ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

(Python Beginning)

Цели и задачи курса

Сформировать у студентов знания и умения в области объектно-ориентированного программирования на языке *Python*, позволяющие им решать практические задачи, связанные с разработкой программ с графическим интерфейсом пользователя, предназначенных для обработки данных и построения графиков.

Рабочий инструментарий

Рабочая среда – *Python IDLE*, *JetBrains PyCharm*.

Библиотеки – стандартная библиотека языка *Python*, плюс – *matplotlib*, *numpy*, *scipy*.

Входной порог

Студент должен владеть основами процедурного и объектно-ориентированного подходов к программированию. Для студентов, обучающихся по специальности 09.03.01 «Информатика и вычислительная техника», это знание языков программирования C/C++ в рамках учебных курсов «Программирование», «Структуры и алгоритмы обработки данных» и «Объектно-ориентированное программирование».

Урок 1. (Lesson 1) Основы программирования

Введение

Python – это интерпретируемый язык программирования, который поддерживает механизмы трансляции кода (создания ***байт-кода***, ускоряющего процесс выполнения программ).

Для запуска *Python* – программы на компьютере пользователя должна быть установлена среда исполнения *Python* (как и в случае с *Java*).

Что представляет собой *Python* – программа и как запустить ее на исполнение на вашем компьютере?

Запуск из командного интерпретатора («Пуск» → «Выполнить» → «*cmd*», либо комбинация клавиш «*Win-R*» → «*cmd*»):

```
$python -c "<набор python-команд>"
$python -c "import os; os.system('CLS')"
$python -c "import os; print(os.getpid()); os.system('PAUSE')"
$python -c "for i in range(10): print(i)"
```

```
$python myProrg.py
```

...отдельная песня для Linux–систем, так как он требует от администратора больших затрат на настройку политики безопасности системы…

То есть, если у пользователя есть право запускать *Python*, то помимо *Python* он сможет получить доступ и к другим программным инструментам…

Проверка установки Python

Запустите любой текстовый редактор и создайте в нем документ с расширением «*.py*», содержащий следующий текст.

```
# filename: test001.py
# проверка установленной версии Python
import sys # подключение модуля, аналог директивы include
print(tuple(sys.version_info))
try:
    raw_input() # Python 2.x
except NameError:
    input()     # Python 3.x
```

Эта программа может быть запущена как из командной строки:

```
Командная строка> python test001.py
```

так и из среды *Python IDLE* (открыть/создать и запустить)

В программе, поддерживающей кириллицу, необходимо указать используемую/текущую кодовую страницу (*code page*):

```
# -*- coding: <кодировка> -*-
```

Для *MS Windows*

```
# -*- coding: cp1251 -*-
```

Для *UNIX/Linux* – платформ

```
# -*- coding: utf-8 -*-
```

Самая первая программа на *Python*`е:

(***консольный вариант***)

```
# filename: test002.py
# -*- coding: cp1251 -*-
print("Hello World") # Выводим строку текста
input()              # Ожидаем нажатия клавиши <Enter>
```

(***вариант приложения с графическим интерфейсом***)

```
# filename: test003.pyw
# -*- coding: cp1251 -*-
from tkinter import *
root = Tk()
label = Label(root, text = "Hello World\nПривет, мир")
label.pack()
root.mainloop()
```

Комментарии в Python

Язык программирования Python поддерживает два вида комментариев:

1. «***обычные***», относящиеся к категории пояснений для кода, начинаются с символа # и продолжающиеся до конца строки
2. ***встроенная документация***, внедряемая в текст скрипта и в свойства python-модуля. Такие комментарии представляют собой блок текста, заключенный в ***тройные двойные кавычки***… """<текст>""". Текст, заключенный в ***тройные двойные кавычки***, не форматируется, то есть не обрабатывается *Python* – интерпретатором. Такие комментарии используются для оформления *SQL*-команд.

Следует учесть, что ***блоки встроенной документации*** лишь похожи на комментарии, но таковой не являются.

Комментарии нужны программисту (специалисту группы сопровождения/тестирования), а не интерпретатору *Python*! Комментарии помогают делать исходный текст программы более читаемым…

Выполните следующие примеры в среде *Python IDLE*:

```
>>> #:
>>> # Это комментарий

>>> A = 12 # создать целочисленный объект и связать его с ссылкой A

>>> print(“Привет, моя прелесть =).”) # Выводим текст с помощью print

>>> # print(“Hello World!”) а эта инструкция выполнена не будет

>>> print(“# Это не комментарий”)

>>> """
Эта инструкция выполнена не будет
print("Hello World!")
"""
```

Обратите внимание, что данный фрагмент кода приводит к созданию строкового объекта и в результате получаете следующее сообщение от *shell*:

```
'\nЭта инструкция выполнена не будет\nprint("Hello World!")\n'
>>>
```

Мы можем использовать комментарии, заключенные в ***тройные двойные кавычки*** как обычный текст:

```
>>> A = """Странный комментарий..."""
>>> print(type(A))
<class 'str'>
>>>

>>> B = """Hello World\nHello World"""
>>> C = "Hello World\nHello World"
>>> print(B)

>>> print(C)

>>> D = r"Hello World\nHello World"
>>> print(D)
```

Отдельный интерес вызывает результат работы последнего примера, так как мы косвенно затронули тему ***read-only*** строк. Это связано с тем, что в *Python* ***много разных строк***…

```
>>> A1 = 'This is a string'
>>> A2 = "This is a string"
>>> A3 = """This is a string"""
>>> A4 = '''This is a string'''
```

Продолжим сеанс работы в *Python IDLE* и введем следующие инструкции.

```
>>> A = [A1, A2, A3, A4]
>>> for i in A: print(i)

>>> print(type(A))

>>> for i in A: print(type(i))
```

Обратите внимание на некоторые особенности исполнения *Python* – кода в *shell* (***Python IDLE***) – **после ввода инструкций цикла нам потребовалось лишний раз нажать клавишу <*Enter*>**. Это связано с тем, что среда *shell* выполняет автоматическую разметку вводимого нами кода, подлежащего непосредственному исполнения (имеется в виду тот факт, что мы исполняем команды, а не *Python* – программы).

Ключевые особенности языка программирования *Python*

1. *Python* – это интерпретируемый язык программирования, но поддерживающий средства оптимизации производительности за счет создания ***байт–кода***, то есть промежуточного откомпилированного кода, записываемого в файл с расширением «***.pyc***»…

2. `Python` поставляется по модифицированной ***GNU*** лицензии, то есть его использование для разработчиков становится совершенно бесплатным!
3. `Python` построен по модульному принципу: ***стандартная поставка*** (стандартная библиотека) `Python` содержит ***минимальный набор модулей*** пакетов (библиотек). Если разработчику нужны дополнительные модули, он скачивает из ***открытых источников*** и устанавливает их при помощи сервиса ***pip*** либо ***pip3***, входящих в состав среды исполнения **Python**.
4. Python использует ***свой*** стиль оформления исходного текста программ! Разметка текста имеет существенно значение. Неправильно использованный пробельный символ рассматривается как синтаксическая ошибка!
5. Правила оформления исходного текста получили название «***Дзэн Питона***»:

   1) *Beautiful is better than ugly*. Красивое лучше уродливого.
   2) *Explicit is better than implicit*. Явное лучше неявного.
   3) *Simple is better than complex*. Простое лучше сложного.
   4) *Complex is better than complicated*. Сложное лучше усложнённого.
   5) *Flat is better than nested*. Плоское лучше вложенного.
   6) *Sparse is better than dense*. Разрежённое лучше плотного.
   7) *Readability counts*. Удобочитаемость важна.
   8) *Special cases aren't special enough to break the rules*. Частные случаи не настолько существенны, чтобы нарушать правила.
   9) *Although practicality beats purity*. Однако практичность важнее чистоты.
   10) *Errors should never pass silently*. Ошибки никогда не должны замалчиваться.
   11) *Unless explicitly silenced*. За исключением замалчивания, которое задано явно.
   12) *In the face of ambiguity, refuse the temptation to guess*. В случае неоднозначности сопротивляйтесь искушению угадать.
   13) *There should be one — and preferably only one — obvious way to do it*. Должен существовать один — и, желательно, только один — очевидный способ сделать это.
   14) *Although that way may not be obvious at first unless you're Dutch*. Хотя он может быть с первого взгляда не очевиден, если ты не голландец.
   15) *Now is better than never*. Сейчас лучше, чем никогда.
   16) *Although never is often better than *right* now*. Однако, никогда чаще лучше, чем прямо сейчас.
   17) *If the implementation is hard to explain, it's a bad idea*. Если реализацию сложно объяснить — это плохая идея.
   18) *If the implementation is easy to explain, it may be a good idea*. Если реализацию легко объяснить — это может быть хорошая идея.
   19) *Namespaces are one honking great idea — let's do more of those!* Пространства имён — прекрасная идея, давайте делать их больше!

6. В *Python* отказались от традиционного представления программной переменной. Если в C/C++, *Java* и других языках ***атомы данных*** связаны с переменными, имеющими определенный тип. То в *Python* для доступа к данным используются ссылки. Ссылка может ссылаться на объект любого типа данных, она может быть изменена, то есть быть ***переадресована*** на другой объект в памяти. Ссылку можно даже удалить, а потом создать заново.
7. *Python* поддерживает несколько парадигм программирования, включая процедурную, объектно-ориентированную, функциональную. Развитие и поддержка *Python* осуществляется *Python – сообществом*, возглавляемым голландским программистом Гвидо ван Руссом, признанным «великодушным пожизненным диктатором» (*BDFL*) проекта *Python*.
8. Пошаговое исполнение *Python* – программ накладывает некоторые ограничения на процесс поиска ошибок в коде… Поэтому приходится использовать различные сторонние программные инструменты, содержащие семантические анализаторы кода, такие как *MS Visual Studio*, *Oracle NetBeans*, *Eclepse* и другие.

Остальное Вы сами увидите по мере изучения языка *Python*.

Особенности запуска Python – программ в UNIX/Linux

По правилам, принятым в *UNIX/Linux*, первая строка ***скриптовой программы*** должна содержать путь к интерпретатору, то есть программе, отвечающей за исполнение данного кода. Для *UNIX/Linux* систем эта строка принимает следующий вид (один из них).

```
#!/usr/bin/python
```

```
#!/usr/local/bin/python
```

```
#!/usr/bin/python3
```

Можно указать путь к переменным системного окружения

```
#!/usr/bin/env python3
```

Но процедура запуска программы из *shell* останется неизменной.

```
$python3 my_prorg.py
```

Обратите внимание на тот факт, что нужно указывать версию интерпретатора, так как по умолчанию *Linux* поставляется с *Python* версии 2.x, а она ***частично несовместима*** с *Python 3.x*, которую мы будем изучать…

И еще один очень важный факт из мира *UNIX/Linux*: для *Python* – программ (файлов с расширением *.py* и *.pyw*), как для любых пользовательских

программ, необходимо установить права в 755 (*-rwxr-xr-x*). Для непривилегированного пользователя по умолчанию, в лучшем случае, Вы имеете право только на создание, чтение и изменение файлов. А программы нужно еще и исполнять. ***Нормальный UNIX/Linux – пользователь*** всегда ***озабочен*** настройкой прав пользователя, от имени которого он входит в систему. Тем более что он ***никогда не пойдет в Интернет с правами*** `root` (администратора)!

Краткий обзор возможностей среды Python IDLE

Рассмотрим некоторые особенности исполнения инструкций в *Python IDLE*.

```
>>> var1 = 1
>>> var2 = int(1)
>>> var3 = int("1")
>>> var = (var1, var2, var2)
>>> var = (var1 = 1, var2 = int(1))
SyntaxError: invalid syntax
>>> var = (1, int(1), int("1"))
>>>
```

В *Python IDLE* реализован пошаговый отладчик для вводимых инструкций. Вводимые инструкции выполняются сразу послу ввода их в поле приглашения, которое начинается после символов >>>.

В одной строке *Python* – программы можно разместить несколько инструкций разделяя их символом ';' (***точка с запятой***).

```
>>> A = 2; B = 3; C = A + B; print(C)
5
>>>
```

Порядок исполнения инструкций в таком выражении слева направо. Это похоже на механизм использования ***операции «запятая»*** в C/C++.

В *Python* используется ***правило горизонтального выравнивания блоков текста программ***: вложенные блоки выделяются пробельными отступами. Обычно используются 2 либо 4 пробельных символа для формирования отступа. Но внутри модуля нужно использовать только одно правило, иначе интерпретатор выдаст сообщение о синтаксической ошибке.

```
>>> s1 = "Hello, World of Python"
>>>  s2 = "This is error..."

SyntaxError: unexpected indent
>>>
```

Отдельно заслуживает внимания выделение инструкций блока, выполняемое внутри *Python IDLE*, так как оно визуально нарушает правила горизонтального

выравнивания. Для сохранения визуального размещения блока инструкций *Python IDLE* вставляет ***дополнительные пробельные символы***.

```
>>> i = 1
>>> while i < 11:
     print("i = %s" % i)
     i = i + 1


i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
>>>
```

Для циклов можно формировать инструкции, записываемые в одну строку:

```
>>> for i in range(1, 11): print(i)
```

но следующий вариант является более предпочтительным:

```
>>> for i in range(1, 11):
        print(i)
```

Если инструкция является ***слишком длинной***, то её можно перенести на следующую строку одним из следующих способов:

1. В конце строки поставить символ '\'. После этого символа должен следовать символ перевода строки. Другие символы, в том числе и комментарии, недопустимы.

```
x = 100**100 + 10\
    - math.exp(y)
```

2. Можно поместить выражение в круглые скобки.

```
x = (100**100 + 1
     - math.exp(y))
```

3. Определение списка и словаря можно разместить на нескольких строках, так как оно заключается внутри скобок.

```
bob = [“Bob Smith”, 42, 30000,
       “softvare”]
```

Стиль программирования

Стиль программирования определяет:

1. Именование объектов в зависимости от типа, назначения, области видимости.
2. Оформление функций, методов классов, модулей и их документирование в коде программы.
3. Декомпозиция программы на модули с определенными характеристиками.
4. Способ применения отладочной информации.
5. Применение тех или иных функций (методов) в зависимости от предполагаемого уровня совместимости разрабатываемой программы с различными компонентами платформы.
6. Ограничение на использование функций из соображения безопасности (многострадальная функция `eval()` в языке *Python*, являющаяся «образцом дыры» в безопасности системы…).

Что же нам дополнительно сулят требования «***Python Style Guide***»?

- Отступы в 4 пробела.
- Длина строки не должна превышать 79 символов.
- Длинные строки инструкций можно ***логически разбивать*** неявным образом, то есть размещать внутри скобок.

```
def draw(figure, color = "White", border_color = "Black",
         size = 5):
    if color == border_color or \
       size == 0:
        raise "Bad figure"
    else:
        _draw(size, size, (color,
                           bordr_color))
```

- Не рекомендуется ставить пробелы сразу после открывающей скобки или перед закрывающей, перед запятой, точкой с запятой, перед открывающей скобкой при записи вызова функции или индексного выражения. Не рекомендуется ставить более одного пробела вокруг знака равенства в присваиваниях. Пробелы вокруг знака равенства не ставятся в случаях, когда он применяется для указания значения по умолчанию в определении параметров функции или при задании значений именованных аргументов.

```
def func(arg1=5, arg2=3):
    . . .
```

- Рекомендуется применять одиночные пробелы вокруг низкоприоритетных операций сравнения и оператора присваивания. Пробелы вокруг более приоритетных операций ставятся в равном количестве слева и справа от знака операции

```
print("A = %d" % A)   #  текст комментария к инструкции
```

- Символ '#' должен отстоять от комментируемого оператора как минимум на два пробела.
- Все модули, классы и методы, предназначенные для использования за пределами модуля, должны иметь строку встроенной документации.

```
class myClass:
    """Строка описания класса."""  # атрибут __doc__
    print("инструкция, часть конструктора")
    def __init__(self, firstName = "Black", secondName = "Jack"):
        """Конструктор класса"""
        self.firstName = firstName
        self.secondName = secondName
```

- Использование служебной переменной *__version__* для указания номера текущей версии модуля программы.

```
__version__ = 1.02
```

Встроенные типы данных

Тип данных определяет ***множество значений***, которые можно хранить в переменной указанного типа, и ***множество операций***, которые можно выполнять над переменными данного типа. Тип данных является фундаментальным понятием для любого языка программирования.

Язык программирования Python относится к языкам с динамической типизацией. Ранее изученные языки C/C++ относятся к ***слабо типизированным языкам***. Хотя, справедливости ради, стоит отметить, что типизация в *C++* более сильная, нежели в *C*. Что же стало с ***типом переменной*** в *Python*?

В ***привычном языке программирования***, типа *C/C++*, существует понятие типа переменной. В *Python* же, отказались от этого ***понятия***, так как программист оперирует программными переменными – ссылками. ***Ссылка*** связывается с объектом в памяти программы и, в отличие от *C++*, ссылка в *Python* ***может изменять эту связь***, то есть быть связана с другой переменной…

Данное пояснение по части переменных – ссылок ориентировано на студентов, ранее изучавших язык C++, и предназначено для пояснения внутренних механизмов *Python* работы с объектами в памяти.

При объявлении ссылка должна быть инициализирована, то есть связана с объектом в памяти. Например:

```
>>> Var1 = 123   # Создание ссылки, связанной с объектом типа int()
>>> Var2 = None # Создание ссылки, связанной с объектом типа НИЧТО
```

В языке программирования *Python* все встроенные типы данных можно отнести к одной из двух категорий: изменяемые и неизменяемые типы данных.

К встроенным типам данных в *Python* относятся:

1. ***Специальные типы***: *None*, *NotImplemented*, *Ellipsis*.
2. ***Числовые типы***: целые (*int*, *long*, *bool*), вещественные (с плавающей точкой, *float*), комплексно сопряженные (*complex*).
3. ***Последовательности***: изменяемые (списки *list*), неизменяемые (строка *str*, *unicode* – строка *unicode*, кортеж *tuple*).
4. ***Отображение*** – словарь *dict*.
5. ***Объекты***: функции, функции-генераторы, методы (встроенные и пользовательские), классы, экземпляры классов (если имеют метод *__call__*).
6. ***Модули***.
7. ***Файлы*** *file*.
8. ***Вспомогательные типы*** *buffer*.

Ещё раз хочется отметить тот факт, что в *Python* не используется понятие переменная, вместо него используется ссылка:

<Имя_ссылки> = <Объект_в_памяти>

```
>>> A = 10              # целочисленное значение
>>> A = dict()          # словарь
>>> A = list()          # список
>>> def anyFunc():      # определение функции
     print("Hi!")


>>> A = anyFunc         # ссылка – указатель на функцию
>>> A()                 # вызов функции через ссылку - указатель
Hi!
>>> del A               # удалить ссылку, а не объект…
>>> A()                 # пойди туда, не знаю куда...
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    A()
NameError: name 'A' is not defined
>>> A = [1,2,3]         # словарь из трех элементов
     >>> print(type(A))  # получение информации об объекте, связанным
с
<class 'list'>
>>>
```

Во время работы программы связь «***ссылка–объект***» может быть неоднократно изменена…

Правила выбора имен в *Python* совпадают с правилам в языке *C*, на базе которого, кстати, и был разработан язык *Python*. А разработчики языка – голландские студенты, это наложило огромный отпечаток на «***дух разработки***» на *Python*… Ничего плохого не хочу сказать о голландцах, ведь им мы обязаны и концепцией микроядра, прошедшей успешную апробацию в проекте *Minix* (Энрдю Таненбаум).

Забегая вперед, хочется отметить тот факт, что *Python* может быть использован как великолепный калькулятор. Попробуйте на какой-нибудь другой платформе вычислить 100^{100}. В *Python* для этого нужно всего лишь вызвать инструкцию: 100**100 и получить результат.

```
>>> 100**100
10000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
>>>
```

Да, такое вот большое значение и никаких дополнительных библиотек для «***длинной арифметики***».

Операторы и выражения

В некоторых источниках операции принято называть операторами, но никакой роли при трансляции выражений, в которых они используются, это не играет. Из математики мы помним, что выражение – это комбинация операндов и знаков операция, определяющая некоторый вычислительный процесс (вычисление математического выражения, вызов функций, создание объектов в памяти и др.). В мире алгоритмизации сложилось понятие оператора, как символа, обозначающего набор действий по обработке данных.

Так как Python разработан средствами языка С, то большинство операций в нем заимствованы из этого языка, но есть и некоторые различия.

Таблица 1. - Операторы и их применение

Опе-ратор	Название	Объяснение	Примеры
+	Сложение	Суммирует два объекта	`3 + 5` равно `8`; `'a' + 'b'` равно `'ab'`
–	Вычитание	Даёт разность двух чисел; если первый операнд отсутствует, он считается равным нулю	`-5.2` даст отрицательное число, а `50 - 24` равно `26`
*	Умножение	Даёт произведение двух чисел или	`2 * 3` равно `6`. `'la' * 3` равно `'lalala'`

		возвращает строку, повторённую заданное число раз.	
**	Возведение в степень	Возвращает число x, возведённое в степень y	3 ** 4 равно 81 (т.е. 3 * 3 * 3 * 3)
/	Деление	Возвращает частное от деления x на y	4 / 3 равно 1.3333333333333333
//	Целочисленное деление	Возвращает неполное частное от деления	4 // 3 равно 1
%	Деление по модулю	Возвращает остаток от деления	8 % 3 равно 2. -25.5 % 2.25 равно 1.5
<<	Сдвиг влево	Сдвигает биты числа влево на заданное количество позиций. (Любое число в памяти компьютера представлено в виде битов - или двоичных чисел, т.е. 0 и 1)	2 << 2 равно 8. В двоичном виде 2 представляет собой 10. Сдвиг влево на 2 бита равно 1000, что в десятичном виде означает 8
>>	Сдвиг вправо	Сдвигает биты числа вправо на заданное число позиций.	11 >> 1 равно 5. В двоичном виде 11представляется как 1011, что будучи смещённым на 1 бит вправо, равно 101, а это, в свою очередь, ни что иное, как десятичное 5
&	Побитовое И	Побитовая операция И над числами	5 & 3 равно 1
|	Побитовое ИЛИ	Побитовая операция ИЛИ над числами	5 | 3 равно 7
^	Побитовое ИСКЛЮЧИТЕЛЬНО ИЛИ	Побитовая операция ИСКЛ. ИЛИ (XOR)	5 ^ 3 равно 6
~	Побитовое НЕ	Побитовая операция НЕ для числа x соответствует -(x+1)	~5 равно -6.
<	Меньше	Определяет, верно ли, что x меньше y. Все операторы	5 < 3 равно False, а 3 < 5 равно True. Можно составлять произвольные цепочки

		сравнения возвращают True или False. Обратите внимание на заглавные буквы в этих словах.	сравнений: `3 < 5 < 7` равно `True`
`>`	Больше	Определяет, верно ли, что x больше y	`5 > 3` равно `True`. Если оба операнда - числа, то перед сравнением они оба преобразуются к одинаковому типу. В противном случае всегда возвращается `False`
`<=`	Меньше или равно	Определяет, верно ли, что x меньше или равно y	`x = 3; y = 6; x <= y` равно `True`
`>=`	Больше или равно	Определяет, верно ли, что x больше или равно y	`x = 4; y = 3; x >= 3` равно `True`
`==`	Равно	Проверяет, одинаковы ли объекты	`x = 2; y = 2; x == y` равно `True`. `x = 'str'; y ='stR'; x == y` равно `False`. `x = 'str'; y = 'str'; x ==y` равно `True`
`!=`	Не равно	Проверяет, верно ли, что объекты не равны	`x = 2; y = 3; x != y` равно `True`
`not`	Логическое НЕ	Если `x` равно `True`, оператор вернёт `False`. Если же x равно `False`, получим `True`.	`x = True; not x` равно `False`
`and`	Логическое И	`x and y` даёт `False`, если x равно `False` , в противном случае возвращает значение y	`x = False; y = True; x and y` возвращает `False`, поскольку `x` равно `False`. В этом случае Python не станет проверять значение `y`, так как уже знает, что левая часть выражения `'and'` равняется `False`, что подразумевает, что и всё выражение в целом будет равно `False`, независимо от значений всех остальных операндов. Это называется укороченной оценкой булевых (логических) выражений
`or`	Логическое ИЛИ	Если `x` равно `True`, в результате получим `True`, в противном случае получим значение `y`	`x = True; y = False; x or y` равно `True`. Здесь также может производиться укороченная оценка выражений

Таблица 2. - Приоритет операций в Python

Оператор	Описание

`lambda`	лямбда-выражение
`or`	Логическое “ИЛИ”
`and`	Логическое “И”
`not x`	Логическое “НЕ”
`in, not in`	Проверка принадлежности
`is, is not`	Проверка тождественности
`<, <=, >, >=, !=, ==`	Сравнения
`|`	Побитовое “ИЛИ”
`^`	Побитовое “ИСКЛЮЧИТЕЛЬНО ИЛИ”
`&`	Побитовое “И”
`<<, >>`	Сдвиги
`+, -`	Сложение и вычитание
`*, /, //, %`	Умножение, деление, целочисленное деление и остаток от деления
`+x, -x`	Положительное, отрицательное
`~x`	Побитовое НЕ
`**`	Возведение в степень
`x.attribute`	Ссылка на атрибут
`x[индекс]`	Обращение по индексу
`x[индекс1:индекс2]`	Вырезка
`f(аргументы ...)`	Вызов функции
`(выражения, ...)`	Связка или кортеж
`[выражения, ...]`	Список
`{ключ:данные, ...}`	Словарь

Рассмотрим следующие примеры.

```
# filename: test004.py
anInt = -12
aString = 'cart'
aFloat = -3.1415 * (5.0 ** 2)
anotherString = 'shop' + 'ping'
aList = [3.14e10, '2nd elmt of a list', 8.82-4.371j]

# filename: test005.py
import sys; x = 'foo'; sys.stdout.write(x + '\n')

>>> x = 1
>>> y = x = x + 1
>>> print(x, y)

>>> m 12  # здесь специально вставлен код с ошибкой.

>>> m = 12
>>> m %= 7
>>> print(m)
>>> m **= 2
>>> print(m)
>>> aList = [123, 'xyz']  # список - это не массив!
```

```
>>> aList +=[45.6e7]  # к списку прибавили другой список
>>> print(aList)
```

Вопросы для самопроверки:

- Из чего состоит выражение: `Var1 = 14 + 23 / 5`
- Чем отличаются выражения: `10 + 25` от `Var1 = 10 + 25`
- Чем отличаются выражения: `10 + 25` и `"10 + 25"`

Задания (примеры)

Выполните следующий код (инструкции) и дайте пояснения к его работе.

```
33.0345 + 234.657
"H"+"ello"+", "+" world"
"Hi" * 10
1 + 0.897
"Hello" + 123
type("Hi")
type(123)
type(123.345)
type("56")
int("56")
type(int("56")
int(4.098)
int("one")
str(56)
type(str(56))
srt(3.1415926)
1+23
str(1+23)
float(234)
float("234")
float("123" + "456")
"123"+"45"+"."+"098"
"123"+""45"+"."+098A"
float("123"+""45"+"."+098A")
```

Множественное присваивание

Множественное присваивание в Python позволяет делать непривычные вещи…

```
>>> x = y = z =1
>>> print(x, y, z)
>>> x, y, z = 1, 2, "this is a string"
>>> print(x, y, z)
```

```
>>> if True:
     print(r"Ура!!!")
```

```
# filename: test006.py
if True:
    print(r"Ура, заработало!")
else:
    print(r"Слоны улетают на север...")
```

```
>>> x = 3
>>> print(x ** 10)
```

```
>>> x =4; y = r”Пиастры, пиастры...”
>>> print(x); print(y)
```

Вывод в стандартный поток

Разберем особенности форматного вывода в *Python*, он очень похож на форматный вывод с помощью функции *printf*() в *ANSI C* (многих студентов от этих слов пробивает нервная дрожь…).

```
print( [<объект>][, sep=‘ ‘][, end=’\n’][, file=sys.stdout])
```

Функция *print*() преобразует <объект> в строку и посылает её в ***стандартный вывод***. С помощью параметра *file* можно перенаправить вывод в другое место, например в файл.

```
>>> print(“Строка1”, “Строка2”, end=’*’)
>>> print(“Строка3”)
```

```
# filename: test007.py
for n in range(1, 5):
    print(n, end=' ')
print()
```

```
# filename: test008.py
print("""String1  # Блок, заключенный в тройные двойные кавычки,
String2           # представляет сбой неформатируемую строку,
String3""")       # отображаемую 'как есть'...
```

Параметры функции *print*():

sep(arator) – символ-разделитель. Значение по умолчанию – один пробельный символ. Если задать *sep*='' (пустая строка), то стоковые объекты будут выводиться без символа-разделителя, то есть слитно. В *csv*-файлах в качестве символа-разделителя часто используются следующие значения: *sep*='*' либо *sep*='|' и так далее.

end(line) – символ (по умолчанию используется '\n' – переход на новую строку) вставляемый в конец сформированной строки.

Кроме того, в Python есть «системные средства» для вывода текстового сообщения в стандартный поток вывода.

```
>>> import sys
```

```
>>> sys.stdout.write("String1\n")
>>> sys.stdout.write("String2")
```

Ввод данных. Метод `input()`

```
[<Значение>=]input([<Сообщение>])
```

Для использования метода *input*() есть существенное ограничение: при достижении конца файла или при нажатии комбинации клавиш *<Ctrl>+<Z>*, а затем *<Enter>* генерируется исключение *EOFError*.

```
>>> name = input("Как Вас зовут? ")
>>> print("Здравствуй,", name)
```

Замечание. Функция *input*() читает данные из ***стандартного потока ввода***, результатом её вызова является ***строковый объект***.

```
>>> print(r"Введите что-нибудь : ")
>>> A = input()
>>> print(A, ": ", type(A))
```

Строки, формируемые вызовом *input*(), могут быть явно преобразованы к нужному виду.

```
>>> B = (int)("1234")
>>> print(type(B))
>>> C = int(input("Введите целое значение: "))
>>> print("Вы ввели:", C)
```

```
import math
print("Введите значения коэффициентов для квадратного уравнения
      (Ax^2 + Bx + C = 0): ")
A = float(input("A = "))
B = float(input("B = "))
C = float(input("C = "))
D = B**2 - 4*A*C
print("Дискриминант D = %.2f" % Discr)
if D > 0:
    x1 = (-b + math.sqrt(D)) / (2 * a)
    x2 = (-b - math.sqrt(D)) / (2 * a)
    print("x1 = %.2f \nx2 = %.2f" % (x1, x2))
elif D == 0:
    x = -b / (2 * a)
    print("x = %.2f" % x)
else: print("Действительных корней нет")
```

«***REBOOT***»… *Что случится с человеком, если он совершит прыжок из самолёта с парашютом с высоты более 200м, а парашют не раскроется? В России живет человек, у которого парашют не раскрылся, но он остался жив и даже не стал инвалидом! Кто это?*

Десерт

Вам предстоит совершить небольшой экскурс по возможностям библиотеки *Tkinter*, предназначенной для построения графического интерфейса пользователя программ, написанных на языке *Python*.

```
# filename: test009.pyw
from tkinter import *
root = Tk()
Label(root, text="Hello GUI World!").pack()
Button(root, text="EXIT", command=root.quit).pack()
root.mainloop()
```

```
# filename: test010.pyw
from tkinter import *
msg="Hello GUI World!"
root = Tk()
Label(root, text=msg).pack()
Button(root, text="EXIT", command=root.quit).pack()
root.mainloop()
```

```
# filename: test011.pyw
from tkinter import *
name=input("Hello, hello. What is your name? ")
msg="Hello, "+name
root = Tk()
Label(root, text=msg).pack()
Button(root, text="EXIT", command=root.quit).pack()
root.mainloop()
```

```
# -*- coding: cp1251 -*-
# filename: test012.pyw
from tkinter import *
win = Tk()
win.title("Python GUI")
win.mainloop()
```

```
# -*- coding: cp1251 -*-
# filename: test013.pyw
from tkinter import *
win.resizable(0, 0)  # запретить пользователю изменять размер окна
win = Tk()
win.title("Python GUI")
win.mainloop()
```

Метод *tkinter.resizable()* позволяет управлять возможностью для изменения размера «главного окна» *Python*-приложения

```
resizable(<по_горизонтали>, <по_вертикали>)
```

Управляющего параметра может принимать значение *True* (1) или *False* (0)

```
win.resizable(True, False)
```

```
# -*- coding: cp1251 -*-
# filename: test014.pyw
from tkinter import *
win.geometry(“200x100”)
win.resizable(0, 0)
win = Tk()
win.title("Python GUI")
win.mainloop()
# END ------------------
```

```
# filename: test015.pyw
from tkinter import *
root1 = Tk()
root2 = Tk()
root1.after(500, root1.mainloop())
root2.mainloop()
# END ------------------
```

```
# filename: test016.pyw
from tkinter import *

def button_clicked():
    print(u"Button clicked!")
root = Tk()
button1 = Button()
button1.pack()

button2 = Button(root, bg = "red",
                 text = u"Click me!", command = button_clicked)
button2.pack()
root.mainloop()
# END ------------------
```

```
# filename: test017.pyw
import tkinter as tk
import time

def button_clicked():
    print(u"Button clicked!")
    button['text'] = time.strftime('%H:%M:%S')
root = tk.Tk()

button = tk.Button(root, bg = "red")
button.configure(  text  =  time.strftime('%H:%M:%S'),  command  =
button_clicked)
button.pack()
```

```
root.mainloop()
# END ------------------

# filename: test018.pyw
import tkinter as tk
from random import random

def button_clicked():
    button['text'] = button['bg']
    bg   =   "#%0x%0x%0x"   %   (int(random()*16),   int(random()*16),
int(random()*16))
    button['bg'] = bg
    button['activebackground'] = bg
root = tk.Tk()

button = tk.Button(root, command = button_clicked)
button.pack()
root.mainloop()
# END ------------------

# filename: test019.pyw
from tkinter import *
def button_clicked():
    if label.winfo_viewable():
        label.grid_remove()
    else:
        label.grid(row=1, column=2)
root = Tk()
lab1 = Label(root, text="   ").grid(row=1, column=1)
lab2 = Label(root, text="   ").grid(row=2, column=1)
label = Label(text = u'Help me...')
label.grid(row=1, column=2)
button = Button(root, text=u"Hello from Label-widget!",
                  command = button_clicked)
button.grid(row=2, column=2)
root.mainloop()
# END ------------------

# filename: test020.pyw
from tkinter import *
win = Tk()
win.title(u"Простой калькулятор")
def calc(e):
    try:
        lab2['text'] = str(eval(ent.get()))
    except:
        lab2['text'] = u"Выражение введено с ошибкой"
lab = Label(win, text = r"Введите математическое выражение из чисел и
знаков +, -, *, /, //, %, **")
lab.pack()
```

```
ent = Entry(win, width=50, bg='white', justify=CENTER)
ent.focus()
ent.pack()
ent.bind('<Return>', calc)
lab2 = Label()
lab2.pack()
win.mainloop()
# END --------------------
```