

4.1 Оператор перебора и оператор с предусловием

Оператор перебора позволяет применять одну и ту же последовательность операторов ко всем значениям последовательности. Записывается он так:

```
for x in (1,3,5,7,11,13,17,19):
␣<операторы>
```

при выполнении этого кода операторы будут выполнены столько раз, какова длина последовательности (в нашем случае это 8) и каждый раз `x` будет иметь значение очередного элемента последовательности: 1 на первом витке, 3 — на втором, 5 — на третьем, и т.д. Питон позволяет выполнять оператор перебора относительно нескольких переменных:

```
for x,y in ((1,2),(3,4),(5,6)):
␣<операторы>
```

При этом на каждом проходе пара `x` и `y` (точнее, кортеж, состоящий из этой пары) будет принимать значение соответствующей пары последовательности. Если структура последовательности не подходит, интерпретатор питона выдаст ошибку: Распаковка не-последовательности (*unpack non-sequence*).

Конечно, каждый раз указывать все значения — дело достаточно утомительное, поэтому в питоне есть встроеная функция `range()`, генерирующая список последовательных целых чисел в нужном интервале. С этой функцией мы мельком познакомились ещё в прошлой лекции. Эту чрезвычайно полезную функцию можно использовать тремя способами:

`range(n)`

создаст список целых чисел от 0 включительно до `n` невключительно.

`range(f,t)`

создаст список целых чисел от `f` включительно до `t` невключительно.

`range(f,t,s)`

создаст список чисел из интервала [`f`,`t`) вида *f, f+s, f+2*s, ...* — может быть полезно при использовании вещественных чисел.

При использовании чрезвычайно больших списков ради экономии памяти можно воспользоваться функцией `xrange()`, которая, работая абсолютно аналогичным образом, не вычисляет сразу значение каждого элемента итоговой последовательности, а создаёт определённый объект, элементы которого вычисляются только при непосредственном обращении к ним. Математик сказал бы, что `range()` реализует абстракцию актуальной бесконечности, тогда как `xrange()` — абстракцию потенциальной достижимости.

Если ваш список содержит несколько миллионов элементов, а одновременно нужны из них бывают только два или три, вы сможете заметить разницу в скорости выполнения программы при переходе с `range()` на `xrange()` невооружённым взглядом. Например, программа

```
from whrandom import choice
from time import clock
beg=clock()
A=range(30000000)
b=choice(A)
print clock()-beg
```

выполняется на компьютере AMD Duron 750MHz с 256Mb оперативной памяти и операционной системой Windows за 65-75 секунд, не считая пяти, а то и десяти минут выгрузки интерпретатора операционной систе-

мой, тогда как версия с `xrange()` выполняется за немногим более одной десятитысячной доли секунды.

Упражнение. Придумайте пример, когда время работы программы не может существенно измениться при переходе с `range()` на `xrange()`.

Но вернёмся к операторам циклов. Более высокоуровневую абстракцию повторяющихся операторов представляет собой цикл while — цикл с предусловием. При его использовании вместо прямого перечисления всех пробегаемых значений переменной цикла программист формулирует условие, которое остаётся истинным, если нужно выполнять итерацию и становится ложным в противном случае. Существуют языки программирования, специально ориентированные на такие условия (там они называются инвариантами). Все алгоритмы для программирования на подобных языках должны быть переформулированы с определение инварианта для каждой не единожды выполняемой строчки. Так далеко решаются заходить немногие, но циклы с условиями уже успели стать неотъемлимой частью всех алгоритмических языков.

Записывается цикл с предусловием так:

```
while <условие>
␣<операторы>
```

И, пока (а именно так, как вы знаете, переводится слово while) условие будет истинно, операторы будут выполняться ещё и ещё. Интерпретатор действует следующим образом: сначала проверяется условие и, если оно ложно, управление передаётся оператору, следующему за циклом while (говорят: *происходит выход из цикла*). Если же условие истинно, выпдняются все операторы цикла (которые, как известно, находятся в отступе относительно самого оператора), после чего опять проверяется условие и в случае его истинности всё повторяется с начала, а в случае ложности происходит выход из цикла.

Очевидно, что цикл

```
while (1): ...
```

будет вечным (из него никогда не будет выхода), а цикл

```
while (0): ...
```

не будет выполнен ни разу. Цикл не может быть пустым, в случае необходимости используют ничего не делающий оператор pass:

```
while (1): pass # вечное бездействие
```

Для экстренного выхода из цикла также существуют особые методы. Для безусловного выхода используется всевдооператор break:

```
while (1):
␣i/=10
␣if␣(!i):␣break
```

Теперь становится обоснованным применение вечных циклов, не так ли?

Для условного выхода (ещё называемого продолжением вычислений) используют continue. Встретив этот псевдооператор, интерпретатор передаёт управление в точку, где происходит проверка условия цикла. Таким

образом, при ложном условии continue вызывает выход из цикла, а при истинном — очередной виток вычислений.

Ясно, что средства break и continue применимы и к циклу перебора: первый прерывает цикл, а второй вызывает новый виток вычислений, если текущее значение переменной цикла не последнее, иначе также завершает перебор.

Оператор перебора и цикл с предусловием слабо эквивалентны, то есть для каждого конкретного условия будет достаточно легко перейти от одного типа цикла к другому, а общее преобразование куда сложнее. Из for в while можно построить автоматическое преобразование, которое будет неэффективным, а из while в for это вообще осуществить невозможно. Несмотря на столь очевидную связь, подобные мысли о взаимозаменяемости for и while концептуально категорически недопустимы. Эти циклы соответствуют абсолютно разным подходам к реализации вычислений: немедленные (for) и т.н. отложенные (while) вычисления. В качестве другого примера отложенных вычислений можно привести уже изученную нами функцию `xrange()`.

VII
Лекция седьмая

4.2 Понятие подпрограммы

В наше время существуют два принципиально разных подхода к реализации подпрограмм:

1. **Процедура** — это имеющая собственное имя часть программы, которая при вызове получает некоторые параметры и в соответствии с ними изменяет окружение, после чего возвращает управление в точку вызова. Процедура также может изменять собственные параметры (если их больше нуля). Такой тип подпрограммы широко используется в архаичных языках программирования (Фортран, Паскаль) и подчас (в том случае, если изменение окружения эквивалентно передаче данных через переменную, как в языке Форт) полностью равносилен следующему.

2. **Функция** — это имеющая имя часть программы, которая при вызове получает некоторые параметры и в соответствии с ними возвращает своё значение, не меняя окружение. Это определение куда ближе к математическому понятию функции.

В широко распространённых языках программирования эти подходы присутствуют, будучи перемешаны в том или ином соотношении. Паскаль, например, использует термины *процедура* и *функция*, но функции в нём могут изменять окружение. В Си++ любая подпрограмма является функцией,

но может возвращать значение типа `void` (пусто), превращаясь таким образом в процедуру. В Лиспе процедур мало, все они стандартны (например, процедуры вывода на экран) и называются *псевдофункциями*.

Существуют, безусловно, языки программирования, идущие в следовании тому или иному подходу дальше, чем этот подход планировал. Например, в языке ассемблера совсем не обязательно возвращать управление из процедуры, либо это можно сделать в место, отличное от точки вызова. В чисто функциональных языках (Лисп, МЛ, КЛОС) функции не нужно (хотя и можно) иметь имя.

Также понятно, что хорошо бы остановится где-то посередине, имя возможность применять подходы сообразно стоящей задаче. Одну подпрограмму, огранизующую соединение с сервером для дальнейшего обмена данными, логично было бы организовать процедурой, а другую подпрограмму, вычисляющую синус, — функцией.

В питоне процедуры и функции определяются весьма сходными конструкциями, но используются, конечно, по-разному. Вот так определяется процедура:

```
def␣becool(boy):
␣print␣boy,'is␣cool'
```

А так используется:

```
becool('Python')
```

Вот так определяется функция:

```
def␣logn(n,x)
␣return␣log(x)/log(n)
```

а так используется:

```
print logn(20,x)+sin(x)
```

Как видно из определений, ключевое отличие состоит в слове return — это и есть то самое *возвращение значения*, о котором уже была речь. Его формат таков:

```
return <значение>
```

Можно возвращать и несколько значений, перечисляя их через запятую — в этом случае питон, не изменяя самому себе, возвращает единым значением неявно создаваемый кортеж. Такую функцию можно использовать двояко:

```
def powers(x):
 return x*x,x*x*x,x*x*x*x
X=powers(2)
X2,X3,X4=powers(3)
```

В первом случае в `X` загружается целиком весь кортеж, во втором же — он декомпозиционируется и распадается на три элемента. При этом используются обычные правила присваивания кортежей, рассмотренные нами ранее.

Питон позволяет пользоваться подпрограммами, меняющими свою сущность от запуска к запуску. Например, вот так:

```
def br(a):
 if (a):
```

```
  return '('+str(a)+')'
 else:
  print "Error in br("+`a`+')'
```

При использовании функции как процедуры в программном режиме её значение теряется, а в интерактивном — выдаётся на экран. При использовании процедуры как функции считается, что она автоматически возвращает значение `None`.

Если вы хотите стабильно использовать вашу подпрограмму как функцию, позаботьтесь о том, чтобы при любом прохождении через её тело встречался только один return.

Следует твёрдо помнить, что return кроме возвращения значения прерывает выполнение функции (и возвращает управление в точку вызова) и производить все необходимые вычисления до него. Экстренный возврат из процедуры может быть записан как return без параметров:

```
return
```

Рассмотрев оператор `def`, мы затронули один важный момент, без разбора которого было бы немыслимо идти дальше.

4.3 Область действия имён переменных

Что будет, если в программе объявить некую переменную, а потом внутри функции попробуем ей воспользоваться? Получится у нас изменить её значение? Правильный ответ: просто так не получится, но можно, если постараться.

Под термином *область действия имён переменных* мы будем понимать область видимости используемых переменных. В более ранних языках программирования, часть из которых уже канула в Лету, а часть каким-то образом зацепилась за действительность, глобальные переменные были единственным средством создания переменных. Если даже переменная создавалась внутри функции, это просто была ещё одна глобальная переменная. Позже появилась концепция локальной переменной, не видной снаружи. Глобальные переменные всё же могли быть как прочитаны, так и изменены любой функцией. Отношение классиков теории программирования к этому вопросу не было единогласным. Эдсгер Дейкстра считал использование глобальных переменных в подпрограммах одним из самых ужасных нарушений дисциплины программирования, а Альфред Ахо при создании компиляторов не только допускал глобальные переменные как средство передачи информации от подпрограммы к подпрограмме, но и, можно сказать, пропагандировал его своими исходниками. Такие споры связаны с тем, что использующая глобальные переменные подпрограмма не является замкнутой системой, и при разных запусках с одинаковыми параметрами может возвращать разные результаты. С математической точки зрения, это превращает язык программирования в язык, порождаемый контекстно-зависимой грамматикой. Мы не будем вдаваться в лингвистические теории, но переход от контекстно-свободных грамматик к контекстно-зависимым существенно усложняет работу проектировщику компилятора.

Питон в этом плане более прогрессивен. В каждый момент выполнения программы (или ввода инструкций в интерактивном режиме) в питоне существуют две области действия имён переменных: глобальная и локальная. Первая относится к программе в целом, вторая — к текущей подобласти: телу функции, содержанию объекта, и т.д. Без дополнительных телодвижений изнутри функции глобальные переменные **не видны**. Например, после выполнения следующей функции:

```
x=1
def change(): x=-1
change()
```

значение глобальной переменной `x` не изменится. Вместо этого в локальной области будет создана новая переменная, имя которой совпадёт с именем глобальной переменной. Если же попытаться проверить значение глобальной переменной до попытки изменения её значения, будет выдано значение именно глобальной:

```
x=1
def trytoget(): print x
trytoget()
```

Таким образом, чтение глобальных переменных не считается питоном нарушением стиля, а запись в них данных — считается. Но, используя оператор `global`, можно обойти и это ограничение. Так,

```
x=1
def incr():
 x+=1
incr()
```

вызовет ошибку «обращение к локальной переменной до первого присваивания» (*local variable 'x' referenced before assignment*), а:

```
x=1
def incr():
 global x
 x+=1
incr()
```

будет работать. При определении вложенных функций локальная область не становится глобальной для подфункции, так что локальные области вложенных друг в друга подпрограмм не коррелируют.

Функции в питоне являются полноправными типами данных, поэтому их можно присваивать друг другу:

```
def a(x,y):return x+y+2
b=a
```

После чего `a` и `b` будут указывать на одну и ту же функцию, и она будет существовать до тех пор, пока не выполнить оператор `del` по отношению и к `a`, и к `b`. Такие правила существования справедливы для всех объектов, о чём мы узнаем уже через несколько лекций.

4.4 Особые приёмы работы с функциями

Таких приёмов насчитывается пять штук. Это именованные параметры, необязательные для указания параметры, параметры с неизвестной длиной и параметры с неизвестными именами. Ещё один приём, лямбда-исчисление, будет рассмотрен нами отдельно, так как он стоит этого.

- **Именованные параметры** — это механизм, позволяющий при вызове подпрограммы менять местами её параметры, зная их имена. Например, мы объявили функцию:

  ```
  def qualify(author,name,quality):
   print author+"'s", name, 'is a', quality, 'book.'
  ```

  Её можно вызывать так:

  `qualify('G.Booch', 'OOA&D with Applications', 'good')`

  А можно — так:

  `qualify(quality='good', name='OOA&D with Applications', author='G.Booch')`

  Результат будет одинаковым: `G.Booch's OOA&D with Applications is a very good book.`.

  Можно комбинировать этот подход с обычным перечислением параметров по порядку, но при этом именованные параметры должно стоять после всех обычных:

  `qualify(author='G.Booch', 'OOA&D', 'good')` — нельзя

  `qualify('G.Booch', quality='good',name='OOA&D')` — можно

- **Необязательные для указания параметры** — это механизм, позволяющий при вызове подпрограммы указывать значения только критических параметров, без которых она работать не будет, имея, тем не менее, возможность указания их при желании. Это реализуется путём указания значений по умолчанию для всех необязательных параметров при определении функции:

  `def qualify(author,name="book quality="bad"):`

  `print author+\s name, 'is a', quality, 'book.'`

  Теперь наша функция может принимать от одного до трёх параметров, причём, воспользовавшись предыдущим приёмом, можно задать качество книги без указания названия:

  `qualify('G.Booch',qualify='very good')`

  Рекомендуется при использовании необязательных для указания параметров присваивать им значения только неизменяемых типов (строк, чисел, кортежей), потому что используемое по умолчанию значение присваивается только один раз. Например, у вас есть функция:

```
def addel(n,x=[]):
 x.append(n)
 print x
```

Теперь попробуем запустить её несколько раз:

```
addel(2,[3,4]) # даёт [3,4,2] - правильно
addel(1) # даёт [1] - пока правильно
addel(2) # даёт [1,2] - неправильно!
addel(3) # даёт [1,2,3] - уж совсем неправильно!
```

Вместо этого следует пользоваться проверкой внутри тела функции, менее удобной, но работающей правильно.

- **Параметры неизвестной длины** — это механизм, позволяющий реализовывать подпрограммы, полностью инвариантные относительно количества предоставляемых им параметров. Записывается это так:

```
def qualifyAuthors(*several):
 for one in several:
  qualify(one)
```

При этом, как вы поняли, все параметры питон собирает в один кортеж и его отдаёт в качестве указанной переменной. Кортеж, безусловно, может быть пуст.

- **Непредусмотреные параметры** — это механизм, позволяющий реализовывать подпрограммы, стойкие к лишним параметрам и инвариантные относительно их имён. При этом ещё однам именем обозначается словарь, который либо пуст, если все параметры предусмотрены, либо состоит из пар название-значение. Синтаксис таков:

```
def qualify(author,name="book",quality="bad",**aux):
 print author+"'s", name, 'is a', quality,
 if aux:
  print 'book,',
  for a in aux.keys():
   print a,'works in',aux[a],',',
  print 'as one can read.'
 else:
  print 'book.'
```

Тогда нашей весьма выросшей процедурой можно пользоваться вот так:

```
qualify('G.Booch','OOA&D','very good')
qualify('A.V.Aho,R.Sethi,J.D.Ullman', quality='perfect',
name='Dragon Book', Aho='AT&T Bell Labs',Sethi='AT&T Bell
Labs',Ullman='Stanford University')
```

что выдаст:

```
G.Booch's OOA&D is a very good book.
```

```
A.V.Aho,R.Sethi,J.D.Ullman's Dragon Book is a perfect book,
Aho works in AT&T Bell Labs, Sethi works in AT&T Bell Labs,
Ullman works in Stanford University, as one can read.
```

VIII
Лекция восьмая

4.5 Лямбда-исчисление

В первой половине XX века американский математик Алонзо Чёрч предложил использовать для описания частично рекурсивных функций достаточно простой формализм, названный им лямбда-исчислением. Он же сформулировал так называемый **тезис Чёрча** (на котором базируются тезисы Тьюринга и Маркова) о том, что любая функция, вычислимая в интуитивном смысле эквивалентна некоей частично рекурсивной функции. Этот тезис содержит в себе нестрогое определение, поэтому, с одной стороны, не может быть доказан, а с другой, позволяет упростить некоторые теоретические выкладки. Частично рекурсивные функции суть функции, которые могут зависеть от собственного значения при других входных данных и могут быть определены не для всех входных данных.

Впервые лямбда-выражения появились в языке Лисп в конце 1950-х годов. Позаимствовав термин у Чёрча, его создатели, безусловно, внесли множество изменений. Позже было создано несколько языков чисто функционального типа, как на базе Лиспа (Схема, Лупс, КЛОС, Миранда, Хаскелл), так и сильно отличающихся от него (ФП, МЛ, Хоуп, Эрланг). Функциональное программирование — это отдельная парадигма программирования, где программист задаёт зависимость функций друг от друга, определяя таким образом их свойства и значения. В языках, наиболее точно соответствующих этой концепции, нет переменных, которые могут влиять на контекстную независимость, как мы видели на прошлой лекции. Элементы функционального программирования есть и в питоне.

Лямбда-функция в питоне — это функция без имени, о которой известно только количество аргументов и формула для вычисления итогового значения, причём формула должна записываться единым выражением. Вот пример лямбда-функции, складывающей три числа:

```
lambda x,y,z:x+y+z
```

Проще и не придумать. Понятно, что описывать огромную функцию, вызывающуюся много раз, лямбда-функцией, по меньше мере неразумно, но в некоторых случаях (которые мы рассмотрим на этой лекции) лямбда-функции бывают полезны. Во-первых, их можно присваивать:

```
R=lambda x,y:pow(x*x+y*y,0.5)
```

```
print R(3,4)
```

На печать будет выдано `5.0`. Сравните ту же самую функцию, объявленную стандартным питоновским способом:

```
def R(x,y):
 return pow(x*x+y*y,0.5)
```

Длиннее, многословнее и, что более важно, менее очевидно. Когда же мы перейдём к применению лямбда-функций в приёмах функционального программирования, экономия места будет ещё больше.

Второе применение лямбда-функций следует из того, что определение обычной функции не может быть сгенерировано программой «на лету», а определение лямбда-функции — может, и очень просто:

```
def genincr(n):
 return lambda x,i=n:x+i
```

Эта функция возвратит функцию-инкрементатор, увеличивающую свой аргумент на `n`, где `n` даётся при создании функции. Для любителей экзотики сразу отвечаем утвердительно на возникший у них вопрос. Да, так тоже можно:

```
genincr=lambda n:lambda x,i=n:x+i
```

Это определение функции `genincr` полностью эквивалентно предыдущему.

Упражнение. Почему нельзя было задать возвращаемую функцию-инкрементатор как `lambda x:x+n`? (Возможно, если вы затрудняетесь ответить на этот вопрос, вам следует повторить материал прошлой лекции).

4.6 Элементы функционального программирования

Кроме всех этих очевидных применений лямбда-функций, существуют ещё четыре стандартных приёма:

1. **Вызов функций — apply()**.

   Выполнять функции можно, как мы знаем, пользуясь скобками: `func()`, где `func` — имя функции. Но в некоторых случаях бывает удобно сначала последовательно подготовить все аргументы, и только потом вызвать функцию. Функция `apply` принимает два или три аргумента (третий необязателен). Первый — функция: либо имя переменной, содержащей функцию, либо определение лямбда-функции. Второй — кортеж (или другая последовательность, которая внутри функции `apply` всё равно преобразуется в кортеж) с параметрами. Третий аргумент — словарь со всеми именованными параметрами. Так,

   ```
   A=1,2,[3,4],[5,6],7
   ```

   ```
   B=2,3
   ```

   ```
   apply(lambda x,y,z:x[y].append(z),(A,2,B))
   ```

   аналогично следующему:

```
A=1,2,[3,4],[5,6],7
B=2,3
def func(x,y,z):
 x[y].append(z)
func(A,2,B)
```

2. **Отображение списков — map().**

   Функция `map()` порождает новый список из значений функции, применённой к каждому элементу первоначального списка. Легко догадаться, что эта функция берёт не менее двух аргументов: функции для применения и последовательности (списка или кортежа) её параметров. В этом случае наша функция должна брать только один параметр. Можно использовать и многопараметрические функции, но в этом случае нужно давать столько списков или кортежей, сколько у неё параметров. Конечно же, они должны быть одинаковой длины. Вне зависимости от типов последовательностей, данных функции `map`, она вернёт список.

   Функция `map` является как бы обобщением предыдущей функции, `apply`, последовательно применяя данную функцию к элементам последовательности. Можно, например, вычислить значения синусов чисел от 1 до 10:

   ```
   from math import sin
   map(sin,range(1,10))
   ```

   Если же нужен не синус, а, скажем, возведение в третью степень, нам поможет лямбда-функция:

   ```
   map(lambda x:x*x*x,range(1,10))
   ```

   В употреблении функции `map` существует ещё одна хитрость: можно вместо функции подставить `None`, тогда будет использована функция по умолчанию — т.н. функция идентичности, возвращающая свои аргументы. Догадливые программисты могут использовать этот факт для краткой записи транспонирования матрицы. С использованием всех полученных нами знаний можно определить функцию транспонирования матрицы произвольного размера следующим образом:

   ```
   trans=lambda X:map(list,apply(map,[None]+X))
   ```

   Матрица должна быть представлена в виде списка списков. Это удобное представление не только для работы с элементами, но и для функции `apply`. Не хватает только первого (точнее, нулевого) элемента — имени функции, что мы и добавляем. Затем выполняется `apply`, а точнее, `map`, собственно транспонирующий наш список списков в список кортежей, что исправляется (нехорошо изменять структуру представления при транспонировании) применением функции `list` к каждому элементу списка (к каждому кортежу). Запусками нашей функции с

различными параметрами можно убедиться, что она работает как для квадратных, так и для прямоугольных матриц.

3. **Фильтрация списков — filter()**.

   Функция `filter()` генерирует новый список из тех элементов исходного списка, для которых проверочная функция истинна. Сами значения элементов при этом не изменяются. Первым аргументом даётся проверочная функция, а вторым следует список (или кортеж, или строка, ...). Если функция — `None`, то аналогично уже описанному используется функция идентичности, то есть из последовательности выбрасываются все нулевые или пустые элементы.

   Например, список нечётных чисел от 2 до 15 можно получить так:

   ```
   filter(lambda x:x%2,range(2,16)
   ```

   Как видно, лямбда-функции хорошо себя рекомендуют и тут. И только врождённая честность не позволяет нам утаить тот факт, что список нечётных чисел можно получить и проще, пользуясь третьим, необязательным параметром функции `range()`.

4. **Цепочечные вычисления — reduce()**.

   Функция `reduce()` производит цепочечные вычисления, многократно применяя данную функцию к каждому элементу, подставляя аккумулятор в качестве первого параметра, а сам элемент — в качестве второго. При этом она берёт от двух до трёх аргументов: функцию для вычисления и последовательность как обязательные и стартовое значение аккумулятора как необязательный. Например, факториал числа можно считать так:

   ```
   fact=lambda n:reduce(lambda a,N:a*N,range(1,n+1),1L)
   ```

   Стартовое значение в `1L` необходимо для того, чтобы результат был длинным целым, иначе нельзя будет посчитать даже факториал 10.

   Цепочечные вычисления идут слева направо. Например, при выполнении `lambda x,y:x+y,range(1,5)` порядок выполнения будет таков: `(((1+2)+3)+4)`.

4.7 Поиск простых чисел

Пришла пора нам попробовать применить полученные знания о функциональном программировании. Итак, не боясь длинных выражений, будем помнить все приёмы. Попробуем написать лямбда-функцию, которая будет находить все простые числа, не превосходящие данного. Что такое простое число? Согласно определению, простое число не делится без остатка ни на какое другое число. Понятно, что нет смысла проверять делимость на числа, превышающие исходное. Напишем сначала функцию, составляющую список всех остатков от деления данного числа на другие:

```
Z=lambda n:map(lambda a,b=n:b%a,range(2,n))
```

Если в этом списке есть хотя бы один ноль, это означает, что число `n` делится без остатка на какое-то другое число, то есть, что `n` — не простое. Построим функцию, возвращающую 1, если в данном ей списке нет нулей и 0 в противном случае:

```
Y=lambda l:reduce(lambda c,d:c*d!=0,l,1)
```

Теперь `Y(Z(n))` возвращает 1, если `n` — простое и 0 в противном случае. Половина дела уже сделана. Осталось реализовать перебор всех чисел из какого-то промежутка, то есть построить отображение (map) списка последовательных чисел в список нулей и единиц. В таком случае единицы будут стоять на местах, обозначающих номер простого числа. Будет лучше, если вместо единиц мы будем выдавать само число, тогда, удалив все нули и списка, мы получим список простых чисел без лишних усилий.

```
X=lambda m:map(lambda e:e*Y(Z(e)),range(2,m))
```

Ну, удалить нули для нас проще простого:

```
W=lambda k:filter(None,X(k))
```

Если вспомнить арифметику, то станет понятно, что делитель числа не может превышать его корня. Поэтому из соображений оптимизации по скорости `Z` можем изменить следующим образом:

```
Z=lambda n:map(lambda a,b=n:b%a,range(2,1+pow(n,0.5)))
```

Всё, задание выполнено и, даже более того, выполнено оптимально. Если мы теперь подставим `Z` в `Y`, `Y` в `X`, а `X` в `W`, то сами ужаснёмся результату наших усилий:

```
V=lambda k:filter(None,map(lambda e:e*reduce(lambda
c,d:c*d!=0,map(lambda a,b=e:b%a,range(2,1+pow(e,0.5))),1),range(2,k)))
```

Для тех, кто любит создавать программы, которые невозможно прочесть кому-либо, кроме их создателя (да и ему это под силу только спустя день-два после написания), можно напомнить, что разные переменные из разных областей действия имён могут иметь одинаковые имена. Если это учесть (а у нас нигде не употребляется более двух имён параметров одновременно), то функция примет вид:

```
U=lambda x:filter(None,map(lambda x:x*reduce(lambda
x,y:x*y!=0,map(lambda y,x=x:x%y,range(2,1+pow(e,0.5))),1),range(2,x)))
```

Потрясающе! Пользуйтесь приёмами функционального программирования, и ваши программы будут мутны и нечитаемы! А вообще, рассмотренный нами пример есть курьёз, хоть и вполне жизнеспособный, как правило, элементы функционального программирования у программистов на питоне так далеко не идут. Но игнорировать этот аппарат нельзя — слишком велика выгода от его применения, как в визуальном представлении алгоритмов, так и в скорости выполнения программ.

4.8 Подпрограммы как средство поднятия уровня абстракции

Рассмотрев различные виды функций и их применения, мы можем подвести итог. Подпрограммы являются чрезвычайно полезным инструментом,

без которых невозможна реализация сколь-нибудь крупных проектов. Подпрограммы позволяют скрывать от проектировщика подробности реализации тех или иных мелких подзадач и дают сконцентрироваться на их композиции и решении больших задач путём собирания их из готовых решений подзадач. Подпрограммы дают возможность смены терминологии, её укрупнения. Например, при реализации межпрограммного взаимодействия по сети одни подпрограммы будут решать задачу пересылки серии байт в порт компьютера, другие будут пересылать целые массивы сложных строковых данных с контролем целостности, пользуясь при этом первыми, третьи будут управлять работой удалённого компьютера и приложений на нём, посылая команды путём запуска других функций, четвёртые подпрограммы будут позволять запросто работать на одном компьютере, при этом оперируя окном приложения с другого, пользуясь третьими подпрограммами. Всё это пришлось бы делать одновременно, если бы не было этого аппарата.

На практике дело обстоит куда лучше, для многих элементарных задач существуют уже написанные решения, поэтому программисту-разработчику достаточно только приспособить их для своих нужд, то есть использовать их в своих подпрограммах. Это называется *повторным использованием кода* и применяется очень широко. И этого не было бы без аппарата создания подпрограмм.

Подпрограммы позволяют создавать некий алгоритм решения проблемы и называть его одним именем, пользуясь этим именем позже, при составлении более сложных алгоритмов, и так далее. Сегодня мы заканчиваем первую часть курса, посвящённую процедурному программированию. Следующая лекция и все следующие за ней будут рассказывать уже об объектном подходе, модели, позволяющей писать ещё б*о*льшие проекты, и делать это проще, чем поддерживание спецификации сотен тысяч мелких подпрограмм.

IX
Лекция девятая

Сегодня мы наконец перейдём ко второй части курса. Если в первой части вы получили некоторые знания о языках программирования, о методах работы в питоне, о его стиле, об основных типах данных и управляющих конструкциях, то во второй вы ознакомитесь с современными принципами работы программиста и проектирования больших программных комплексов. Эти принципы, конечно, пригодны, и для малых программ, но в этом случае не настолько эффективны.

Когда компьютеры были большими, программы были маленькими, писались на перфокартах и на было необходимости в каких-то серьёзных научных методах их проектирования, потому что всё равно разобраться в

сколь-нибудь крупной программе (а такие всё же писались) без программиста или команды программистов, её создавших, было невозможно. Сейчас быстро развиваются как вычислительные мощности компьютеров, так и технология хранения информации и объёмы носителей. Это даёт возможность писать настолько большие программы, что в их создании задействованы тысячи людей и понадобилась технология, обеспечивающая, во-первых, взаимодействие этих людей, озабоченных каждый только решением своей маленькой задачи, а во-вторых, безболезненную замену одного программиста другим без переработки всего проекта.

Сначала для этого использовался структурный подход и теория спецификаций и абстракций. При этом большая задача решалась абстрактно, и её решение выражалось через решения ряда более простых задач. Это называется алгоритмическая декомпозиция, то есть разделение алгоритма на части. Если для решения какой-то мелкой подзадачи писалась процедура, к ней прилагалась так называемая спецификация, объясняющая её смысл и формат запуска так, чтобы обращение к ней могло происходить без знания внутренней структуры этой процедуры. Этот метод работал хорошо до тех пор, пока не были предприняты первые попытки создания систем, где число таких подпрограмм исчислялось тысячами. Никакие спецификации не помогут даже самому одарённому программисту запомнить тысячи подпрограмм. Кроме того, в этом случае нарушается второе требование — безболезненно заменить опытного программиста нельзя, потому что новый, пришедший на его место, не будет столь же компетентен, пока не запомнит первую пару сотен названий процедур.

На смену структурному проектированию пришла...

5 Объектно-ориентированная технология

5.1 Объектная модель и связанные с ней термины

Объектно-ориентированная технология основывается на так называемой объектной модели. Основными её принципами (мы рассмотрим их подробно) являются: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость. Каждый из этих принципов сам по себе не нов, но в объектной модели они впервые применены в совокупности.

Объектно-ориентированный анализ и проектирование принципиально отличаются от традиционных подходов структурного проектирования: здесь нужно по-другому представлять себе процесс декомпозиции, а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для структурного программирования. Отличия обусловлены тем, что структурное проектирование основано на структурном программировании, тогда как в основе объектно-ориентированного проектирования лежит методология объектно-ориентированного программирования. К сожалению, для разных

людей термин «объектно-ориентированное программирование» означает разное. Ренч, один из видных теоретиков, верно предсказал: *объектно-ориентированное программирование будет занимать такое же место, какое занимало структурное программирование. Оно всем будет нравиться. Каждая фирма будет рекламировать свой продукт как созданный по этой технологии. Все программисты будут писать в этом стиле, причём все по-разному. Все менеджеры будут рассуждать о нём. И никто не будет знать, что же это такое.*

Речь шла о восьмидесятых годах прошлого столетия, хотя предсказание продолжает сбываться и в наши дни. Мы попытаемся внести хоть небольшую ясность в осознание этого термина нашими слушателями (читателями).

Объектом называется некая сущность, которой можно оперировать. На самом деле такого определения уже достаточно, дальнейшая конкретизация может только погубить всё дело. В разных программах объектом может быть источник звука, устройство ввода/вывода, буква, книга, самолёт, планета, дыхание или полёт стрелы. Объект, как мы увидим позже, имеет состояние, поведение и идентичность. Состояние описывает текущее положение объекта, поведение — то, как он будет реагировать на различные ситуации, идентичность отличает объект от других.

Класс — это множество очень схожих объектов. Англичанин, наверное, объяснил бы различие между понятиями *класс* и *объект* через определённые и неопределённые артикли своего родного языка. Лишённые такой возможности, мы поясним на примерах: кошка — это класс, а трёхцветная кошка по имени Мурка, лежащая на подоконнике — объект. Карта PCI — класс, а внутренний модем Acorp/PCI/56k/V90 или звуковая карта SBLive! — разные объекты этого класса. Литературное произведение — класс, «Слово о полку Игореве» — объект. Вообще, что называть классом, а что — объектом, определяется программистом на этапе проектирования системы. Кроме того, класс можно понимать как `тип объекта`.

Объектно-ориентированное программирование, или object-oriented programming (OOP), определяется его следующим образом:

Объектно-ориентированное программирование — это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённом класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части:

- OOP использует в качестве базовых элементов объекты, а не алгоритмы
- Каждый объект является экземпляром какого-либо определённого класса
- Классы организованы иерархически

Конечно, программа будет объектно-ориентированной только при соблюдении всех трёх указанных требований. В частности, программирова-

ние, не основанное на иерархических отношениях, называется программированием на основе *абстрактных типов данных*.

Объектно-ориентированное проектирование, или object-oriented design (OOD) — термин хоть и близкий к OOP, но не идентичный ему. Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование же, напротив, основное внимание уделяет правильному и эффективному структурированию сложных систем. Мы определяем объектно-ориентированное проектирование следующим образом:

Объектно-ориентированное проектирование — это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приёмы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части: объектно-ориентированное проектирование

- основывается на объектно-ориентированной декомпозиции;
- использует многообразие приёмов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также её статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором — алгоритмами. Иногда мы будем использовать аббревиатуру OOD для обозначения метода объектно-ориентированного проектирования, потому что иначе при использовании русских аббревиатур мы быстро перепутаем программирование с проектированием.

Объектно-ориентированный анализ, или object-oriented analysis (OOA), направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ — это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

А как же соотносятся между собой OOP, OOD и OOA? На результатах, полученных OOA, формируются модели, на которых основывается OOD, которое, в свою очередь, создаёт фундамент для окончательной реализации системы с использованием методологии OOP.

5.2 Объекты и классы в питоне

В соответствии с определением OOP, не все языки программирования являются объектно-ориентированными. Страуструп, автор C++, одного из первых объекто-ориентированных языков, считал, что объектно-ориентированный язык — это язык, имеющий средства хорошей поддержки

объектно-ориентированного стиля программирования... Обеспечение такого стиля в свою очередь означает, что в языке удобно пользоваться этим стилем. Если написание программ в стиле ООР требует специальных усилий или невозможно, то этот язык не отвечает требованиям ООР. А отвечает ли требованиям объектно-ориентированного языка питон, выбранный нами? Это легко проверить. Попробуем создать какой-нибудь класс и его объект. Если мы этого сделать не сможем или нам будет очень трудно, придётся сложить оружие и признать, что питон не объектно-ориентирован.

Напишем класс `Dog`, описывающий собаку, которая может лаять. Для этого внутри описания класса (то есть после отступа) зададим новую функцию `hawl()`, от которой в нашем простом примере много не требуется. Затем создадим объект, то есть совершенно определённую собаку по имени Шарик. Попросим её полаять.

```
class Dog:
 def hawl(self):
  print "Гав!"
Sharik=Dog('Шарик')
Sharik.hawl()
```

Она действительно лает (можете проверить). Итак, всё было сделано быстро и заключилось в пяти строках кода. Вывод: питон — объектно-ориентированный язык!

Подведём итоги правил задания классов и объектов в питоне: класс задаётся ключевым словом `class` с указанием имени класса. При этом создаётся новая область действия имён переменных — можно создавать внутренние переменные класса и внутренние подпрограммы (они называются *методами*). Для создания объекта используется оператор присваивания, в правой части которого стоит имя класса. (Таким образом, каждый объект должен принадлежать какому-то классу). Обращение к методам объекта происходит через имя объекта и имя метода, разделённые точкой.

Есть только одно отличие метода от обычной процедуры или функции: первым параметром он обязан брать сам объект, частью которого является. При вызове метода с именем объекта оно подставляется автоматически, мы ведь написали `Sharik.hawl()`, а не `Sharik.hawl(Sharik)`. А вот при вызове метода с именем класса (и такое можно в питоне!) объект нужно передавать в явном виде: `Dog.hawl(Sharik)`. Таким образом, нельзя пользоваться методами класса, не создав ни одного экземпляра.

Следуя традиции, мы называем первый параметр `self` (сам). Это не более чем соглашение, и любой программист может в целях экономии места или по любой другой причине переименовать его.

В некоторых классах есть необходимость произведения некоторых действий при создании и/или при уничтожении объекта. Для этого используются *конструкторы* и *деструкторы*. Хорошим стилем объектно-ориентированного программирования считается записывание *всего*, что нужно выполнить при создании объекта, в конструктор, а всего, что нужно при уничтожении — в деструктор. Записываются они следующим образом:

```
class Dog:
 def __init__(self,newname='Бобик'):
  self.name=newname
 def __del__(self):
  print self.name,':',
  self.hawl()
 def hawl(self):
  print "Гав!"

Sharik=Dog('Шарик')
Bobik=Dog();
```

Конструктор — это та подпрограмма, которая вызывается при создании экземпляра класса со всеми параметрами, написанными между скобками после имени класса. Как видно из примера, никто не запрещает использовать в написании конструктора особых приёмов работы с подпрограммами, в частности, параметров со значениями по умолчанию. Вообще, конструктор — такой же метод класса, как и любой другой (в качестве любого другого можно рассмотреть `hawl`), только имеющий особое имя, распознаваемое интерпретатором как служебное. Подробнее о именах свойств и методов мы поговорим позже.

Деструктор вызывается неявно при удалении объекта (то есть, в том случае, когда мощность объекта станет нулевой), поэтому не может иметь никаких параметров (опять же, кроме самого объекта). Деструктор вызывается **до** действительного удаления, так что все данные ещё живы и могут быть спасены от уничтожения путём копирования в безопасное место.

X
Лекция десятая

6 Составные части объектного подхода

Объектная модель не появляется просто так. Существует большая теория, аргументирующая применения той или иной техники. Мы ограничимся беглым описанием основных составных частей объектно-ориентированного программирования и проектирования и их реализации в языке питон.

6.1 Абстрагирование

Абстрагирование является одним из основных подходов, используемых для решения сложных задач. Абстрагирование — это стратегический выбор точки зрения, это то, что опрееляет важные для нас свойства и качества моделируемого. Не имея возможность познать окружающий его мир полностью, человек выделяет из него то, что кажется ему важнее для постав-

ленной задачи. Они из крупнейших теоретиков и практиков программирования прошлого века, основоположник структурного программирования, голландский математик Эдсгер Дейкстра писал, что «абстрагирование проявляется в нахождении сходств между определёнными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих свойств, отвлекаясь на время от имеющихся различий».

Не отрекаясь от уважения к великому человеку, предложим своё определение: *Абстракция определяет концептуальные границы некоторого объекта, опуская одни его характеристики и выделяя другие, отличающие его ото всех прочих видов объектов.*

Гради Буч, один из главных теоретиков OOD, ввёл в дополнение к определению так называемый **принцип наименьшего удивления**, согласно которому абстракция должна охватывать всё поведение объекта, не больше и не меньше, не привнося сюрпризов и побочных эффектов.

Правильный выбор набора абстракций для предметной области — главная задача OOD. Существуют три основных типа абстракций:

- **Абстракция сущности** — объект олицетворяет собой модель некой сущности, принадлежащей предметной области.
- **Абстракция поведения** — объект состоит из обобщённого множества операций для работы с элементами предметной области.
- **Абстракция виртуальной машины** — объект группирует операции, находящиеся на одном уровне абстракции.

В качестве примера абстракции сущности можно привести первый класс, созданный нами в питоне — «собаку». Если взять какое-либо приложение, вынужденное обмениваться сообщениями по сети, для которого, тем не менее, этот процесс не является основным (не лежит в предметной области), то объект «соединение» будет абстракцией поведения, потому что, не отвечая никакой сущности, он содержит методы «послать», «получить» и т.д.

Объект, реализующий всевозможные функции работы с определённым форматом файлов, представляет собой типичнейший пример абстракции виртуальной машины. Его реализация использует объект «файл», лежащий на более низком уровне абстракции, его интерфейс может использоваться объектом более высокого уровня, собственно работающим с этим форматом.

Соглашение о взаимодействии абстракций двух объектов называют контрактом. Контрактную модель программирования впервые рассмотрел Бертран Мейер — человек, сделавший для OOD не меньше Буча. Эта модель базируется на инвариантах (логических условиях, значение которых должно сохраняться). Для каждой операции объекта задаются предусловия (инварианты, предполагаемые операцией) и постусловия (инварианты, которым удовлетворяет операция). Изменение инварианта нарушает контракт, связанный с абстракцией. В нарушении предусловия нужно

винить вызывающую сторону, постусловия — выполняющую. В современных языках программирования для надзора за контрактами применяются средства работы с исключительными ситуациями. Есть они и в питоне.

Исключительные ситуации (или исключения) — это ошибки, возникающие во время выполнения программы. Когда программы исполнялись неинтерактивно (без взаимодействия с пользователем), соответствующая реакция на исключительную ситуацию заключалась в том, чтобы напечатать сообщение об ошибке и завершить выполнение программы. Однако реакция на исключение в интерактивной среде не модет быть ограничена сообщением, а должна также включать восстановление. Это может быть продолжение вычислений, возврат к последнему узлу, в котором пользователь может выбрать другой вариант, либо выход из внутренней процедуры с последующим её блокированием. В системах, которые должны работать стабильно по своей природе (например, в программном комплексе системы управления запуском ракет), восстановление должно выполняться без участия человека!

Восстановление при ошибках не даётся даром — какие-то накладные расходы неизбежны. Но нужно следить за тем, чтобы:

- В случае отсутствия исключения издержки должны быть очень небольшими,
- Обработка исключения должна быть простой, быстрой и свободной от ошибок.

Действительно, мы рассчитываем на то, что исключительные ситуации, как правило, **не возникают**, поэтому они не должны слишком замедлять выполнение программы. И, конечно, программирование реакции на исключения не должно быть слишком утомительно по той же причине.

Упражнение. Является ли алгоритм обработки исключительных ситуаций эквивалентным условному оператору? Почему? Какие у них общие черты? Какие различия?

Теперь поговорим об этом механизме в питоне. Исключительная ситуация (exception) *возникает* (raised) в том месте, где происходит ошибка. Она может быть *перехвачена* (handled) специальным аппаратом, о котором и идёт речь. Исключительная ситуация может возникнуть не только по вине, но и по желанию программиста:

```
raise <имя>
```

При этом имя должно означать имя стандартной исключительной ситуации, строку с именем новой или же экземпляр какого-либо класса. Исключительные ситуации сами по себе являются объектами соответствующих классов. Так, конкретная ошибка деления на ноль — это объект класса `ZeroDivisionError`. Если в качестве имени дать что-либо другое (например, число), исключительная ситуация возникнет, но другая — `TypeError`. Вторым, необязательным, параметром оператора raise может идти параметр ошибки.

Оператор попытки выполнения (а именно он занимается перехватом исключительных ситуаций) бывает двух видов. Вот так записывается первый:

```
try:
 <операторы>
except <Имя>:
 <операторы>
else:
 <операторы>
```

Этот вид оператора мы будем называть try/except. Операторы, помещаемые между try и первым except, называются *испытательными инструкциями* (термин в англоязычной литераторе — *try clause*). Они выполняются в любом случае, и, если ничего не происходит (не возникает ошибки), выполнение программы продолжается с оператора, следующего за всей конструкцией. Дальше идут обработчики ошибок, начинающиеся с ключевого слова `except` и названия ошибки, после чего идёт собственно код обработчика. Таких обработчиков может быть несколько. Необязательная else-ветвь оператора try/except выполняется в том случае, если ни одна ошибка не произошла. Обычно в этом месте располагается код, являющийся продолжением кода из испытательной части, но ошибки в котором не должны обрабатываться (или должны обрабатываться по другому — не стоит забывать о том, что конструкции try могут быть вложенными).

Второй вид, называемый нами try/finally, имеет следующий вид:

```
try:
 <операторы>
finally:
 <операторы>
```

Финализирующие инструкции выполняются *в любом случае*: была ошибка или нет. В том случае, если ошибка произошла, её обработка задерживается. Сначала выполняются все операции, стоящие после `finally`, и только затем исключение возникает ещё раз. Эту конструкцию используют в основном для того, чтобы успеть что-то сделать до выхода из программы (выдать сообщение, закрыть файл, etc).

В питоне существуют двадцать семь стандартных исключения. С помощью определённых объектно-ориентированных приёмов можно написать любое количество своих исключений. Основные стандартные исключения таковы:

- **ArithmeticError** — арифметические ошибки (кроме деления на ноль),
- **AttributeError** — ошибка доступа к несуществующему свойству или методу,
- **EOFError** — ошибка конца файла,
- **EnvironmentError** — ошибка интерактивной среды,

- **FloatingPointError** — ошибка при вычислениях с плавающей точкой,
- **ImportError** — ошибка подключения внешних модулей,
- **IndentationError** — ошибка отступов (отступ без составного оператора, отсутствие отступа, etc),
- **IndexError** — попытка чтения несуществующего элемента последовательности,
- **KeyError** — попытка чтения несуществующего элемента словаря,
- **MemoryError** — ошибка выделения памяти,
- **NameError** — неверное имя переменной,
- **NotImplementedError** — возникает при вызове абстрактного метода объекта, то есть метода с описанием, но без тела,
- **OSError** — внешняя ошибка питона, предположительно относящаяся к операционной системе (у этого класса наследует класс **WindowsError** — класс ошибок, относящихся к операционной системе Microsoft Windows),
- **OverflowError** — ошибка переполнения,
- **RuntimeError** — ошибка времени выполнения (базовый класс, наследниками которого являются многие другие),
- **SyntaxError** — ошибка синтаксиса,
- **SystemError** — внутренняя системная ошибка питона,
- **TypeError** — ошибка приведения типов данных (например, сложение числа со строкой),
- **UnboundLocalError** — при попытке сослаться на локальную переменную, с которой не связано ни одно значение,
- **UnicodeError** — ошибка при кодировании или раскодировании уникодовых строк,
- **ValueError** — ошибка значения, очень часто возникает, имеет много наследников,
- **ZeroDivisionError** — ошибка деления на ноль.

Специально заучивать названия ошибок, конечно, не нужно. Те, что будут у вас возникать достаточно часто, запомнятся и без лишних усилий, а об остальных всегда может поведать стандартная документация.

XI
Лекция одиннадцатая

6.2 Инкапсуляция

Инкапсуляция позволяет скрывать детали реализации, на являющиеся важными для использования объекта. Как писал Ингаллс, автор объектно-ориентированного языка программирования Смоллток: «Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части».

Абстракция и инкапсуляция взаимодополняют друг друга: абстрагирование направлено на придание смысла внешним свойствам объекта, а инкапсуляция устраняет влияние внутренней реализации. Инкапсуляция делает границы между абстракциями более чёткими, не позволяя абстракциям высокого уровня вмешиваться в работу абстракция более низкого уровня (а обратное влияние невозможно в принципе).

Практическое следствие объединения инкапсуляции с абстракцией означает всего лишь вот что: любой класс можно мысленно разделить на две части: интерфейс и реализацию. *Интерфейс* описывает абстракцию поведения объектов данного класса, он отражает внешнее поведение. *Реализация* же воплощает это поведение, описывая алгоритмы достижения желаемого.

Гради Буч считает, что такой принцип разделения «соответствует сути вещей», потому что интерфейс собирает всё, связанное с взаимодействием данного объекта с любыми другими, а реализация успешно скрывает все детали.

Итак, дадим такое определение: *Инкапсуляция — это процесс отделения интерфейса объекта, отражающего его внешнее поведение, от реализации, описывающей собственно алгоритмы достижения желаемого поведения объекта. Инкапсуляция изолирует контрактные обязательства абстракции от их реализации.*

Соблюдение инкапсуляции в питоне почты полностью является ответственностью программиста. Однако некоторые средства, помогающие ему, присутствуют. Согласно сложившейся традиции, свойства или методы (то есть переменные области имён объекта) делятся на свободные, личные, скрытые и служебные. Сорт определяется именем.

- Свободные свойства и методы — это основная их масса. Они не имеют никаких ограничений на имена. Доступ к ним может получить кто угодно, хотя прямой доступ к свойствам другого объекта не считается правильным стилем программирования. Для прямого низкоуровневого общения со свойством обычно пишутся две функции: возвращающая (accessor или getter) и изменяющая (mutator или setter). Вот как это может выглядеть:

```
class A:
 x=1
```

```
def getX(self):return x
def setX(self,n):x=n
```

- Личные свойства и методы содержат в начале имени один символ подчёркивания. Доступ к ним интерпретатором не запрещается, но подчёркивание подразумевает некоторую степень служебности. Поэтому такие имена обычно даются переменным, к которым вообще не предполагается никакого доступа извне, то есть переменным, являющимся частью внутренней структуры (инкапсуляция реализации абстракции). Например, как в данном коде:

  ```
  class B:
  ␣␣d=''
  ␣def␣add(self,s):self._d+='␣'+s
  ␣def␣get(self):return␣self._d.split('␣')\leftbr1:\rightbr
  ```

- Скрытые свойства и методы содержат не меньше двух подчёркиваний в начале имени и не больше одного — в конце. Их инкапсуляцией занимается интерпретатор питона, но делает это достаточно своеобразно. Получить доступ к скрытой переменной можно, но для этого нужно знать имя класса, экземпляром которого является объект-носитель.

  ```
  class␣C:
  ␣def␣__init__(self):self.__x=0
  ␣def␣add(self,n):self.__x+=n
  ␣def␣sub(self,n):self.__x-=n
  ␣def␣write(self):print␣self.__x
  c=C()
  ```

  При попытке обратиться к переменной `c.__x` мы получим ошибку AttributeError. Такой переменной нет, есть переменная `ci._C__x` — реальное имя создаётся из символа подчёркивания, имени класса и исходного имени. При этом подпрограммы объекта могут использовать краткую форму имени без указания имени класса (потому как задаются непосредственно внутри него). Это не относится к приобретённым методам (то есть методам, присвоенным классу или объекту уже после завершения его создания):

  ```
  def␣ext(a,b):␣a.__x=b
  class␣C:
  ␣def␣__init__(self):self.__x=0
  ␣def␣add(self,n):self.__x+=n
  ␣def␣sub(self,n):self.__x-=n
  ␣def␣write(self):print␣self.__x
  ```

```
c=C()
C.set=ext
```

В данном случае вызов, например, `c.set(10)` не приведёт к изменению переменной `c._C__x`, потому что будет изменять `c.__x` — ту переменную, непреобразованное имя которой содержит исходный текст функции. Будьте осторожны при работе со скрытыми свойствами, а не то они могут оказаться скрытыми даже от вас самих. Только подпрограммы класса, заданные при его описании, имеют право использовать краткую форму записи имени скрытого свойства.

- Служебные свойства и методы начинаются и завершаются двумя символами подчёркивания, как, например, уже используемые нами конструктор `__init__` и деструктор `__del__`. Прочие служебные свойства включают:
  - **Представление объекта строкой:** `__str__` и `__repr__` будут использоваться при запуске стандартных питоновских функций `str` и `repr` соответственно, предлагая две возможности краткого представления объекта строкой.
  - **Сравнение с другими объектами:** `__lt__` (меньше), `__le__` (меньше либо равно), `__eq__` (равно), `__ne__` (не равно), `__gt__` (больше) и `__ge__` (больше либо равно) задают правила сравнения данного объекта с другими объектами (на обязательно того же класса!). Все эти функции должны возвращать булево значение (во всяком случае, то, что они будут возвращать, будет *трактоваться* именно как булев тип). Вместо них можно задать одну обобщённую `__cmp__`, возвращающую отрицательное число, если другой объект превосходит наш, положительное, если базовый объект превосходит чужой, и ноль в случае их равенства. Таким образом, если программист запишет `x<y`, на самом деле будет выполнено `x.__lt__(y)`.
  - **Размер объекта и проверка на ложность:** как мы уже знаем, ложными считаются нули, пустые строки, последовательности и объект `None`. Для определения проверки на пустоту объекта используется функция `__nonzero__`, возвращающая значение булева типа. Если такая функция не определена, вместо неё будет использоваться функция `__len__`, возвращающая длину объекта в каких-то единицах измерения. Внимание, если ни одна из этих функций не определена, все объекты данного класса будут считаться истинными! Конечно, `__len__` имеет и прямое применение (через стандартную функцию `len`).
  - **Использование объекта как функции:** в питоне можно обращаться с некоторыми объектами как с функциями, при этом вызов такого вида: `x(arg1,arg2,...)` будет заменён на вызов `x.__call__(arg1,arg2,...)`

– **Использование объекта как последовательности:** аналогичным образом существуют служебные методы, вызывающиеся при попытке взять элемент объекта как массива: x[n]. Они включают: `__getitem__` для взятия элемента (приведённый пример будет на самом деле выполнен как `x.__getitem__(n)`), `__setitem__` для изменения значения элемента присваиванием и `__delitem__` для тех случаев, когда описываемая объектом последовательность поддерживает удаление элементов. Также вводятся функции `__getslice__`, `__setslice__` и `__delslice__` для сечений. Важным атрибутом последовательности может стать служебный метод `__contains__`, возвращающие истинное значение, если его аргумент принадлежит последовательности и ложное в противном случае.

Вообще, следует помнить слова Бьёрна Страуструпа: «инкапсуляция защищает от ошибок, но не от жульничества».

XII
Лекция двенадцатая

6.3 Модульность

Во многих современных объектно-ориентированных языках программирования (и питон — не исключение) классы не являются единственным возможным механизмом декомпозации. Классы и объекты составляют логическую структуру системы, они классифицируются и размещаются по модулям, которые образуют физическую структуру системы. Каждый модуль представляет собой почти замкнутую подсистемы: классы внутри модуля могут иметь тесную взаимосвязь, тогда как внешние связи слабы или же вовсе отсутствуют.

Модульность — это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собою модули.

Модульность позволяет программисту группировать абстракции и хранить их сообразно тому, как они связаны между собой. Правильное разбиение программы на модули — задача, не уступающая по сложности таким проблемам OOD, как выбор набора абстракций или отделение реализации объекта от интерфейса.

На макроуровне модулей (*макро* относительно уровня классов) также используется деление на интерфейс и реализацию. Несмотря на то, что в одних языках программирования это деление более явно и вписано в синтаксис, а в других почти нивелируется, такое разделение является одной из немногих испытанных технологий проектирования, поэтому широко используется программистами и документаторами. Интерфейс модуля иногда называют спецификацией, придавая ему таким образом чисто декларатив-

ный смысл. Это даёт моральное право называть реализацию телом модуля.

Очевидно, что задача группировки классов по модулям имеет два крайних решения:

1. Помещение всех классов в один модуль.

   Метод пригоден только для небольших или малых задач, в сколь-нибудь серьёзных проектах он может вызывать осложнения как в документировании, так и в поддержании системы с течением времени. Зачастую же этот метод используют при начальном проектировании части большой системы как отдельного приложения.

2. Помещение каждого класса в свой модуль.

   Вполне подходящий метод для систмы из нескольких очень больших классов, чрезвычайно мало связанных между собой и большей частью используемых по раздельности. Например, стандартный питоновский модуль `cmd` (используемый для написания интерпретаторов команд) содержит только класс `Cmd`, выполняющий все необходимые операции. Но размещение в нескольких тысячах файлов мелких одиночных объектов или вообще констант (которые в чисто объектно-ориентированных языках также являются объектами) — это дурная привычка, не приводящая ни к чему, кроме долгой перетрансляции всей системы при малейших изменениях и сизифовому труду документаторов.

Согласно Гради Бучу, «деление программы на модули бессистемным образом гораздо хуже, чем отсутствие модульности вообще». И во многих случаях это действительно так. Число всевозможных распределений N классов по модулям равно 2^N, а сколько из них разумных? На этот вопрос даже дискретная математика ответа не даёт.

XIII
Лекция тринадцатая

текст утерян

XIV
Лекция четырнадцатая

6.5 Иерархия

Последняя главная составляющая объектного подхода — это, конечно же, иерархия. Предыдущая тема могла немного сбить нас с основного курса, поэтому сделаем предварительные выводы. Ни один программист, даж самый

гениальный, не в состоянии разобраться в большом проекте, если будет его рассматривать целиком во всех подробностях. Для декомпозиции большого проекта используется абстракция, которая выделяет определённую часть системы, унифицируя её с реальным объектом. Помощь инкапсуляции также может быть легко облечена в слова: инкапсуляция прячет незначительные, запутывающие, отвлекающие детали абстракции. Таким образом, мы сначала смотрим на систему в целом, затем определяем границы её составных частей, затем даём каждой части имя и подменяем рассмотрение части лицезрением имени (и его внешних свойств, разумеется). Модульность позволяет зайти с другого края, выделив в системе подсистемы.

Существуют системы настолько крупные, что число абстракций в них превышает пределы человеческих перцептивных возможностей. При этом они могут быть тесно связаны, лишая программиста возможности применения модульной декомпозиции. Рассмотренные нами подходы не дадут ответа на вопрос: «Что же делать в таком случае?» Ответ — в следующем определении.

Иерархия — это упорядочение абстракций, расположение их по уровням.

Этот метод, позволяющий рассматривать разнородные абстракции по раздельности, есть великое изобретение голландского математика Эдсгера Дейкстры и просто находка для объектного подхода. Иерархия не только позволяет рассматривать, исследовать различные совокупности абстракций, но и разделить собственно программирование на логически завершённые этапы, на каждом из которых реализуются абстракции определённого уровня.

Поясним идею подразделения абстракций на подчинённые друг другу группы на примере технологии написания современных крупных 3D-проектов: мультфильмов, спецэффектов и т.п. Будем считать сценарий уже написанным и все технические проблемы решёнными.

Сначала группа художников просто рисует, как в том или ином ракурсе выглядит описываемый предмет. Затем группа модельщиков на основе этих входных данных создаёт статичные трёхмерные объекты, используя примитивные методы, уже содержащиеся в программе для моделирования (мэппинг, рендеринг, текстурирование, шейдинг, фильтрация, антиалиасинг, эффекты коррекции света и др.). Затем описывают более сложные функции: перемещение камеры вокруг предмета, изменение освещённости, простейшие движения самого предмета, его частей. Следующим этапом стоит написание крупных процедур, реализующих движение предмета: как он делает шаг, как хватает другой предмет, как говорит, как падает... Одновременно начинают работу озвучиватели, стремящиеся добиться максимальной отдачи и реального трёхмерного звука с помощью реверберации, акклюзий, обструкций, морфинга, позиционирования и прочих готовых продуктов. Следующий этап ещё сложнее, чаще всего он выполняется с привлечением не только художников, но и актёров. Он включает описание таких процессов, как ходьба, произнесение длинной речи с жестикуляцией, взаимодействие предметов друг с другом и с окружающей средой и т.п.

Финальным этапом является готовый фильм (или сцена).

Таким образом, мы своим непрофессиональным взглядом насчитали 6 этапов, каждый следующий из которых построен из предыдущего. Процедуры, реализующиеся на различных этапах, принадлежат разным уровням абстракции. Мы не обязаны думать о поведении каждого пикселя, делая шаг и о каждом шаге, моделируя бег большой толпы.

Вернувшись к теории объектно-ориентированного подхода (кстати, повысив при этом уровень абстракции), мы можем дать ответ на мучивший нас не так давно вопрос. Что делать, если система слишком сложна для понимания, несмотря на то, что все подходы были применены верно? Полный ответ таков: добавить уровень абстракции и рассматривать каждый уровень отдельно от других. Сложные действия куда проще описывать через более мелкие действия, чем писать их заново на каждом этапе.

Совершенство иерархии заключается в том, что уровни абстракции можно добавлять потенциально бесконечно. На практике обычно довольствуются их конечным числом.

Основными типами иерархических структур в сложных системах являются иерархия «является», иерархия «используется» и иерархия «часть».

Содержание