Заметки на тему Python

Редакция 1.13, от 21.11.2017

Оглавление

Предистория

Следующий далее текст представляет собой, собственно, «заметки на память», накапливавшиеся по ходу работы с языком Python, на протяжении ряда лет, перемежающейся длительными перерывами с «отходом» в другие языки программирования. В 2013 году эти заметки было предложено оформить как серию из 10-ти статей и опубликован на ресурсе IBM developerWorks. Но время идёт, и часть из написанного там или устарело, или взгляды на него были пересмотрены. И дополнительно накопилось много интересных мелочей. Следующий далее текст представляет собой переработку тех материалов 4-х летней давности, существенно дополненных новыми «находками». С другой стороны, из них выжата была «вода» и некоторые красивости, которые требуют издатели серьёзных журналов.

Это не систематизированное описание языка Python, и не материал пригодный для изучения языка. Это разрознённые заметки о некоторых особенностях языка, показавшихся **мне** интересными, мало освещёнными в публикациях — вопросы и ответы на эти вопросы, которые возникали в работе с Python. Возможно, это покажется интересным и полезным ещё кому-то.

Здесь описывается не **как** программировать на Python — это невозможно описать в несколько десятков страниц и об этом написаны тысяче-страничные руководства (см. библиографию в конце текста). Здесь описываются, главным образом, вопросы **технологии** Python: что и как использовать, как установить и что это даёт.

Беглый обзор, версии

Языку программирования Python посвящено великое множество публикаций. Эта популярность, кроме вездесущей «моды» на новые языки и технологии программирования, во многом связана с тем, что Python **на практике** доказал свою исключительную эффективность для **быстрой разработки** программного обеспечения. Несмотря на то, что временами эта скорость разработки достигается за счёт снижения надёжности и качества получившегося кода, во многих ситуациях такой «размен» оказывается оправданным.

Это **главное** достоинство Python, и поэтому разработчикам, даже являющимся экспертами в других, более традиционных языках программирования, например, C++ или Java, стоит включить в свой арсенал и Python так же, если не как основной язык, то как быстрый инструмент для прототипирования и проверки различных предположений.

Использование

За счёт ряда своих притягательных особенностей, Python приобрёл множество сторонников и начал активно использоваться в самых различных областях:

- системное программирование, написание сценариев, командных файлов, отдельных утилит операционной системы — область, которая до Python традиционно принадлежала таким языкам как `bash`, `perl`, `awk` … ;
- самодостаточные проекты, начиная с отдельных автономных приложений и

заканчивая крупнейшими комплексными проектами, полностью ведущимися на Python (например, проект OpenStack, развивающийся при поддержке RedHat);

- отдельные плагины к крупным проектам (на C/C++), выполняющие задачи конфигурирования и настройки этих проектов, например VoIP-проекты телефонных коммутаторов Asterisk и FreeSWITCH, использующие для быстрого написания рутинных процедур встроенные интерпретирующие языки Python, JavaScript, Lua;
- создание для Python специализированных модулей на C/C++, делающих для Python-проектов доступным всё множество целевых библиотек, разработанных для C/C++ (это ещё одна сторона простоты связи Python с C-кодом, но уже «в обратную сторону» в сравнении с предыдущим пунктом);
- Web-программирование с использованием Python, особенно в серверной его части (back-end), использующей принципы CGI — для этих целей также были созданы развитые среды, например, `Django`;
- создание Python-приложений с графическим интерфейсом (GUI) — за счёт простоты их разработки и сопровождения - при наличии интерфейсов к большинству известных графических библиотек;
- традиционно научные области обработки данных, традиционно относящиеся к области таких специализированных инструментов как `MathCad` и `MathLab`, особенно за счёт простоты визуального представления и отображения результатов в графике Python (проект `NumPy` и библиотека `Matplotlib`);
- развитые интерфейсы Python к `области` компьютерного зрения (проект `OpenCV`), машинного обучения, анализа и работы с большими данными (такой проект как `Scikit-learn`);

И это ещё не всё… Такой широченный набор областей приложения языка Python открывает дополнительные профессиональны преимущества для всех, изучающих и владеющих этим языком.

Ещё одним стратегическим преимуществом Python является то, что программы на Python могут быть, при соблюдении некоторой дисциплины и минимального набора ограничений, независимы от операционной системы. Один и тот же исходный код, без внесения каких-либо изменений, можно с одинаковым результатом исполнять в Windows, Linux, Solaris, FreeBSD, MacOS, а в некоторых случаях и в мобильных средах Android и iOS. С помощью этой технологии можно строить **мульти-платформенные** проекты, не требующие портирования.

Основы синтаксиса и приёмы программирования на Python были многократно описаны в разнообразных публикациях — часть таких, показавшихся самыми интересными автору, перечислены в конце. Поэтому целью этого текста есть не ещё одно изложение принципов работы с Python, а разбор на практических примерах некоторых **тонких** вопросов семантики и использования языка. Подобные вопросы не всегда достаточно освещаются в литературе. Эти аспекты особенно важны для программистов, выросших на традиционных, строго типизированных языках, таких как C++ или Java, для лучшего восприятия отличительных принципов программирования на Python.

Вопросы WEB-программирования, что является, вообще то, самой широкой областью применений Python (а поэтому и наиболее полно описаны в публикациях), меня, как автора, не интересует … от слова «вообще». Всё дальнейшее изложение ориентировано, в основном, на расчётные применения, в частности, в инженерных и научных областях.

Версия 2 или 3?

Прежде, чем переходить к рассмотрению примеров, стоит коротко остановиться на том, какую версию языка Python и интерпретатора Python мы станем использовать. Во многих публикациях и книгах зачастую просто не указывается, к какой конкретно версии

интерпретатора относится предлагаемый исходный код. Поэтому даже простое воспроизведение представленных примеров может привести к получению непредусмотренных результатов или ошибок.

Одновременно существуют две **линии** версий реализации Python: линия версий 2 и линия версий 3. Такая параллельность связана с тем, что при переходе от версии 2 к версии 3 произошли существенные изменения синтаксиса языка, нарушившие совместимость «снизу вверх». Но так как на версии 2 уже было написано очень большое число проектов, то был предусмотрен некоторый период для перехода от версии 2 к версии 3. Этот период, однако, затянулся уже почти на 10 лет, начиная с 2008 года, и только в 2013 году Python версии 3 был объявлен «официальным Python» для дистрибутива Ubuntu операционной системы Linux. Но в то же время, в других дистрибутивах (Fedora, Debian, и др.) версией по умолчанию всё ещё является Python 2. В большинстве операционных систем обе версии Python могут быть инсталлированы **одновременно**, а то, какая версия **интерпретатора** будет использоваться зависит от формы (команды) запуска. Версии Python с своей системе сотрите так:

```
$ python -V
Python 2.7.11
$ python3 -V
Python 3.4.3
```

При некоторой тщательности, код на Python можно писать так, что он будет одинаково успешно выполняться как в Python 2, так и в Python 3. Все показанные примеры написаны именно так. Но «одинаково успешно» вовсе не означает «с одинаковым результатом»: один и тот же программный код, в принципе, может выполнять различные действия в Python 2 и в Python 3, именно из-за отличий в семантике языка, имеющихся между версиями. Это особенно интересно, и на таких тонких отличиях мы будем специально останавливаться.

Примечание: Самой частой причиной невозможности исполнить под Python 3 код, написанный ранее для Python 2, является использование в коде **оператора** `print`. В Python 3 этот оператор перешёл в разряд **функциональных вызовов** `print()`. Но в синтаксисе Python 2 тоже есть вызов функции `print()`, наряду с оператором `print`. Все последующие примеры написаны именно с использованием функции `print()`, чтобы они **без любых изменений** могли запускаться в любой версии Python.

Некоторые конструкции и вызовы версии 3 будут вызывать исключения в версии 2, **и наоборот**. Такие «опасные» конструкции могут выполняться внутри оператора `try`. Пример такого исполнения показан в листинге ниже (его исходный файл `introduse/fact2.py` находиться в каталоге introduse архива примеров, везде далее файлы примеров будут указываться именно так, с указанием каталога) — код, обходящий несовместимости версий (`introduse/fact3.py`):

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys

if len( sys.argv ) > 1:  n = int( sys.argv[ 1 ] )
else: n = int( input( "число?: " ) )

try:               # Python 2
    factorial = lambda z: reduce( lambda x, y: x * y, range( 1, z + 1 ) )
    print( "n={} => n!={}".format( n, factorial( n ) ) )
except NameError:  # Python 3
    import functools
```

```
    factorial = lambda z: functools.reduce( lambda x, y: x * y,
                                            range( 1, z + 1 ) )
    print( "n={} => n!={}".format( n, factorial( n ) ) )
```

В листинге представлен один из возможных вариантов рекурсивного вычисления факториала. Несовместимость здесь вызвана тем, что в Python 2 функция `reduce()` - встроенная, а в Python 3 она уже отнесена к модулю `functools` который должен импортироваться, а вызов функции просто по имени возбуждает исключение `NameError`. Перехват этого исключения в операторе `try`, что и обеспечивает совместимость версий. Показанный код демонстрирует как **может** писатьсся такой код, а вовсе не в назидание, что так **нужно** писать.

К вопросам рекурсии и глубины рекурсии в Python мы ещё отдельно вернёмся, а пока, попутно, обратите внимание как Python работает с очень большими числами и с вычислениями высокой точности ():

```
$ python fact3.py 500
n=500                                   =>                                   n!
=1220136825991110068701238785423046926253574342803192842192413588385845373153881997605496
447502203281863013616477148203584163378722078177200480785205159329285477907571939330603
772960859086270429174547882424912726344305670173270769461062802310452644218878789465754
777149863494367781037644274033827365397471386477878495438489595537537990423241061271326
984327745715546309977202781014561081188373709531016356324432987029563896628911658974769
572087926928871281780070265174507768410719624390394322536422605234945850129918571501248
706961568141625359056693423813008856249246891564126775654481886506593847951775360894005
745238940335798476363944905313062323749066445048824665075946735862074637925184200459369
692981022263971952597190945217823331756934581508552332820762820023402626907898342451712
006207714640979456116127629145951237229913340169552363850942885592018727433795173014586
357570828355780158735432768888680120399882384702151467605445407663535984174430480128938
313896881639487469658817504506926365338175055478128640000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Для детального **анализа** различных случаев несовместимости можно использовать следующее тестовое приложение — некоторые причины потенциальной несовместимости версий (`introduse/3vs2.py`)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import functools
import sys
import re

# кортеж тестируемых выражений языка ('' - разделитель группы):
prgs = '', \
       '10 / 3', '10 / 3.', '10 / float( 3 )', \
       '', \
       'type( "строка UTF-8" )', 'len( "строка UTF-8" )', \
       '', \
       'type( range( 5 ) )', 'range( 5 )', 'list( range( 5 ) )', \
       'type( xrange( 5 ) )', 'map( int, [ "1", "2", "3" ] )', \
       'list( map( int, [ "1", "2", "3" ] ) )', \
       '', \
       'reduce( lambda x, y: x + y, range( 5 ) )', \

def arg_exit( arg, cod=1 ):
    print( 'недопустимый параметр командной строки: {} !'.format( arg ) )
```

```
    sys.exit( cod )

# разбор диапазонов
def dialist( ls ):
    lim = re.split( '\D+', ls )
    print( lim )
    if len( lim ) != 2: arg_exit( ls, 2 )
    try:
        ilim = list( map( int, lim ) )
    except ValueError: arg_exit( ls, 3 )
    print( '===', ilim )
    args = list( range( min( ilim ), max( ilim ) + 1 ) )
    return args

def arglist( la ):
    lb = []
    for x in la:                      # разнос параметров через ,
         lb += re.split( ',', x )
    args = []
    for x in lb:                      # разбор параметров
        try:
            args.append( int( x ) )
        except ValueError:
            args += dialist( x )
    return args

print( 'версия Python {}.{}.{} на платформе {}'. \
       format( sys.version_info.major, sys.version_info.minor,
               sys.version_info.micro, sys.platform ) )
if len( sys.argv ) == 1: # запуск без параметров
    r = functools.reduce( lambda x, y: x + y,
                          map( lambda x: int( len( x ) == 0 ), prgs ) )
    args = list( range( r + 1 ) )
else:                    # обработка списка параметров
    args = arglist( sys.argv[ 1: ] )
sarg = set( args )       # удаление дубликатов

n = 0
for x in prgs:
    if len( x ) == 0:
        n += 1
        if n not in args: continue
        print( '{:02d}: ------------------------------------'.format( n ) )
    else:
        if n not in args: continue
```

```
        try:
            print( '{} = {}'.format( x, eval( x ) ) )
        except ( NameError, SyntaxError ) as err:
            print( '{} => {} !'.format( x, err ) )
        except:
            print( '{} => непонятное исключение !'.format( x ) )
```

В этом листинге анализируемые (потенциально несовместимые по версиям) синтаксические конструкции выписаны как элементы кортежа `prgs` (показано 4 типичные для иллюстрации), а вы можете самостоятельно добавлять туда любые «подозрительные» выражения в форме текстовых строк, которые программа попытается выполнить. Некоторая громоздкость приложения связана с тем, что все выражения в `prgs` разделены на тематические группы (разделителем группы является пустая строка `''`), а программа позволяет параметрами запуска выбрать только обрабатываемые группы (списком или диапазоном), так как с ростом числа тестируемых конструкций вывод становится слишком объёмным.

Вот как выглядит полный вывод данного сценария…

Для Python версии 2:

```
$ python 3vs2.py
версия Python 2.7.3 на платформе linux2
01: -------------------------------------
10 / 3 = 3
10 / 3. = 3.33333333333
10 / float( 3 ) = 3.33333333333
02: -------------------------------------
type( "строка UTF-8" ) = <type 'str'>
len( "строка UTF-8" ) = 18
03: -------------------------------------
type( range( 5 ) ) = <type 'list'>
range( 5 ) = [0, 1, 2, 3, 4]
list( range( 5 ) ) = [0, 1, 2, 3, 4]
type( xrange( 5 ) ) = <type 'xrange'>
map( int, [ "1", "2", "3" ] ) = [1, 2, 3]
list( map( int, [ "1", "2", "3" ] ) ) = [1, 2, 3]
04: -------------------------------------
reduce( lambda x, y: x + y, range( 5 ) ) = 10
```

Для Python версии 3:

```
$ python3 3vs2.py
версия Python 3.2.3 на платформе linux2
01: -------------------------------------
10 / 3 = 3.3333333333333335
10 / 3. = 3.3333333333333335
10 / float( 3 ) = 3.3333333333333335
02: -------------------------------------
type( "строка UTF-8" ) = <class 'str'>
len( "строка UTF-8" ) = 12
03: -------------------------------------
type( range( 5 ) ) = <class 'range'>
```

```
range( 5 ) = range(0, 5)
list( range( 5 ) ) = [0, 1, 2, 3, 4]
type( xrange( 5 ) ) => name 'xrange' is not defined !
map( int, [ "1", "2", "3" ] ) = <map object at 0xb7428eec>
list( map( int, [ "1", "2", "3" ] ) ) = [1, 2, 3]
04: ------------------------------------
reduce( lambda x, y: x + y, range( 5 ) ) => name 'reduce' is not defined !
```

Особый интерес представляет группа 1 с математическими вычислениями, так как достаточно неожиданно, что результат элементарной операции деления целочисленных операндов будет различаться в разных версиях. Крупные приложения, со множеством математических вычислений, могут показывать в таких условиях очень неожиданные результаты.

Во всём дальнейшем изложении мы будем, по возможности, показывать код, запуск которого даёт одинаковые результаты как в версии 2, так и в версии 3. Когда это полезно, будут демонстрироваться сравнительные результаты для различных версий (а иногда и для различных операционных систем: Linux и Windows).

В некоторых случаях различия между версиями могут быть настолько существенными, что написание переносимого кода крайне усложняется (например, при создании межязыковых интерфейсов Python к C/C++).

Производительность

Измерять производительность кода на разных языках программирования посредством хронометрирования — дело неблагодарное! Потому как существует специфика, под которую «заточен» каждый язык, и один класс задач может выполняться быстрее на языке A чем B, а в другом классе задач соотношение для тех же языков будет в точности обратным. Поэтому попытки точной оценки скорости программ сравнением — это из области спекуляций.

Тем не менее, понятно, что **порядки** производительности можно прикинуть. И понятно, что скорость выполнения кода на интерпретирующем языке Python будет **всегда** ниже, чем эквивалентного кода на компилирующих языках типа C, C++, или Go.

Для сравнительной оценки скорости выполнения нужно подобрать задачу с очень высокой степенью роста вычислительной сложности от размерности данных. Для грубых оценок вполне пригодна задача рекурсивного вычисления чисел Фибоначчи. Например, на языке C (как заведомо самый быстрый эталон) это может выглядеть так:

```
#include <stdio.h>
unsigned long fib( int n ) {
   return n < 2 ? 1 : fib( n - 1 ) + fib( n - 2 );
}

int main( int argc, char **argv ) {
   unsigned num = atoi( argv[ 1 ] );
   printf( "%ld\n", fib( num ) );
   return 0;
}
```

На Python аналогичный код:

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

def fib( n ) :
```

```
    if n < 2 : return 1
    else: return fib( n - 1 ) + fib( n - 2 )

n = int( sys.argv[ 1 ] )
print( "{}".format( fib( int( sys.argv[ 1 ] ) ) ) )
```

Сравнительное выполнение (язык C, Python 2, Python 3):

```
# time nice -19 ./fibo 35
14930352
real   0m0.064s
user   0m0.061s
sys    0m0.002s

# time nice -19 python fibo.py 35
14930352
real   0m6.460s
user   0m6.427s
sys    0m0.007s

# time nice -19 python3 fibo.py 35
14930352
real   0m8.202s
user   0m8.176s
sys    0m0.003s
```

Отсюда следует, что на некоторых классах алгоритмов, главным образом сложных вычислительных, вы должны ожидать возможность замедления выполнения до 100 или более раз по сравнению с компилирующими языками, непосредственно выполняющими машинный код.

Приятно в этом эксперименте то, что Python 3, в силу существенного усложнения его семантики (по сравнению с 2), проигрывает Python 2 что-то порядка 20-30% — ещё годах в 2013-2014-х этот проигрыш составлял 2-3 **раза**. Идёт активное развитие линии Python 3-й версии…

Типы данных

Python располагает широким набором типов данных и механизмов высокого уровня для их структуризации (списки, кортежи, …). Всё это замечательно и многократно описано — и какие типы, и как их использовать. Мы же проведём некоторое исследование **способов** типизации объектов, доступных в Python. Это может оказаться особенно интересно для программистов, работающих с традиционными языками программирования со строгой типизацией, например, PASCAL, C/C++ или Java, потому как подход, применяемый для динамической типизации в Python, радикально отличается от вышеупомянутых языков. Поэтому этот вопрос необходимо подробно рассмотреть для лучшего восприятия отличительных принципов программирования на Python.

Динамическая типизация

В языках со строгой типизацией переменных используется **статический** способ типизации. Так, в PASCAL, C++, Java и других подобных языках любой объект должен быть **предварительно** описан с привязкой его имени к одному из типов данных. Сам этот тип (или класс) данных может быть как предопределён в языке, так и сконструировано

разработчиком. Статический способ типизации может отличаться в том, что типизация может быть структурной (как в PASCAL), так и именной (как в C++), но любой объект будет относиться к своему типу, использовавшемуся для его создания, и только к этому типу на всём протяжении своего жизненного цикла. Это относится даже к языкам с нестрогой типизацией, таким как ранний FORTRAN, Perl или BASIC, не требующих явного предварительного объявления переменных — тип объекта остаётся неизменным от момента его создания на протяжении всей его жизни.

В Python же используется другой подход с **динамической** типизацией объектов, где:

- любой объект программы является ссылкой;
- типом объекта является то, на что он в данный момент ссылается;
- тип объекта может произвольно меняться по ходу выполнения кода, когда ссылка начинает ссылаться на совершенно другой объект (например, в результате операции присвоения), возможно, совершенно другого типа.

Естественно, что при таком подходе информация о типе данных объекта должна храниться в самом объекте, и быть доступна программисту динамически. В листинге ниже представлен пример с использованием динамической типизации.

Динамическое изменение типа данных в Python (`types/tp2.py`):

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def show():
    msgt = '{}'.format( type( i ) )
    msgi = 'id={:09x}'.format( id( i ) )
    msgv = '{}'.format( str( i ) )
    try:
        msgh = 'hash={:012d}'.format( hash( i ) )
    except TypeError:
        msgh = 'не хэшируемый тип!'
    print( 'i - это : {:23} |{} |{} ==> {}'.format( msgt, msgi, msgh, msgv ) )

i = 1; show()
i = 1.5e-2; show()
i = complex( 3.0, 5.5 ); show()
i = "теперь это UNICODE строка"; show()
i = [ 1, 2, 3 ]; show()
i = [ 3 * x for x in range( 3 ) ]; show()   # списковая сборка
i = [ [ x, x**2 ] for x in range( 3 ) ]; show()
i = ( 1, 2, 3 ); show()
i = { 1, 2, 3 }; show()                     # множество
i = { 1:"one", 2:"two", 3:"three" }; show() # словарь
i = lambda x: "фиктивная функция"; show()
i = compile( 'lambda x: "ещё одна фиктивная функция"', '', 'eval' ); show()
i = iter( '12345' ); show()

class own1:
    def __init__( self, id ):
```

```
        self.id = id


i = own1( 987 ); show()
i = own1; show()

class own2:
    def __init__( self, id ):
        self.id = id
    def __hash__( self ):
        return None


i = own2( 987 ); show()
i = own2; show()
```

Здесь одна и та же переменная `i` последовательно проходит через ряд операций присваивания, на каждом из которых она меняет свой **тип** и **значение**. Ниже приведен фрагмент вывода с результатами запуска данного примера для версий Python 2 и 3 с соответствующими отличиями:

```
$ python ./tp2.py
i - это : <type 'int'>     |id=0084e10b0 |hash=000000000001 ==> 1
i - это : <type 'float'>   |id=0084ea304 |hash=-02061781074 ==> 0.015
i - это : <type 'complex'> |id=0b765d9c8 |hash=-00345735165 ==> (3+5.5j)
i - это : <type 'str'>     |id=0b7652360 |hash=000132808738 ==> теперь это ...
i - это : <type 'list'>    |id=0b76c0f6c |не хэшируемый тип! ==> [1, 2, 3]
i - это : <type 'list'>    |id=0b7662cec |не хэшируемый тип! ==> [0, 3, 6]
i - это : <type 'list'>    |id=0b76c0f6c |не хэшируемый тип! ==> [[0, 0], [1, 1] ...
i - это : <type 'tuple'> |id=0b7661784 |hash=-00378539185 ==> (1, 2, 3)
i - это : <type 'set'>   |id=0b76acf7c |не хэшируемый тип! ==> set([1, 2, 3])
i - это : <type 'dict'>  |id=0b766379c |не хэшируемый тип! ==> {1: 'one', 2: 'two' ...
i - это : <type 'function'> |id=0b765b10c |hash=-00881435888 ==> <function <lambda> ...
i - это : <type 'code'>  |id=0b76578d8 |hash=-02089338066 ==> <code object <module> ...
i - это : <type 'iterator'> |id=0b766e52c |hash=-00881430958 ==> <iterator object ...
i - это : <type 'instance'> |id=0b766e4ac |hash=-00881430966 ==> <__main__.own1 ...
i - это : <type 'classobj'> |id=0b765626c |hash=-00881437146 ==> __main__.own1
i - это : <type 'instance'> |id=0b766e54c |не хэшируемый тип! ==> <__main__.own2 ...
i - это : <type 'classobj'> |id=0b765620c |hash=-00881437152 ==> __main__.own2
```

```
$ python3 ./tp2.py
i - это : <class 'int'>    |id=0475b1580 | hash=000000000001 ==> 1
i - это : <class 'float'>  |id=00902370c | hash=001578400478 ==> 0.015
i - это : <class 'complex'> |id=0b73dd1e8 | hash=-01068741806 ==> (3+5.5j)
i - это : <class 'str'>   |id=0b743df60 | hash=001282498349 ==> теперь это ...
i - это : <class 'list'>  |id=0b7475c6c | не хэшируемый тип! ==> [1, 2, 3]
i - это : <class 'list'>  |id=0b747574c | не хэшируемый тип! ==> [0, 3, 6]
i - это : <class 'list'>  |id=0b7475c6c | не хэшируемый тип! ==> [[0, 0], [1, 1] ...
i - это : <class 'tuple'> |id=0b7474194 | hash=-00378539185 ==> (1, 2, 3)
```

```
i - это : <class 'set'>   |id=0b746ab1c | не хэшируемый тип! ==> {1, 2, 3}
i - это : <class 'dict'>  |id=0b7505bdc |не хэшируемый тип! ==> {1: 'one', 2: 'two' ...
i - это : <class 'function'> |id=0b742c26c |hash=-00881578970 ==> <function <lambda> ...
i - это : <class 'code'>  |id=0b742eb10 |hash=-00858576570 ==> <code object <module> ...
i - это : <class 'str_iterator'> |id=0b743530c |hash=-00881576656 ==> <str_iterator ...
i - это : <class '__main__.own1'>  |id=0b743538c  |hash=-00881576648  ==>
<__main__.own1 ...
i - это : <class 'type'> | id=009118fdc | hash=-01064232707 ==> <class '__main__.own1'>
i - это : <class '__main__.own2'>  |id=0b74353c  |не хэшируемый тип!  ==>
<__main__.own2 ...
i - это : <class 'type'> | id=009116c14 | hash=001083250369 ==> <class '__main__.own2'>
```

К значениям `id` и `hash` мы вернёмся позже, но уже сейчас видно, как изменяется **тип** переменной `i` и соответствующее этому типу значение (последняя колонка, после разделителя '==>'). Переменная `i`, начав с числовых значений, последовательно становится строкой, списком, множеством, функцией, **классом** и **объектом** этого класса... — то есть фактически всем чем угодно! Эта особенность Python принципиально отличает его от классических (компилируемых и статически типизированных) языков программирования.

Изменяемые и неизменяемые данные

Все типы данных в Python относятся к одной из 2-х категорий: изменяемые (**mutable**) и неизменяемые (**unmutable**).

Примечание: Во всех русскоязычных переводах используется терминология «изменяемый» и «неизменяемый». Это не самый удачный вариант, так как он вносит неоднозначность, ассоциируясь с некоей константностью. Термины «мутирующий» и «немутирующий» были бы уместнее и точнее отображали бы суть происходящего: может ли объект этого типа изменять свою **структурность**? Но такие термины неблагозвучные и, поэтому, не прижились. Например: строка `s = 'abcdef'` - это неизменяемый тип, так как в Python нельзя, в отличие от C/C++ изменить некоторый одиночный символ в строке, например, через `s[ 2 ] = 'z'`, не говоря уже о том, чтобы вставить символ **внутрь** строки. Но это можно сделать `s = s[ :2 ] + 'z' + s[ 3: ]` и получить в результате, требуемую строку `'abzdef'`, только это будет совершенно **другая строка**, размещённая по совершенно другому адресу в памяти, а `s` — переустановленная ссылка, с тем же именем, на эту новую строку. Но изменить саму строку или её длину (её структурность) по текущей ссылке — невозможно. В этом и состоит неизменяемость объекта — это не константность, так как его **значение** можно изменить, но это будет уже ссылка на другой объект с этим новым значением.

Многие из предопределённых типов данных Python — это типы неизменяемых объектов: числовые данные (`int`, `float`, `complex`), символьные строки (`class 'str'`), кортежи (`tuple`). Другие типы определены как изменяемые: списки (`list`), множества(`set`), словари (`dict`). Вновь определяемые пользователем типы (классы) могут быть определены как неизменяемые или изменяемые. Изменяемость объектов определённого типа является принципиально важной характеристикой, определяющей, может ли объект такого типа выступать в качестве ключа для словарей (`dict`) или нет, как будет показано в следующем разделе.

Ключи в словарях

Словари — один из встроенных сложно структурированных типов (коллекций) Python (наряду со списками, кортежами, множествами). Но словари в Python имеют особое, исключительное значение, так как сам интерпретатор Python весь основывается на словарях, а текущее пространство имён в точке исполнения — это словарь, ключами которого являются имена объектов.

Ключами элементов словаря могут быть численные значения, строки, кортежи (`tuple`) и объекты **собственных классов** или даже функции, как показано ниже:

```
def pow2( i ):
    return i * i


d = { 1:"111", 'two':"222", (3, 5, 7):"333", pow2:"444" }; print( d )
```

Эти типы данных объединяет то, что все они являются неизменяемыми (**unmutable**). Изменяемые типы данных не могут быть ключами словаря, и их использование в таком качестве вызовет исключение `TypeError` **периода выполнения** с сообщением «`unhashable type`» (нехэшируемый тип) — исключение возникнет прямо во время исполнения, так как интерпретатор не может определить хэшируемость типа ключа до его фактического выполнения. Как становится понятно, неизменяемые типы данных Python — это типы данных для которых не определено значение, возвращаемое функцией `hash()`. Теперь можно снова вернуться к результатам кода выше, где видно для каких типов данных возвращается значение вызова `hash()`, а для каких такой вызов возбуждает исключение `TypeError`.

Примечание: Ещё одно значение, показанное в таблице, это значение, возвращаемое вызовом `id()` — **уникальный** для каждого объекта программы идентификатор, из числа специальных атрибутов объектов. Для 32-битной реализации на платформе x86 `id()` представляется логическим адресом размещения объекта в оперативной памяти. Но это совершенно необязательно для других платформ. Важное требование состоит в том, чтобы для любых 2-х объектов их `id()` различались, и таким образом этот атрибут уникально идентифицирует каждый объект.

Словарь, как можно понять из документации, организован как хэш-таблица на `2**N` входов, где `N` — такое минимальное число, что `2**(N-1)` превышает текущее число элементов словаря (с ростом словаря значение `N` может увеличиваться). Неизученным остался вопрос, как создать свой собственный класс, объекты которого могут (или не могут) быть использованы в качестве ключей при построении словарей. В следующем листинге представлен сценарий (`types/tp-hash.py`), в котором демонстрируется работа с хэшируемыми и нехэшируемыми типами данных.

Хэшируемые и нехэшируемые типы данных:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


def show_instances( v ):
    print( "объект: {0} (число таких объектов {1})".format( v, v.numInstances ) )
    try:
        print( "{0}, class: {1}, dictionary: {2}".format( type( v ),
               v.__class__, v.__dict__ ) )
        print( "id=0x{0:012x}, hash={1:012}[0x{0:012x}]".format( id( v ),
               hash( v ), hash( v ) ) )
    except TypeError:
        print( "Не хэшируемый объект! : {0}".format( type( v ) ) )


class key0( object ):
    numInstances = 0
    def __init__( self, id ):
```

```
        self.id = id
        key0.numInstances = key0.numInstances + 1

class key1( key0 ):
    numInstances = 0
    def __init__( self, id ):
        key0.__init__( self, id )
        key1.numInstances = key1.numInstances + 1
    def __hash__( self ):
        return self.id * self.id

class key2( key0 ):
    numInstances = 0
    def __init__( self, id ):
        key0.__init__( self, id )
        key2.numInstances = key2.numInstances + 1
    def __hash__( self ):
        return 123

class key3( key0 ):
    numInstances = 0
    def __init__( self, id ):
        key0.__init__( self, id )
        key3.numInstances = key3.numInstances + 1
    def __hash__( self ):
        raise TypeError

class key4( key0 ):
    numInstances = 0
    def __init__( self, id ):
        key0.__init__( self, id )
        key4.numInstances = key4.numInstances + 1
    def __hash__( self ):
        return None

tk = ( key0, key1, key2, key3, key4 )       # кортеж выбираемых классов
for clas in tk:                             # выбор класса!
    print( "--------------------------------" )
    t = [ clas( x ) for x in range( 1, 4 ) ] # список объектов выбранного класса
    show_instances( t[ 2 ] )
    try:
        d = { t[ 0 ]:"#1", t[ 1 ]:"#2", t[ 2 ]:"#3" }; # словарь с ключами из классов
        print( d )
        print( d[ t[ 1 ] ] )
    except TypeError:
```

```
        print( "Не может использоваться как ключ словаря!" )
    print( "-----------------------------------------------------------" )
```

Не будем показывать результирующий вывод этой программы, так как он получится очень объёмным (журнал выполнения программы находится там же в архиве). Но можно сделать краткие выводы (которые, кстати, совпадают для Python 2 и Python 3):

1. любой класс, определённый программистом, может быть хэшируемым, и его объекты в таком случае могут использоваться в качестве ключей для словарей;
2. если в классе (`class key0`) не был определен собственный метод `__hash__()`, то будет использоваться его реализация из базового класса `object`, который возвращает в этом случае `id()`;
3. вы можете произвольно переопределить результат вызова `hash()` для объектов собственного класса (`class key1`), реализовав в классе собственный метод `__hash__()`;
4. хэш-функция словаря строится на базе возвратов `hash()`, но вовсе не тождественна им: если класс (`class key2`) даже возвращает константное значение `hash()` для любых созданных объектов этого класса, то и в этом случае на нём строится нормально функционирующий словарь;
5. для того, чтобы класс был идентифицирован как неизменяемый (`class key4`) достаточно чтобы его собственная реализация `__hash__()` возвращала `None` - это будет «сигналом» для интерпретатора, что такой класс **нельзя** использовать в качестве ключа для словарей.

В показанном примере есть ещё несколько любопытных деталей из области динамической типизации, которые следует упомянуть:

- `tk` (в примере) — это последовательность **классов** (в данном случае это кортеж, но это может быть и список, и множество, и другие контейнерные типы) — не объектов этих классов, а именно **классов** как определений типов;
- **присвоение** в цикле переменной `clas` поочерёдно различных значений класса из этой последовательности, с последующим созданием необходимых объектов (в нужном количестве) класса `clas` (имя переменной);

Выше рассмотрены некоторые не очевидные вопросы представления типов данных в Python, необычные для программиста, использующего «классические» языки программирования. К этим деталям мы будем неоднократно возвращаться в ходе дальнейшего рассмотрения, но уже при поверхностном рассмотрении становится видной особая гибкость подобного подхода, в чём-то напоминающая возможности языка Lisp. Из этой же гибкости проистекает и высокая скорость **разработки** программного обеспечения на подобных языках (разработки, но не исполнения). Но эта же гибкость может стать причиной тяжёлых ошибок вследствие неправильно понятой семантики языка.

Функциональное программирование

Python не является языком функционального программирования, если сравнивать его с такими истинно функциональными языками как, скажем Lisp, Scheme, Ocaml, или Haskell. Но развитая объектность языка Python позволяет на нём выразить некоторые идеи функционального программирования и использовать функциональный стиль написания кода.

Существует большое количество публикаций, посвящённых реализациям концепций функционального программирования на языке Python, но большая часть этих материалов написана одним автором — Девидом Мертцом (David Mertz, см. ссылки в конце), или отталкиваются от его изложения. Многие из этих статей уже устарели и разбросаны по различным сетевым ресурсам. В этой части мы попробуем освежить и упорядочить доступную информацию, особенно учитывая большие различия, имеющиеся между

версиями Python линии 2 и линии 3.

Функции

Функции в Python определяются 2-мя способами: через определение `def` или через анонимное описание `lambda`. Оба этих способа определения доступны, в той или иной степени, и в некоторых других языках программирования (даже в позднем C++). Особенностью же Python является то, что функция в нём является таким же именованным объектом кода, как и любой другой объект некоторого типа данных, скажем, как целочисленная переменная. В листинге ниже представлен простейший пример (файл `functional/func.py`).

Определения функций:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys

def show( fun, arg ):
    print( '{} : {}'.format( type( fun ), fun ) )
    print( 'arg={} => fun( arg )={}'.format( arg, fun( arg ) ) )

if len( sys.argv ) > 1: n = float( sys.argv[ 1 ] )
else: n = float( input( "число?: " ) )

def pow3( n ):                      # 1-е определение функции
    return n * n * n
show( pow3, n )

pow3 = lambda n: n * n * n          # 2-е определение функции с тем же именем
show( pow3, n )

show( ( lambda n: n * n * n ), n ) # 3-е, использование анонимного описание функции
```

При вызове всех трёх объектов-функций мы получим один и тот же результат:

```
$ python func.py 1.3
<type 'function'> : <function pow3 at 0xb7662844>
arg=1.3 => fun( arg )=2.197
<type 'function'> : <function <lambda> at 0xb7662bc4>
arg=1.3 => fun( arg )=2.197
<type 'function'> : <function <lambda> at 0xb7662844>
arg=1.3 => fun( arg )=2.197
```

Ещё более отчётливо это проявляется в Python версии 3, в которой всё является классами (в том числе, и целочисленная переменная), а функции являются объектами программы, принадлежащими к классу `function`:

```
$ python3 func.py 1.3
<class 'function'> : <function pow3 at 0xb74542ac>
arg=1.3 => fun( arg )=2.1970000000000005
<class 'function'> : <function <lambda> at 0xb745432c>
```

```
arg=1.3 => fun( arg )=2.1970000000000005
<class 'function'> : <function <lambda> at 0xb74542ec>
arg=1.3 => fun( arg )=2.1970000000000005
```

Примечание. Существуют ещё 2 типа объектов, допускающих осуществление функционального вызова — функциональный метод класса и функтор, о которых мы поговорим позже.

Если функциональные объекты Python являются такими же объектами, как и другие объекты данных, значит, с ними можно и делать (помимо вызова как функции) всё то, что можно делать с любыми данными:

• динамически **изменять** в ходе выполнения;

• встраивать в более сложные структуры данных (коллекции);

• присваивать другим переменным;

• передавать в качестве параметров и возвращаемых значений и т.д.

На этом (манипуляции с функциональными объектами как с объектами данных) и базируется функциональное программирование. Python, конечно, не является настоящим языком функционального программирования, так, для полностью функционального программирования существуют специальные языки: Lisp, Planner, а из более свежих: Scala, Haskell, Ocaml, ... Но в Python можно «встраивать» приёмы функционального программирования в общий поток императивного (командного) кода, например, использовать методы, заимствованные из полноценных функциональных языков. Т.е. «сворачивать» отдельные фрагменты императивного кода (иногда достаточно большого объёма) в функциональные выражения.

Временами спрашивают: «В чём преимущества функционального стиля написания отдельных фрагментов для программиста?». Основным преимуществом функционального программирования является отсутствие побочных эффектов, а, значит, то, что после однократной отладки такого фрагмента в нём, при последующем многократном использовании, не возникнут ошибки за счёт побочных эффектов, связанных с присвоениями и конфликтом имён.

Достаточно часто при программировании на Python используют типичные конструкции из области функционального программирования, например:

```
print ( [ ( x,y ) for x in ( 1, 2, 3, 4, 5 ) \
                      for y in ( 20, 15, 10 ) \
                         if x * y > 25 and x + y < 25 ] )
```

В результате запуска получаем:

```
$ python funcp.py
[(2,20), (2,15), (3,20), (3,15), (3,10), (4,20), (4,15), (4,10), (5,15), (5,10)]
```

Функции как объекты

Создавая объект функции оператором `lambda` (как было показано в листинге в начале главы), можно привязать созданный функциональный объект к имени `pow3` в точности так же, как можно было бы привязать к этому же имени числовое значение 123 или строку `"Hello!"`. Этот пример подтверждает статус функций как объектов первого класса в Python. Функция в Python — это всего лишь ещё одно значение, с которым можно что-либо сделать.

Наиболее частое действие, выполняемое с функциональными объектами первого класса, — это передача их во встроенные функции высшего порядка: `map()`, `reduce()` и `filter()`. Каждая из этих функций принимает объект функции в качестве своего первого аргумента.

1) `map( func, s1, s2, s3, ... )` — применяет переданную функцию к каждому элементу в переданном 2-м параметром списке (и последующими параметрами спискам) и возвращает список результатов (той же размерности, что и входной):

```
a = [ 1, 2, 3, 4, 5 ]
b = map( lambda x: 3 * x, a )     # b = [ 3, 6, 9, 12, 15 ]
```

Если в качестве функции указано `None`, подразумевается тождественная функция, если передано **несколько** списков — возвращается список кортежей, в котором каждый кортеж будет содержать по одному элементу из каждого списка:

```
c = [ 101, 102, 103, 104, 105 ]
b = map( None, a, c )             # b = [ (1,101), (2,102), (3,103), (4,104), (5,105) ]
```

2) `reduce( func, s )` — применяет переданную функцию 2-х переменных func к каждому значению в списке и ко внутреннему накопителю результата. Функция применяется к первым 2-м элементам списка, затем результат применяет к 3-му элементу списка, затем к 4-му и так далее... Например, выражение ниже означает 10! (факториал):

```
reduce( lambda n, m: n * m, range( 1, 10 ) )
```

3) `filter( func, s )` — в применяет переданную функцию к каждому элементу списка и возвращает список тех элементов исходного списка, для которых переданная функция вернула значение истинности, например:

```
b = filter( lambda n, n < 4, a ) # b = [ 1, 2, 3 ]
```

Комбинируя только эти три функции, можно реализовать неожиданно широкий диапазон операций потока управления, не прибегая к императивным утверждениям, а используя лишь выражения в функциональном стиле, как показано в следующем листинге (файл `functional/funcH.py`).

Функции высших порядков Python:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
def input_arg():
    global arg
    arg = ( lambda: ( len( sys.argv ) > 1 and int( sys.argv[ 1 ] ) ) or \
                      int( input( "число?: " ) ) )()
    return arg

print( 'аргумент = {}'.format( input_arg() ) )
print( list( map( lambda x: x + 1, range( arg ) ) ) )
print( list( filter( lambda x: x > 4, range( arg ) ) ) )

import functools
print( '{}! = {}'.format( arg, functools.reduce( lambda x, y: x * y,
                                                  range( 1, arg ) ) ) )
```

Этот код несколько усложнён по сравнению с предыдущим примером из-за следующих аспектов, связанных с совместимостью Python версий 2 и 3:

• Функция `reduce()`, объявленная как встроенная в Python 2, в Python 3 была вынесена в модуль `functools` и её прямой вызов по имени вызовет исключение `NameError`, поэтому для корректной работы вызов должен быть оформлен как в примере или включать строку: `from functools import *`.

• Функции `map()` и `filter()` в Python 3 возвращают не список (что уже показывалось при обсуждении различий версий), а объекты-итераторы вида:

```
<map object at 0xb7462bec>
<filter object at 0xb75421ac>
```

Для получения всего списка значений для этих итераторов вызывается функция `list()`.

Поэтому такой код сможет работать в обеих версиях Python:

```
$ python3 funcH.py 7
аргумент = 7
[1, 2, 3, 4, 5, 6, 7]
[5, 6]
7! = 720
```

Если переносимость кода между различными версиями не требуется, то подобные фрагменты можно исключить, что позволит несколько упростить код.

Рекурсия

В функциональном программировании рекурсия является основным механизмом, аналогично циклам в итеративном программировании.

В некоторых обсуждениях по Python неоднократно приходилось встречаться с ошибочными заявлениями, что в Python глубина рекурсии ограничена «аппаратно», и поэтому некоторые действия реализовать невозможно в принципе. В интерпретаторе Python действительно по умолчанию установлено ограничение глубины рекурсии, равным 1000, но это численный **параметр**, который всегда можно переустановить, как показано в следующем листинге (файл `functional/fact2.py`).

Вычисление факториала с **произвольной** глубиной рекурсии:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys

arg = lambda : ( len( sys.argv ) > 1 and int( sys.argv[ 1 ] ) ) or \
               int( input( "число?: " ) )

factorial = lambda x: ( ( x == 1 ) and 1 ) or x * factorial( x - 1 )

n = arg()
m = sys.getrecursionlimit()
if n >= m - 1 :
   sys.setrecursionlimit( n + 2 )
    print( "глубина рекурсии превышает установленную в системе {}, переустановлено в {}".\
          format( m, sys.getrecursionlimit() ) )

print( "n={} => n!={}".format( n, factorial( n ) ) )

if sys.getrecursionlimit() > m :
    print( "глубина рекурсии восстановлена в {}".format( m ) )
```

```
    sys.setrecursionlimit( m )
```

Вот как выглядит исполнение этого примера в Python 3 и в Python2 (правда на самом деле полученное число вряд ли поместится на один экран терминала консоли):

```
$ python3 fact2.py 1001
глубина рекурсии превышает установленную в системе 1000, переустановлено в 1003
n=1001 => n!=4027.....................................................0000000000000
глубина рекурсии восстановлена в 1000
```

Несколько простейших примеров

Выполним несколько простейших трансформаций привычного императивного кода (командного, операторного) для превращения его отдельных фрагментов в функциональные. Сначала заменим операторы ветвления логическими условиями, которые за счёт «отложенных» (lazy, ленивых) вычислений позволяют управлять выполнением или невыполнением отдельных ветвей кода. Вот такая императивная конструкция:

```
if <условие>:
    <выражение 1>
else:
    <выражение 2>
```

Она полностью эквивалентна следующему функциональному фрагменту (за счёт «отложенных» возможностей логических операторов `and` и `or`):

```
# функция без параметров:
lambda: ( <условие> and <выражение 1> ) or ( <выражение 2> )
```

В качестве примера снова используем вычисление факториала. В следующем листинге, для начала сравнения, приведен традиционный императивный код для вычисления факториала (файл `functional/fact1.py`).

Операторное (императивное) определение факториала:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys

def factorial( n ):
    if n == 1: return 1
    else: return n * factorial( n - 1 )

if len( sys.argv ) > 1:
    n = int( sys.argv[ 1 ] )
else:
    n = int( input( "число?: " ) )

print( "n={} => n!={}".format( n, factorial( n ) ) )
```

Аргумент для вычисления извлекается из значения параметра командной строки (если он есть) или запрашивается и вводится с терминала.

Первый вариант изменения уже показывался выше в листинге, где на функциональные выражения были заменены:

- определение функции факториала:

```
factorial = lambda x: ( ( x == 1 ) and 1 ) or x * factorial( x - 1 )
```

- запрос на ввод значения аргумента с консоли терминала:

```
arg = lambda : ( len( sys.argv ) > 1 and int( sys.argv[ 1 ] ) ) or \
                 int( input( "число?: " ) )
n = arg()
```

В файле `functional/fact3.py` появляется ещё одно определение функции, сделанное через функцию высшего порядка `reduce()`:

```
factorial = factorial = lambda z: reduce( lambda x, y: x * y, range( 1, z + 1 ) )
```

Здесь же мы упростим также и выражение для n, сведя его к однократному вызову анонимной (не именованной) функции:

```
n = ( lambda : ( len( sys.argv ) > 1 and int( sys.argv[ 1 ] ) ) or \
                 int( input( "число?: " ) ) )()
```

Наконец, можно заметить, что присвоение значения переменной n требуется только для её использования в вызове `print()` для вывода этого значения. Если мы откажемся и от этого ограничения, то всё приложение выродится вообще в один функциональный оператор (файл `functional/fact4.py`):

```
from sys import *
from functools import reduce
print( 'вычисленный факториал = {}'.format( \
          ( lambda z: reduce( lambda x, y: x * y, range( 1, z + 1 ) ) ) \
             ( ( lambda : ( len( argv ) > 1 and int( argv[ 1 ] ) ) or \
                int( input( "число?: " ) ) )() ) ) )
```

Этот единственный вызов внутри функции `print()` и представляет всё приложение в его функциональном варианте:

```
$ python3 fact4.py
число?: 5
вычисленный факториал = 120
```

Читается ли этот код (файл `fact4.py`) лучше, чем императивная запись (файл `fact1.py`)? Скорее нет, чем да… хотя это вопрос привычки. В чём же тогда его достоинство? В том, что при любых изменениях окружающего его кода, нормальная работа этого фрагмента сохранится неизменной, так как отсутствует риск побочных эффектов из-за изменения значений любых используемых переменных.

Функции высших порядков

При функциональном стиле программирования стандартной практикой является динамическая генерация функционального объекта в процессе исполнения кода, с его последующим вызовом в том же коде. Существует целый ряд областей, где подобная техника может оказаться полезной. Ниже рассмотрены некоторые из основных используемых приёмов (но изобретательность может породить и много других).

Замыкание

Одно из интересных понятий функционального программирования - это **замыкания** (closure). Эта идея оказалась настолько заманчивой для многих разработчиков, что была реализована даже в некоторых не функциональных языках программирования (Perl). Девид Мертц приводит следующее определение замыкания:

Замыкание - это процедура вместе с привязанной к ней совокупностью данных (в противовес объектам в объектном программировании, как: «данные вместе с привязанным к ним совокупностью процедур»).

Смысл замыкания состоит в том, что определение функции «замораживает» окружающий её контекст на **момент определения**. Это может делаться различными способами, например, за счёт параметризации создания функции, как показано в следующем (файл `functional/clos1.py`).

Создание замыкания:

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

def multiplier( n ):    # multiplier возвращает функцию умножения на n
    def mul( k ):
        return n * k
    return mul

mul3 = multiplier( 3 )  # mul3 - функция, умножающая на 3
print( mul3( 3 ), mul3( 5 ) )
```

Вот как срабатывает такая динамически определённая функция:

```
$ python clos1.py
(9, 15)
$ python3 clos1.py
9 15
```

Другой способ создания замыкания — это использование значения параметра по умолчанию в точке определения функции, как показано в следующем листинге (файл `functional/clos3.py`).

Другой способ создания замыкания:

```
n = 3
def mult( k, mul = n ):
    return mul * k

n = 7
print( mult( 3 ) )
n = 13
print( mult( 5 ) )

n = 10
mult = lambda k, mul=n: mul * k
print( mult( 3 ) )
```

Никакие последующие присвоения значений параметру по умолчанию (переменной n) не приведут к изменению ранее определённой функции (функция заморожена), но сама функция (с тем же именем) может быть позже переопределена:

```
$ python clos3.py
9
15
30
```

Частичное применение функции

Частичное применение функции предполагает на основе функции N переменных

определение новой функции с меньшим числом переменных `M < N`, при этом остальные `( N - M )` переменных получают фиксированные «замороженные» значения (используется модуль `functools`). Подобный пример будет рассмотрен ниже.

Функтор

Функтор — это не функция, а объект класса, в котором определён метод с именем `__call__()` (собственно, это то же, когда на C++ в классе **переопределяется** операция ()). При этом, для экземпляра такого объекта может применяться вызов, точно так же, как это происходит и для функций. В листинге (файл `functional/part.py`) демонстрируется использование замыкания, частичного определения функции и функтора, приводящих к получению одного и того же результата.

Сравнение замыкания, частичного определения и функтора:

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

def multiplier( n ):           # замыкания - closure
    def mul( k ):
        return n * k
    return mul

mul3 = multiplier( 3 )         # mul3() - функция 1-й переменной

from functools import partial
def mulPart( a, b ):           # частичное применение функции
    return a * b

par3 = partial( mulPart, 3 )   # par() - функция 1-й переменной

class mulFunctor:              # эквивалентный функтор
    def __init__( self, val1 ):
        self.val1 = val1
    def __call__( self, val2 ):
        return self.val1 * val2

fun3 = mulFunctor( 3 )         # создание нового объекта + вызов метода

print( '{} . {} . {}'.format( mul3( 5 ), par3( 5 ), fun3( 5 ) ) )
```

Вызов всех трёх конструкций для аргумента, равного 5, приведёт к получению одинакового результата, хотя при этом и будут использоваться абсолютно разные механизмы:

```
$ python part.py
15 . 15 . 15
```

Карринг

Карринг (или каррирование, curring) — преобразование функции от многих переменных в функцию, берущую свои аргументы по одному.

Примечание. Это преобразование было введено М. Шейнфинкелем и Г. Фреге и получило своё название в честь математика Хаскелла Карри, в честь которого также назван и язык

программирования Haskell.

Карринг не относится к уникальным особенностям функционального программирования, так каррингговое преобразование может быть записано, например, и на языках Perl или C++. Оператор каррирования даже встроен в некоторые языки программирования (ML, Haskell), что позволяет многоместные функции приводить к каррированному представлению. Но все языки, поддерживающие замыкания, позволяют записывать каррированные функции, и Python не является исключением в этом плане.

В следующем листинге представлен простейший пример с использованием карринга (файл `functional/curry1.py`):

Карринг:

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


def spam( x, y ):
    print( 'param1={}, param2={}'.format( x, y ) )

spam1 = lambda x : lambda y : spam( x, y )


def spam2( x ) :
    def new_spam( y ) :
        return spam( x, y )
    return new_spam


spam1( 2 )( 3 )      # карринг
spam2( 2 )( 3 )
```

Вот как выглядят исполнение этих вызовов:

```
$ python curry1.py
param1=2, param2=3
param1=2, param2=3
```

Параллельное исполнение

Существует большое количество публикаций, описывающих каким образом можно обеспечить параллельное выполнение различных ветвей кода в Python. Здесь мы подробно рассмотрим эти возможности и проанализируем, к каким последствиям может привести их применение.

Параллельное исполнение в современных инструментальных средствах реализуется двумя фундаментальными механизмами: параллельными процессами и параллельными потоками (thread), а все остальные способы являются комбинациями этих вариантов.

Мы начнём изучение аспектов реализации параллельного выполнения в Python именно с потоков, чтобы затем вернуться к процессам. Смысл выбранного порядка рассмотрения, отличающийся от используемого в большинстве публикаций, станет ясен в ходе дальнейшего изложения.

Примечание: Понятия процесс (process) и поток (thread) окончательно разделились, когда аппаратные средства процессоров начали поддерживать защищённый режим с контролем прав доступа (для x86, например, начиная с 386), а операционные системы смогли реализовать изолированные адресные пространства для каждой пользовательской задачи (на основе AVR), и в операционных системах, не разграничивающих адресных пространств (pSOS, VxWorks) эти два понятия (process и thread) вырождаются в единую сущность, в некоторых системах называемую задача (task). Разработки последних лет стараются ввести

ещё более лёгкие механизмы параллелизма и синхронизации, которые преимущественно работают в пространстве пользовательской задачи и минимально затрагивают системные вызовы ядра операционной системы. Из числа таких лёгких механизмов стоит назвать футексы (Fast Userspace muTexes) в новых спецификациях POSIX или Go-рутины языка Go. Но в Python нет таких лёгких механизмов, поэтому мы не будем к ним больше обращаться.

Техника потоков в Python

Механизм потоков в Python реализуется несколькими различающимися модулями, входящими в состав стандартной поставки Python, о чём часто не упоминается в публикациях. Как минимум, это низкоуровневый механизм из модуля `thread` и механизм более высокого уровня из модуля `threading`. Именно последний модуль чаще всего и имеется в виду при обсуждении реализации потоков в Python.

Какой бы механизм не использовался (модуль `threading` экспортирует свои базовые элементы из низкоуровневого `thread` — ниже будут показаны оба), общие принципы работы с потоками в Python остаются неизменными. Потоки в Python не могут быть реализованы с использованием механизмов операционной системы с вытесняющей многозадачностью (preemptive multitasking) и диспетчированием по системному таймеру.

Это связано с тем, что код приложения должен выполняться **внутри** виртуальной машины интерпретатора Python, который сам по себе не является многопоточным и может исполнять только один поток программы. Интерпретатор Python сам переключает потоки в исполняемом коде, но может делать это только после завершения очередного оператора исполняемого байт-кода (virtual instructions, как это названо в документации), а реально выполняет эту операцию границе `N` операторов байт-кода, где `N` обычно равно 100. Но это значение можно изменить вызовом `setcheckinterval()` из модуля `sys`. В листинге представлено приложение `idt.py` (`parallel/set.thread/idt.py`), в котором используется вызов `setcheckinterval()`.

Диагностика и изменение интервала переключения потоков:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys

print( 'версия Python {}.{}.{} на платформе {}'. \
       format( sys.version_info.major, sys.version_info.minor, \
               sys.version_info.micro, sys.platform ) )

first = True; switch = True
while( True ):
    try:
        print( 'интервал *checkinterval : {} '.format( sys.getcheckinterval() ) )
        print( 'интервал *switchinterval : {} сек.'.format( sys.getswitchinterval() ) )
    except AttributeError:
        if first: print( '*switchinterval не реализованы в этой версии Python' );
    switch = False
    else:
        if first: print( '*checkinterval объявлено устаревшими в этой версии Python' )
    first = False
    while( True ):
        try: val = int( input( 'введите checkinterval : ' ) ); break;
```

```
        except ValueError: print( 'checkinterval должно быть целочисленным' )
        except KeyboardInterrupt: print(); sys.exit( 0 )
    sys.setcheckinterval( val )
    if not switch: continue
    while( True ):
        try: val = float( input( 'введите switchinterval : ' ) ); break;
        except ValueError: print( 'switchinterval должно быть вещественным' )
        except KeyboardInterrupt: print(); sys.exit( 0 )
    sys.setswitchinterval( val )
```

Начиная с версии Python 3.2 вызов `setcheckinterval()` был объявлен устаревшим (deprecated) и заменён вызовом `setswitchinterval()`. Если старый метод устанавливает интервал переключения как число виртуальных инструкций, то новая реализация определяет его в секундах, из-за чего изменяется модель поведения. Но в обоих случаях устанавливается только минимальная граница, на которой может произойти переключение, а реально момент переключения может наступить значительно позже, например, если выполняется продолжительная внутренняя инструкция или вызов метода.

Запустим данное приложение в разных версиях интерпретатора Python (программа сама диагностирует версию, а не совсем эстетичные обозначения в тесте, вида `*checkinterval` подчёркивают, что за ними скрыты по 2 библиотечных вызова, `set...()` и `get...()` соответственно):

Для версии 2 мы получим:

```
$ python idt.py
версия Python 2.7.3 на платформе linux2
интервал *checkinterval : 100
*switchinterval не реализованы в этой версии Python
введите checkinterval : 30
интервал *checkinterval : 30
введите checkinterval : 200
интервал *checkinterval : 200
введите checkinterval : ^C()
```

А для версии 3:

```
$ python3 idt.py
версия Python 3.2.3 на платформе linux2
интервал *checkinterval : 100
интервал *switchinterval : 0.005 сек.
*checkinterval объявлено устаревшими в этой версии Python
введите checkinterval : 30
введите switchinterval : .01
интервал *checkinterval : 30
интервал *switchinterval : 0.01 сек.
введите checkinterval : 77
введите switchinterval : .1
интервал *checkinterval : 77
интервал *switchinterval : 0.09999999999999999 сек.
введите checkinterval : ^C
```

В интерпретаторе Python используется глобальная блокировка интерпретатора (Global

Interpreter Lock — GIL), которая захватывается на время выполнения очередного потока, и никакой другой поток не может быть активирован, пока эта блокировка не будет освобождена. В этой модели нет ничего крамольного, но нужно хорошо представлять последствия, к которым может привести её применение и уметь пользоваться предоставляемыми возможностями. К детальному обсуждению этой модели мы вернёмся позже, уже после рассмотрения техники создания потоков в Python.

Потоки низкого уровня

Реализация низкого уровня представлена модулем `thread` стандартной библиотеки Python (в версии 3 этот модуль называется `_thread`), описание которого редко встречается в документации и литературе. Этот модуль обладает ограниченными возможностями для синхронизации, предлагая единственный примитив `LockType` (аналог бинарного мютекса), содержащий только методы для захвата (`acquire()`) и освобождения (`release()`). Тем не менее, даже на одном этом модуле можно построить развитое многопоточное приложение, как показано в листинге(файл `parallel/pytspeed/tlspeed.py`).

Запуск нескольких потоков и ожидание их завершения:

```
#!/usr/bin/python -O
# -*- coding: utf-8 -*-
import getopt
import sys
import time
try:
    import thread as thr
except ImportError:
    import _thread as thr

debuglevel = 0
threadnum = 2
delay = 1
active = 0
numt = 0                      # текущее число активных дочерних потоков
lock = thr.allocate_lock()    # блокировка доступа к числу активных дочерних потоков
wait = thr.allocate_lock()    # блокировка ожидания завершения всех дочерних потоков
barier = { 'numt' : numt, 'lock' : lock, 'wait' : wait }

def delay_in_cycle( delay = 1.0 ):
    t = time.time()
    while time.time() - t < delay :
        pass

def thrfun( delay, num, tstart ):
    st = time.time() - tstart
    barier[ 'numt' ] = barier[ 'numt' ] + 1
    barier[ 'lock' ].release()
    ss = '\t{} : {} <= старт: {:14.11f}'.format( num, id, st )
    if not active : time.sleep( delay )     # пауза
```

```
    else : delay_in_cycle( delay )          # или активное ожидание
    barier[ 'lock' ].acquire()
    barier[ 'numt' ] = barier[ 'numt' ] - 1
    st = time.time() - tstart               # время завершения потока
    print( '{} - финиш: {:14.11f}'.format( ss, st ) )
    if 0 == barier[ 'numt' ] :
        barier[ 'wait' ].release()
    barier[ 'lock' ].release()
    return

opts, args = getopt.getopt( sys.argv[1:], "vt:d:a" )
for opt, arg in opts: # опции (ключи) командной строки (-v, -t, -d, -a)
    if opt[ 1: ] == 'v' : debuglevel = debuglevel + 1
    if opt[ 1: ] == 't' : threadnum = int( arg )
    if opt[ 1: ] == 'd' : delay = int( arg )
    if opt[ 1: ] == 'a' : active = 1
if debuglevel > 0 :
    print( opts )
    print( args )
    print( debuglevel )
    print( threadnum )

barier[ 'wait' ].acquire()   # захват блокировки завершения
for n in range( threadnum ): # запуск threadnum потоков
    barier[ 'lock' ].acquire()
    id = thr.start_new_thread( thrfun, ( delay, n, time.time() ) )
    print( "\t{} : {} => запуск".format( n, id ) )
barier[ 'wait' ].acquire()  # ожидание завершения всех потоков
print( 'завершены все {} потоков \
        завершается ожидавший главный поток'.format( threadnum ) )
```

Это приложение успешно выполняется как в версии 2, так и в версии 3 (для этого пришлось усложнить процесс импортирования модуля `thread` или `_thread` в зависимости от версии):

```
$ python tlspeed.py  -t3 -d2
        0 : -1220105408 => запуск
        1 : -1229980864 => запуск
        2 : -1240466624 => запуск
        0 : -1220105408 <= старт:  0.00027799606 - финиш:  2.00136685371
        1 : -1229980864 <= старт:  0.00010585785 - финиш:  2.00126695633
        2 : -1240466624 <= старт:  0.00015902519 - финиш:  2.00125098228
завершены все 3 потоков,
завершается ожидавший главный поток
```

```
$ python3 tlspeed.py  -t3 -d2
        0 : -1220469952 => запуск
        1 : -1229980864 => запуск
```

```
        2 : -1240466624 => запуск
        0 : -1220469952 <= старт:  0.00017404556 - финиш:  2.00208616257
        1 : -1229980864 <= старт:  0.00010085106 - финиш:  2.00201487541
        2 : -1240466624 <= старт:  0.00019097328 - финиш:  2.00192403793
завершены все 3 потоков
завершается ожидавший главный поток
```

В этой модели поток создаётся и **сразу же запускается на выполнение** (подобно `pthread_t` в POSIX) вызовом `start_new_thread()`. В приложении запускается несколько потоков (число потоков — опция `-t`), которые выполняются некоторое время (число секунд — опция `-d`), а главный поток ожидает их завершения. Опцией `-a` операцией выполнения потоков можно сделать не пассивное ожидание, а выдержку на активных процессорных циклах.

За счёт бедности средств синхронизации, для того, чтобы дождаться окончания дочерних потоков, требуется создавать искусственные конструкции, типа контейнера (структуры) `barier` в программе, работающей по типу счётного семафора. В этом и заключается слабость низкоуровневого механизма потоков Python.

Потоки высокого уровня

Реализация высокого уровня — это модуль `threading` стандартной библиотеки Python, который чаще всего и имеют в виду, когда говорят о потоках в Python. Реализуемая в нём модель является надстройкой над рассмотренной выше, поэтому поведение потоков не будет заметно различаться. Но модуль `threading` предоставляет гораздо больше возможностей и примитивов синхронизации (`Lock`, `RLock`, `Condition`, `Semaphore`, `Event`, `Queue`). С помощью этого модуля функциональность, аналогичную представленной в листинге выше, можно реализовать гораздо проще, короче и понятнее, как и показано в листинге далее (файл `parallel/pytspeed/thspeed.py`)**.**

Запуск нескольких потоков и ожидание их завершения:

```
#!/usr/bin/python -O
# -*- coding: utf-8 -*-
import getopt
import sys
import threading
import time

debuglevel = 0
threadnum = 2
delay = 1
active = 0

def delay_in_cycle( delay = 1.0 ):
    t = time.time()
    while time.time() - t < delay :
        pass

def thrfun( *args ):
    st = time.time() - args[ 2 ]             # время старта потока
    ss = '\t{} : {} <= старт: {:14.11f}'. \
```

```
        format( args[ 1 ], threading.currentThread().getName(), st )
    if not active : time.sleep( args[ 0 ] )  # пауза
    else : delay_in_cycle( args[ 0 ] )       # или активное ожидание
    st = time.time() - args[ 2 ]             # время завершения потока
    print( '{} - финиш: {:14.11f}'.format( ss, st ) )
    return

opts, args = getopt.getopt( sys.argv[1:], "vt:d:a" )
for opt, arg in opts:     # опции (ключи) командной строки (-v, -t, -d, -a)
    if opt[ 1: ] == 'v' : debuglevel = debuglevel + 1
    if opt[ 1: ] == 't' : threadnum = int( arg )
    if opt[ 1: ] == 'd' : delay = int( arg )
    if opt[ 1: ] == 'a' : active = 1
if debuglevel > 0 :
    print( opts )
    print( args )
    print(  debuglevel )
    print(  threadnum )

threads = []
for n in range( threadnum ): # создание и запуск потоков
    parm = [ delay, n, 0 ]
    t = threading.Thread( target=thrfun, args=parm )
    threads.append( t )
    t.setDaemon( 1 )
    print( "\t{} : {} => запуск".format( n, t.getName() ) )
    parm[ 2 ] = time.time()
    t.start()
for n in range( threadnum ): # ожидание завершения всех потоков
    threads[ n ].join()
print( 'завершены все {} потоков \
        завершается ожидавший главный поток'.format( threadnum ) )
```

В этой модели объект потока создаётся вызовом конструктора класса `Thread`, но его запуск на выполнение отсрочен и производится отдельно вызовом метода `start()` этого объекта. В данном случае запуск потока, по сравнению с низкоуровневой моделью, упрощается, но передача параметров в потоковую функцию напротив усложняется. Как и в предыдущем случае, показанный код выполняется с одинаковыми результатами в обеих версиях Python:

```
$ python thspeed.py  -t3 -d2
        0 : Thread-1 => запуск
        1 : Thread-2 => запуск
        2 : Thread-3 => запуск
        0 : Thread-1 <= старт:  0.00039386749 - финиш:  2.00249791145
        1 : Thread-2 <= старт:  0.00040698051 - финиш:  2.00256109238
        2 : Thread-3 <= старт:  0.00021982193 - финиш:  2.00245380402
завершены все 3 потоков
```

```
завершается ожидавший главный поток
```

```
$ python3 thspeed.py  -t3 -d2
        0 : Thread-1 => запуск
        1 : Thread-2 => запуск
        2 : Thread-3 => запуск
        0 : Thread-1 <= старт:  0.00151085854 - финиш:  2.00366187096
        1 : Thread-2 <= старт:  0.00321602821 - финиш:  2.00535202026
        2 : Thread-3 <= старт:  0.00187206268 - финиш:  2.00381302834
завершены все 3 потоков
завершается ожидавший главный поток
```

Параллельные процессы

Аналогичного поведения можно добиться и с помощью параллельных процессов. Такие решения совершенно естественны для всех операционных системы семейства UNIX (совместимых с POSIX). Но всё гораздо хуже в операционных системах Windows … мы вернёмся к этому вскоре. В листинге представлен пример реализации, использующей механизмы операционной системы (файл `parallel/pytspeed/fork.py`).

Запуск нескольких дочерних процессов:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os
import time
import sys
import getopt

delay = 1
procnum = 2
debuglevel = 0

opts, args = getopt.getopt( sys.argv[1:], "p:d:v" )
for opt, arg in opts:     # опции (ключи) командной строки (-v, -t, -d, -a)
    if opt[ 1: ] == 'v' : debuglevel = debuglevel + 1
    if opt[ 1: ] == 't' : threadnum = int( arg )
    if opt[ 1: ] == 'd' : delay = int( arg )
    if opt[ 1: ] == 'a' : active = 1

childs = []
if debuglevel :
     print( 'родительский процесс {}'.format( os.getpid() ) )
for i in range( 0, procnum ) :
    tim = time.time();
    try :
        pid = os.fork();
    except :
```

```
        print( 'error: create child process' ); sys.exit( 33 )
    if pid == 0 :
        trun = time.time() - tim;
        if debuglevel :
           print( 'дочерний процесс {} - задержка на запуск: {:14.11f}'. \
                  format( os.getpid(), trun ) )
        time.sleep( delay )
        trun = time.time() - tim;
        if debuglevel :
           print( 'дочерний процесс {} - время завершения: {:14.11f}'. \
                  format( os.getpid(), trun ) )
        sys.exit( 3 )
    if pid > 0 :
        childs.append( pid )
        if debuglevel :
            print( '{}: создан новый дочерний процесс {}'.format( os.getpid(), pid ) )
print( 'ожидание завершения дочерних процессов ...' )
for p in childs :
    pid, status = os.wait()
    if debuglevel :
        print( 'код завершения процесса {} = {}'.\
                    format( pid, os.WEXITSTATUS( status ) ) )
print( 'все порождённые процессы успешно завершены' )
```

Это приложение также хорошо исполняется в обеих версиях Python:

```
$ python fork.py -v
родительский процесс 13612
13612: создан новый дочерний процесс 13613
13612: создан новый дочерний процесс 13614
ожидание завершения дочерних процессов ...
дочерний процесс 13614 - задержка на запуск:  0.00063419342
дочерний процесс 13613 - задержка на запуск:  0.00130796432
дочерний процесс 13614 - время завершения:  1.00162100792
дочерний процесс 13613 - время завершения:  1.00252485275
код завершения процесса 13613 = 3
код завершения процесса 13614 = 3
все порождённые процессы успешно завершены
```

```
$ python3 fork.py -v
родительский процесс 13647
13647: создан новый дочерний процесс 13648
дочерний процесс 13648 - задержка на запуск:  0.00055909157
13647: создан новый дочерний процесс 13649
ожидание завершения дочерних процессов ...
дочерний процесс 13649 - задержка на запуск:  0.00042104721
дочерний процесс 13648 - время завершения:  1.00205016136
```

```
дочерний процесс 13649 - время завершения:  1.00123310089
код завершения процесса 13648 = 3
код завершения процесса 13649 = 3
все порождённые процессы успешно завершены
```

В данном примере (как и в случаях с потоками) проставляются временные метки, обозначающие задержки при запуске дочерних параллельных ветвей. Как можно заметить, временные затраты на создание и запуск нового процесса или потока для исполняющей системы Python практически одинаковы. Это можно объяснить тем, что основное время съедает интерпретирующая система, а не выполнение системных вызовов POSIX. Этот эффект также встречается и при программировании на языке C: «тяжесть» параллельных процессов проявляется не в скорости их создания или переключения контекста в мультизадачной среде, а в сложности и медленности межпроцессных взаимодействий (IPC) в изолированных (защищённых) адресных пространствах.

Безусловно, не следует забывать и то, что число одновременно запущенных процессов в операционной системе может быть ограничено (это зависит от объёма ресурсов, доступных системе, главным образом, оперативной памяти):

```
$ python fork.py -p721
ожидание завершения дочерних процессов ...
все порождённые процессы успешно завершены
$ python fork.py -p722
error: create child process
$ echo $?
33
```

Мульти-платформенные многопоточные приложения

Решение, показанное в предыдущей статье и **использующее вызов** `fork()` из импортированного модуля `os`, обладает многими преимуществами, но и существенным недостатком. Этой возможностью нельзя воспользоваться в операционных системах семейства Microsoft Windows, так как в ОС Windows никогда не было системного вызова `fork()`, и разработчики модуля `os` для этого вызова оставили только заглушку. Но из-за этого ограничения теряется одно из главных достоинств Python — переносимость проектов и их независимость от платформы.

Поэтому создатели Python предоставляют для реализации параллельных процессов другой модуль – `multiprocessing`, использующий особые приёмы для клонирования процессов, обсуждению которого, в частности, и будет посвящена данная статья.

Мульти-платформенный код

В документации говорится, что из-за отсутствия вызова `fork()` в ОС Windows он эмулируется путём создания нового процесса для исполнения кода, который в ОС Linux выполнялся бы в дочернем процессе. Так как исполняемый код технически не связан с процессом, то он должен быть помещён в процесс перед запуском. Для этого код форматируется и передаётся по каналу из оригинального процесса во вновь созданный. Также новый процесс получает инструкцию запустить код, полученный по каналу, через переданный аргумент командной строки `--multiprocessing-fork`. Если посмотреть на реализацию метода `freeze_support()`, то его задачей является проверка того, должен ли исполняемый процесс запускать код, полученный по каналу, или нет.

При подобном подходе запуск параллельных процессов может производиться, как показано в листинге ниже (файл `parallel/multip/child.py`)

Мульти-платформенный код для запуска нескольких дочерних процессов:

```
#!/usr/bin/python3 -O
# -*- coding: utf-8 -*-
import time
import sys
import os
from multiprocessing import Process, freeze_support

def info( title ):
    if hasattr( os, 'getppid' ):  # only available on Unix
        print( '{0}:\tPID={1} PPID={2}'.format( title, os.getpid(), os.getppid() ) )
    else:
        print( '{0}:\tPID={1}'.format( title, os.getpid() ) )

def fun( name ):
    info( 'порождённый процесс' )
       print(  'процесс  {0}  выполняет  функцию  с  параметром  {1}'.format(  os.getpid(),
name ) )
    time.sleep( 0.5 )

if __name__ == '__main__':
    freeze_support()
    nproc = len( sys.argv ) > 1 and int( sys.argv[ 1 ] ) or 3
    print( 'число дочерних процессов ', nproc )
    info( 'родительский процесс' )
    procs = []
    for i in range( nproc ):
        procs.append( Process( target = fun, args = ( i, ) ) )
    for i in range( nproc ):
        procs[ i ].start()
    for i in range( nproc ):
        procs[ i ].join()
    print( 'завершается родительский процесс' )
```

Как уже объяснялось выше, в Windows в качестве кода процесса используется уже ранее компилированный байт-код приложения, поэтому использование конструкции: `if __name__ == '__main__'` - становится **обязательным**! Без этого фрагмента код порождённого дочернего процесса начнёт снова выполнять код главной ветви приложения, что породит бесконечную рекурсию из-за «размножения» процессов. Использование этой конструкции в операционных системах, реализующих вызов `fork()` не обязательно, но приветствуется, так как такой код становится независимым от платформы исполнения.

Попробуем запустить данное приложение в ОС Windows:

```
$ python child.py
число дочерних процессов  3
родительский процесс:     PID=14562 PPID=2089
порождённый процесс:      PID=14563 PPID=14562
процесс 14563 выполняет функцию с параметром 0
порождённый процесс:      PID=14564 PPID=14562
```

```
процесс 14564 выполняет функцию с параметром 1
порождённый процесс:      PID=14565 PPID=14562
процесс 14565 выполняет функцию с параметром 2
завершается родительский процесс
```

Новый процесс создаётся конструктором класса `Process()`, а целевым кодом для него указывается функция (`target=...`), как это имеет место при создании потока, после чего процесс должен быть запущен вызовом метода `start()`.

Модуль `multiprocessing` предоставляет различные механизмы для взаимодействия созданных процессов, например:

• механизмы взаимодействия IPC: `Queue`, `Pipe`;

• механизмы взаимодействия через разделяемую процессами память `Value`, `Array`;

• специфичные механизмы, такие как `Manager` и `Pool` — пул потоков;

Примеры кода, использующие эти механизмы, включены в архив `parallel/multip`, файлы: `ipc.py`, `mgr.py`, `pool.py`, но мы не будем подробно разбирать их — это всё успешно работающие примеры и там же приложен файл журнала выполнений `multip.hist` — этого достаточно чтобы разобраться в деталях**.**

Многопроцессорное выполнение

Параллельные ветви исполнения (потоков или процессов) в коде программы могут применяться для различных целей:

- квазипараллельный код (попеременно переключающийся с одной ветви на другую) в прикладных системах, где логика системы описывается естественным образом в терминах параллелизма (например, это задачи «производитель-потребитель»);
- параллельное совмещение ветвей кода, имеющих различный характер загрузки процессора: активный ввод-вывод в сочетании с большой вычислительной нагрузкой;
- распараллеливание процессорной нагрузки между несколькими процессорами в многопроцессорных SMP системах.

Ещё не так давно последняя категория приложений была скорее экзотикой, чем практикой. Но за это время произошло массовое внедрение многоядерных процессоров и процессоров с **гиперпоточностью** (hyper-threading), и сегодня рядовой офисный компьютер, с большой вероятностью, является многопроцессорным.

В листинге ниже представлен простой пример для динамического определения числа процессоров в системе (файл `parallel/mthrs/num_proc.py`).

Диагностика числа процессоров:

```
#!/usr/bin/python -O
# -*- coding: utf-8 -*-
from multiprocessing import cpu_count
print( 'число процессоров = {}'.format( cpu_count() ) )
```

Пример запуска этого сценария:

```
$ python num_proc.py
число процессоров = 8
```

Примечание: Это, кстати, показан результат для процессора i7, который имеет только 4 процессора (ядра) но с гипертриэдингом:

```
$ lscpu
Архитектура:x86_64
```

```
CPU op-mode(s):        32-bit, 64-bit
Порядок байт:Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
...
Имя модели:   Intel(R) Core(TM) i7 CPU       Q 720  @ 1.60GHz
```

Известно, что модель потоков, принятая в Python, **непригодна** к многопроцессорному выполнению. Это связано с блокировкой GIL, с которой мы уже встречались, а наиболее детально этот вопрос анализируется в известной статье Дэвида Бизли (см. ссылки в конце текста):

Принцип работы прост. Потоки удерживают GIL, пока выполняются. Однако они освобождают его при блокировании для операций ввода-вывода. Каждый раз, когда поток вынужден ждать, другие, готовые к выполнению, потоки используют свой шанс запуститься.

...

При работе с CPU-зависимыми потоками, которые никогда не производят операции ввода-вывода, интерпретатор периодически проводит проверку. Интервал проверки — глобальный счетчик, абсолютно независимый от порядка переключения потоков.

...

Ожидающий поток при этом может сделать сотни безуспешных попыток захватить GIL.

Мы видим, что происходит битва за две взаимоисключающие цели. Python просто хочет запускать не больше одного потока в один момент. А операционная система щедро переключает потоки, пытаясь извлечь максимальную выгоду из всех ядер.

Подтверждение этого утверждения и его последствия можно увидеть на примере следующего листинга (файл `parallel/mthrs/mthrs.py`).

Сравнение способов параллельного исполнения:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import time
import sys
import getopt
import threading
import os
import multiprocessing

def ncount( n ) : # тестовая CPU-загружающая функция
    while n > 0 : n -= 1

if __name__ == '__main__':
    repnum = 10000000
    thrnum = 2
    mode = 'stpm' # варианты запуска
```

```
try :
    opts, args = getopt.getopt( sys.argv[1:], "t:n:m:" )
except getopt.GetoptError :
    print ( "недопустимая опция команды или её значение" )

for opt, arg in opts :
    if opt[ 1: ] == 't' : thrnum = int( arg )
    if opt[ 1: ] == 'n' : repnum = int( arg )
    if opt[ 1: ] == 'm' : mode = arg

print( "число процессоров (ядер) = {0:d}".format( multiprocessing.cpu_count() ) )
print( "исполнение в Python версия {0:s}".format( sys.version ) )
print( "число ветвей выполнения {0:d}".format( thrnum ) )
print( "число циклов в ветви {0:d}".format( repnum ) )

if 's' in mode :
    print( "============ последовательное выполнение ============" )
    clc = time.time()
    for i in range( thrnum ) : ncount( repnum )
    clc = time.time() - clc
    print( "время {0:.2f} секунд".format( clc ) )

if 't' in mode :
    print( "================ параллельные потоки ================" )
    threads = []
    for n in range( thrnum ) :
        tid = threading.Thread( target = ncount, args=( repnum, ) )
        threads.append( tid )
        tid.setDaemon( 1 )
    clc = time.time()
    for n in range( thrnum ) : threads[ n ].start()
    for n in range( thrnum ) : threads[ n ].join()
    clc = time.time() - clc
    print( "время {0:.2f} секунд".format( clc ) )

if 'p' in mode :
    print( "================ параллельные процессы ================" )
    threads = []; fork = True
    clc = time.time()
    for n in range( thrnum ) :
        try : pid = os.fork();
        except :
            print( "ошибка создания дочернего процесса" )
            fork = False
```

```
                break
            else :
                if pid == 0 : # дочерний процесс
                    ncount( repnum )
                    sys.exit( 0 )
                if pid > 0 :  # родительский процесс
                    threads.append( pid )
        if fork :
            for p in threads :
                pid, status = os.wait()
            clc = time.time() - clc
            print( "время {0:.2f} секунд".format( clc ) )

    if 'm' in mode :
        print( "=============== модуль multiprocessing ==============" )
        parms = []
        for n in range( thrnum ) :
            parms.append( repnum )
        multiprocessing.freeze_support()
        pool = multiprocessing.Pool( processes = thrnum, )
        clc = time.time()
        pool.map( ncount, parms )
        clc = time.time() - clc
        print( "время {0:.2f} секунд".format( clc ) )
```

Приложение тестирует время выполнения большого числа (опция `-n`) циклов простого декремента целочисленной переменной, выполняемого в несколько (опция `-t`) параллельных ветвей исполнения для 4-х вариантов выполнения этой нагрузки:

- без ветвления, весь объём работы выполняется последовательно;
- работа распределяется на `N` **потоков**;
- работа распределяется на `N` **процессов**, разветвлённых `fork()`;
- работа распределяется на `N` **процессов**, разветвлённых с помощью API модуля `multiprocessing`;

Поскольку тестирование может занять весьма продолжительное время, то опцией запуска `-m` можно указать только тот режим тестирования из 4-х, который следует выполнять (соответственно, значения для `-m` будут `'s', 't', 'p', 'm'`). Для запуска приложения в определённых режимах используется подобная команда:

```
$ python mthrs.py -n 5000000 -t 4 -m tm
...
```

Приложение единообразно выполняется как в Linux, так и в Windows, и под версиями Python 2 и 3. Теперь можно проанализировать все возможные варианты исполнения:

На платформе Linux для Python 2 мы получим:

```
$ python mthrs.py
число процессоров (ядер) = 2
исполнение в Python версия 2.7.3 (default, Jul 24 2012, 10:05:39)
[GCC 4.7.0 20120507 (Red Hat 4.7.0-5)]
```

```
число ветвей выполнения 2
число циклов в ветви 10000000
============ последовательное выполнение ============
время 2.89 секунд
================ параллельные потоки ================
время 3.55 секунд
================ параллельные процессы ================
время 1.78 секунд
================ модуль multiprocessing ================
время 1.75 секунд
```

Вот главный результат, из-за которого была написана статья Дэвида Бизли и который может привести в недоумение: выполнение нагрузки на 2-х процессорах в 2 **потока** в Python требует на 23% **больше** времени, чем, если ту же нагрузку просто выполнить последовательно, вообще не создавая никаких потоков! А время выполнения для 2-х независимых **процессов** составляет только 60%.

Также на платформе Linux, но уже для Python 3 мы получим:

```
$ python3 mthrs.py
число процессоров (ядер) = 2
исполнение в Python версия 3.2.3 (default, Jun  8 2012, 05:37:15)
[GCC 4.7.0 20120507 (Red Hat 4.7.0-5)]
число ветвей выполнения 2
число циклов в ветви 10000000
============ последовательное выполнение ============
время 6.57 секунд
================ параллельные потоки ================
время 9.74 секунд
================ параллельные процессы ================
время 3.93 секунд
================ модуль multiprocessing ================
время 3.66 секунд
```

В данном сценарии картина становится ещё радикальнее: при использовании потоков замедление увеличивается до 48%, а для параллельных процессов время исполнения сокращается до 55% от случая последовательного исполнения. Отметим также, что несмотря на исполнение кода на той же системе, что и в предыдущем случае, общее время выполнения того же объёма работы увеличилось более чем в 2 раза, по сравнению с Python 2. За гибкость новых синтаксических возможностей языка приходится расплачиваться увеличением времени, затрачиваемого интерпретатором на исполнение!

Перейдём к Windows XP и Python 2 (выполнение в Python-терминале IDLE):

```
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
число процессоров (ядер) = 2
исполнение в Python версия 2.7.5 (default, May 15 2013, 22:43:36) ...
число ветвей выполнения 2
число циклов в ветви 10000000
```

```
============ последовательное выполнение ============
время 1.19 секунд
================ параллельные потоки ================
время 14.05 секунд
================ параллельные процессы ================
ошибка создания дочернего процесса
================ модуль multiprocessing ==============
время 0.72 секунд
>>>
```

Картина становится ещё хуже, так как выполнение в 2 **потока** занимает в 11.8 раз больше времени по сравнению с простым последовательным выполнением. Но модуль `multiprocessing` при использовании параллельных **процессов**, создаваемых под Windows, обеспечивает прирост производительности на те же 60%

Запуск в Windows XP для Python 3 (код выполняется на той же системе, но на этот раз уже Windows XP работает в виртуальной машине VirtualBox 4.2.6, приложение выполняется в IDLE):

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) ...
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
число процессоров (ядер) = 2
исполнение в Python версия 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) ...
число ветвей выполнения 2
число циклов в ветви 10000000
============ последовательное выполнение ============
время 2.30 секунд
================ параллельные потоки ================
время 2.33 секунд
================ параллельные процессы ================
ошибка создания дочернего процесса
================ модуль multiprocessing ==============
время 1.31 секунд
>>>
```

Похоже, что в версии Python 3.3 (в Linux сценарий выполнялся в Python 3.2) произошли значительные улучшения в области управления потоками, так при использовании 2-х потоков наблюдалось замедление всего на 1.5%.

Отметим также скорость выполнения Python-приложений в Windows под VirtualBox. Можно сказать, что скорость исполнения кода в VirtualBox практически не уступает «родной» среде.

В конце запустим наш сценарий также на ОС Linux, но на достаточно старом дистрибутиве Ubuntu 10.4 и таком процессоре:

```
$ cat /proc/cpuinfo | grep 'model name'
model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz
model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz
model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz
model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz
```

Но, как известно, процессоров Atom с 4-мя ядрами не бывает, это 2 достаточно медленных

ядра с hyper-threading. Выполняем приложение на подобной конфигурации:

```
$ python mthrs.py -t4
число процессоров (ядер) = 4
исполнение в Python версия 2.6.5 (r265:79063, Oct  1 2012, 22:07:21) [GCC 4.4.3]
число ветвей выполнения 4
число циклов в ветви 10000000
============ последовательное выполнение ============
время 12.90 секунд
================ параллельные потоки ================
время 19.14 секунд
=============== параллельные процессы ===============
время 4.59 секунд
=============== модуль multiprocessing ==============
время 4.58 секунд
```

В данном случае мы наблюдаем уже знакомую картину: выполнение в 4 потока только замедляет работу (+48%), но вот выполнение в 4 процесса ускоряет её почти в 3 раза. Попробуем сделать выводы из полученных результатов.

Потоки или процессы?

Так, использование потоков Python в многопроцессорных конфигурациях просто не имеет смысла. Если не учесть это обстоятельство, то можно реализовать проект как многопоточный, рассчитывая ускорить выполнение, а в итоге замедлить его, и, возможно, очень существенно.
Означает ли это, что использование потоков Python нецелесообразно вообще? Нет, не означает. Потоки будут уместны для сценариев, когда параллельные ветви (или некоторые из них) достаточно часто переходят в блокированные состояния, например, ожидая результатов операций ввода-вывода.
В многопроцессорной конфигурации целесообразно будет использовать параллельные процессы. В этом случае в каждом процессе будет выполняться отдельная копия интерпретатора Python, и это может дать существенный выигрыш в производительности.
Параллельные процессы предпочтительнее создавать не средствами операционной системы (вызов `fork()`), а используя API модуля `multiprocessing` из стандартной библиотеки Python.
Означает ли это, что параллельные процессы всегда предпочтительнее потоков, и обладают такой же «лёгкостью» (в смысле скорости) как и потоки? Нет, не означает. Медлительность параллельных процессов будет проявляться в их взаимодействиях между собой, будь это использование механизмов IPC, или использование разделяемой (shared) памяти. Оба эти способа требуют значительных процессорных ресурсов, поскольку для обмена информацией приходится каждый раз преодолевать границы защищённых адресных пространств процессов, что невозможно без вмешательства кода супервизорного режима (ядра операционной системы) и вовлечения MMU (устройства управления памятью).
Данные выводы не зависят от используемой операционной системы. Хотя в примерах были показаны результаты для ОС Linux и Windows, но такие же эффекты наблюдаются и в MacOS. Численные значения могут увеличиваться / уменьшаться, но общие принципы останутся неизменными.

Интеграция Python и C кодов

Среда Python является самодостаточной языковой системой и технологией для разработки программных проектов. Механизм для расширения функциональных возможностей языка реализуется через технику модулей и пакетов, являющуюся фундаментальным компонентом архитектуры Python.

Тем не менее, нужно отметить, что многие из модулей, известные в Python как стандартные, на самом деле реализованы на языке C. Поэтому возникает интересная возможность реализовывать собственные целевые модули Python на C или C++. Хотя мотивы для подобного решения могут быть самыми разными, но стоит перечислить несколько ключевых:

- эффективность реализации ресурсоёмких алгоритмов и вычислительных процедур на C, так как за счёт компилирующей природы этого языка скорость выполнения таких реализаций может быть в 40 (а в отдельных случаях в 100 и более) раз быстрее, чем на Python;
- предоставить Python-приложениям доступ к различным аппаратно или системно-зависимым сущностям (аудио, аппаратно-зависимые счётчики, периферийные устройства … стандартные или специализированные);
- простой способ реализовать интерфейс к множеству уже существующих целевых библиотек из открытых проектов на C, самых разных областей применения, для использования их из Python-приложений;

Далеко не последнюю роль в привлекательности таких смешанных языковых реализаций играет и **простота** построения интерфейса из C в Python, как будет показано ниже. А это не такая простая задача: объединить в единый гибрид код, интерпретируемый внутри виртуальной машины Python, и «чистый» машинный код, получающийся как продукт компиляции кода C. Эдакий Тяни-Толкай ...

Альтернативы

Существуют (разрабатываются или находятся в разной степени готовности к эксплуатации) несколько альтернативных способов реализации интерфейса из C в Python. Каждый из них имеет свои особенности, преимущества и недостатки, связанные с конкретными требованиями к ситуации. Ниже мы перечислим только некоторые из них, которые будут подробно рассмотрены в этой и последующей статьях:

- модуль `ctypes` из стандартной поставки Python (например, `/usr/lib/python2.7/ctypes/*` - версия 2.7 здесь и далее показана совершенно условно, у вас она может отличаться);
- ручное написание интерфейса модуля;
- пакет `distutils` из стандартной библиотеки Python;
- библиотека `Boost.Python` из проекта `Boost`;
- п`роект Cython`;
- п`роект SWIG`;

В случае применения любого инструмента, конечный результат будет выглядеть как **динамически связываемая библиотека** (DLL — `.so` в Linux, или `.dll` в Windows), функции которой можно будет вызывать из кода Python.

Модуль `ctypes`

Модуль `ctypes` присутствует в стандартной поставке Python. Это **самый простой** способ с точки зрения написания кода и сборки, не требующий никаких дополнительных инструментов. Но это и самый опасный способ, так как здесь на стыке Python/C отсутствует не только какой-либо контроль правильности соответствия **типов** ожидаемых параметров, но даже просто элементарный контроль их **количества**!

Для примера создадим несколько произвольных C-функций с самыми разнообразными наборами предполагаемых параметров и возвращаемых значений (см. файл `c_interaction/ctypes/call/call.c`):

Тестовые функции C:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

// c_double fun1( c_short )
double fun1( short i ) {
   printf( "получен параметр = %d\n", i );
   return (double)i;
}

// c_void_p fun2( c_char_p )
const char* fun2( const char* s ) {
   printf( "локализация операционной системы: %s\n",
           setlocale( LC_ALL, "" ) );      // по умолчанию, из установок системы
   int nsym = 0;
   const char *p = s;
   while( 1 ) {
      int n = mblen( p, MB_CUR_MAX );
      if( 0 == n ) break;
      nsym++;
      p += n;
   };
   printf( "получен параметр [ mbchar* ] :
           \"%s\"; длина: в байтах = %d, в символах = %d\n", s, strlen( s ), nsym );
   static const char* ret = "возвращаемая строка";
   return ret;
}

// c_void_p fun3( c_float, c_long, c_char_p )
void* fun3( float f, long l, char* s ) {
   printf( "вызов функции 3-х переменных: \"float\"=%f, \"long\"=%ld,
            \"char*\"=\"%s\"\n", f, l, s );
   return (void*)s;
}
```

Выполним сборку динамической библиотеки следующей командой:

```
$ gcc -shared -o call.so -fPIC call.c
```

Для демонстрации вызовов подготовленных на C функций из кода Python подготовим тестовую программу, представленную листинг (см. файл `c_interaction/test.py`):

Тестирование вызовов C-функций:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

```
import ctypes

lib = ctypes.CDLL( './call.so' )

print '-----------------------------------------------------'
# === double fun1( short ) ===
# pfun = ctypes.CDLL( './call.so' ).fun1
pfun = lib.fun1
pfun.restype = ctypes.c_double
pfun.argtypes = ( ctypes.c_short, )
x = pfun( 3 )
print 'возвращено значение %f' % x
print '-----------------------------------------------------'

# === const char* fun2( const char* ) ===
#pfun = ctypes.CDLL( './call.so' ).fun2
pfun = lib.fun2
pfun.restype = ctypes.c_char_p
pfun.argtypes = ( ctypes.c_char_p, )
x = pfun( "строка вызова" )
print 'возвращено значение : "%s"' % x
print '-----------------------------------------------------'

# === void* fun3( float, long, char* ) ===
pfun = lib.fun3
pfun.restype = ctypes.c_void_p
pfun.argtypes = ( ctypes.c_float, ctypes.c_long, ctypes.c_char_p )
s = "строка"
x = pfun( 3.1415, -12345, s )
print 'возвращено значение\t= %x' % x
print 'id( строка-параметр )\t= %x' % id( s )
print '-----------------------------------------------------'
```

Примечание. В примере из листинга используется синтаксис Python 2, так как в Python 3, где строки представляются **исключительно** в кодировке UTF-8, следует использовать типы `wchar_t` в C коде, и, соответственно, `c_wchar_p` в Python коде. Декодирование строк для версий 2 и 3 выполняется различными способами. Кроме того, в Python 3 присутствует более строгий контроль соответствия типов параметров, передаваемых функциям.

Проверим работоспособность созданного межязыкового интерфейса:

```
$ python test.py
-----------------------------------------------------
получен параметр = 3
возвращено значение 3.000000
-----------------------------------------------------
локализация операционной системы: ru_RU.UTF-8
получен параметр [ mbchar* ] : "строка вызова"; длина: в байтах = 25, в символах = 13
```

```
возвращено значение : "возвращаемая строка"
-----------------------------------------------------
вызов функции 3-х переменных: "float"=3,141500, "long"=-12345, "char*"="строка"
возвращено значение     = b76b15ac
id( строка-параметр )   = b76b1598
-----------------------------------------------------
```

В листинге `call.c` выше был показан условный пример, демонстрирующий работу с самыми разными вариантами параметров и возвращаемых значений. Теперь мы попробуем решить более реалистичную задачу по считыванию 64-х разрядного счётчика периодов частоты процессора, реализованного, начиная с процессора Pentium-II, и возвращаемого ассемблерной командой `RDTSC`. Такая функция позволит замерять временные интервалы в ходе выполнения программ с **наносекундным** разрешением. В листинге приведён пример такой функции на языке C (см. файл `c_interaction/ctypes/call/rdtsc.c`).

Функция `rdtsc()`:

```
unsigned long long rdtsc( void ) {          // ctypes.c_ulonglong rdtsc( void )
   unsigned long long int x;
   asm volatile ( "rdtsc" : "=A" (x) );
   return x;
}
```

Примечание: Реализация функции `rdtsc()` использует такое расширение компилятора GCC как инлайновые ассемблерные вставки. Для поклонников ассемблера здесь демонстрируется, как ассемблерные реализации становятся **непереносимыми** даже при минимальной смене деталей архитектуры (32 или 64 бит), или версии компилятора! Показанная выше реализация пригодна только для архитектуры i686. Для архитектуры X86_64 это должно выглядеть, например, так:

```
#include <stdint.h>
uint64_t rdtsc( void ) {
   union sc {
      struct { uint32_t lo, hi; } r;
      uint64_t f;
   } sc;
   asm volatile( "rdtsc" : "=a"( sc.r.lo ), "=d"( sc.r.hi ) );
   return sc.f;
}
```

А для более старых версий компилятора GCC, который «не знает» такой ассемблерной команды как `rdtsc`, это нужно переписать:

```
unsigned long long rdtsc( void ) {
   unsigned long long int x;
// вариант для старых версий GCC – просто указываем численный код команды:
   asm volatile ( ".byte 0x0f, 0x31" : "=A" (x) );
   return x;
}
```

Соберём библиотеку `rdtsc.so` с помощью следующей команды:

```
$ gcc -shared -o rdtsc.so -fPIC rdtsc.c
```

Далее в листинге представлено тестовое приложение, использующее из кода Python

функцию, реализованную на C (см. файл `c_interaction/ctypes/call/rdtest.py`):

Тестовое приложение для `rdtsc()`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import ctypes
lib = ctypes.CDLL( './rdtsc.so' )
pfun = lib.rdtsc
pfun.argtypes = () # ctypes.c_short, )
pfun.restype = ctypes.c_ulonglong

for i in 1, 2, 3 :
    print( 'RDTSC = {}'.format( pfun() ) )
```

Выполнив это приложение, мы получим **число периодов частоты процессора** с момента последней перезагрузки:

```
$ python rdtest.py
RDTSC = 81688400845550
RDTSC = 81688400995990
RDTSC = 81688401103470
```

Этот же код можно исполнить и в Python 3:

```
$ python3 rdtest.py
RDTSC = 81694859707300
RDTSC = 81694859845670
RDTSC = 81694859918650
```

Отдельно остановимся на важнейшей особенности этого способа (`ctypes`), так как мы не создаём на языке C модуля для последующего экспорта (как это будет делаться во всех других способах), а, напротив, используем искусственный интерфейс к функциям C из языка Python.

Даже из простых представленных примеров видно, что, при использовании `ctypes`, код для вызова C-функций из Python становится громоздким и малопонятным. Тем не менее, это **простейший** способ для взаимодействия с C-библиотеками из Python-приложений (и очень часто используемый на практике). Отличительной особенностью `ctypes` является простота реализации вызовов из уже **существующих** разделяемых системных библиотек, особенно когда требуется обеспечить вызов всего одной или немногих функций из такой библиотеки. Например, вот как могут выглядеть обращения к основным стандартным библиотекам в разных операционных системах (мы не будем обсуждать особенности правил выполнения вызовов, которые подробно описаны в документации, перечисленной в библиографии в конце текста):

- для ОС Windows (расширение .dll будет добавлено автоматически):

```
from ctypes import *
libs = windll.kernel32 // функции системной библиотеки kernel32.dll
                       // используют соглашение о связях stdcall
print( libs.GetModuleHandleA(None) )
...
libc = cdll.msvcrt     // функции стандартной библиотеки msvcrt.dll
                       // используют соглашение о связях cdecl.dll
```

```
print( libc.time( None ) )
```

- для ОС Linux:

```
from ctypes import *
libc = ctypes.CDLL( 'libc.so.6' )
print( libc.time( None ) )
```

Здесь же проявляется ещё одна неприятная особенность использования `ctypes` — то, что синтаксис записи связующего кода различается для разных операционных систем (что, вообще то говоря, не свойственно Python), а это сильно затрудняет создание мульти-платформенного кода.

Создание интерфейса модуля

Это второй из названных ранее альтернативных способов использования C-кода из Python. Если в предыдущем случае мы не создавали отдельный модуль (а непосредственно использовали библиотеку), то теперь мы сможем его создать, полностью прописав на языке C создание интерфейса к такому модулю. Наш модуль будет также реализовывать всё ту же функцию `rdtsc()`, опрашивающую счётчик тактов процессора. Используем тот же файл `rdtsc.c` с реализацией функции, но дополним его соответствующим файлом определений (хедер-файл) с общими определениями `rdtsc.h`, который необходим только для организации взаимодействия.

```
extern unsigned long long rdtsc( void );
```

Теперь можно вручную прописать интерфейс для нашего будущего Python-модуля `rdtsc` в виде файла `rdtsc_wrap.c` (`c_interaction/custom/rdtsc_wrap.c`):

Интерфейс модуля `rdtsc`:

```
#include <Python.h>
#include "rdtsc.h"

PyObject* rdtsc_wrap( PyObject* self, PyObject* args ) {
   if( self != NULL ) return NULL;
   return Py_BuildValue( "L", rdtsc() );
}

static PyMethodDef rdtscmethods[] = {      // таблица описания методов модуля
   { "rdtsc", rdtsc_wrap, METH_NOARGS },
   { NULL, NULL }
}

void initrdtsc() {                         // функция инициализации модуля
   Py_InitModule( "rdtsc", rdtscmethods );
}
```

В коде интерфейса должны присутствовать два обязательных компонента:

- таблица описания всех методов модуля;
- функция инициализации модуля (которая практически всегда одинакова, за исключением своего имени);

Правило для `Makefile`, ответственное за сборку DLL библиотеки, представляющей созданный Python-модуль будет выглядеть так как показано ниже:

```
        gcc -c -fpic rdtsc_wrap.c rdtsc.c -I/usr/include/python2.7
```

```
        ld -shared rdtsc_wrap.o rdtsc.o -lc -o rdtscmodule.so
```

Теперь можно будет осуществить вызов функции `rdtsc()` из кода приложения на Python, представленного в листинге (файл `c_interaction/custom/ptest.py`).

Вызов C-функции с помощью интерфейсного модуля:

```
#!/usr/bin/python -O
# -*- coding: utf-8 -*-
from rdtsc import rdtsc
from calibr import calibr

counter = []
for i in range( 5 ):
    counter.append( rdtsc() )
print "счётчик процессорных циклов:\n", counter

arg = [ 0, 10, 100, 1000, 10000, 100000 ]
msg = "калибровка последовательных вызовов:"
for i in arg:
    s = " %s(%i)" % ( str( calibr( i ) ), i )
    msg = msg + s
print msg
```

Попутно в модуле `calibr` была реализована функция `calibr()`, обычно нужная при хронометрировании очень коротких интервалов в программе. Пример этой функции показан ниже (файл `c_interaction/custom/calibr.py`).

Функция калибровки:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from rdtsc import rdtsc

def calibr( args = 10 ):
    sum = 0L
    if int( args ) <= 0: n = 10
    else: n = int( args )
    m = n
    while n > 0 :
        cf = -( rdtsc() - rdtsc() )
        sum = sum + cf
        n = n - 1
    return sum / m
...
if __name__ == "__main__":
    arg = [ 2, 5, 10, 20, 50, 100 ]
    msg = "калибровка последовательных вызовов:"
    for i in arg:
        s = " %s(%i)" % ( str( calibr( i ) ), i )
        msg = msg + s
```

```
    print msg
```

Эта функция определяет разность значений, полученных из двух последовательных вызовов `rdtsc()`, непосредственно следующих друг за другом, и показывает временные затраты (в тактах частоты процессора) на выполнение одиночного вызова самой функции `rdtsc()`.

Чтобы исключить дисперсию из-за загруженности процессора в многозадачной системе это значение усредняется по большому числу повторных измерений (на различных процессорах, при усреднении по 10-30 и более замеров результат можно считать уже стабильным).

Теперь можно проверить сценарий:

```
$ python -O ptest.py
счётчик процессорных циклов:
[32794250939400L, 32794250943940L, 32794250945180L, 32794250946280L, 32794250947320L]
калибровка последовательных вызовов: 520(0) 478(10) 445(100) 441(1000) ... 422(100000)
```

Для сравнения воспользуемся аналогичной тестовой программой, но написанной на языке C (файл `c_interaction/custom/ctest.c`).

Эталонная программа с аналогичной функциональностью, написанная на C:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "rdtsc.h"

#define NUMB 10
static unsigned calibr( int rep ) {
   uint32_t n, m, sum = 0;
   n = m = ( rep <= 0 ? NUMB : rep );
   while( n-- ) {
      uint64_t cf, cs;
      cf = rdtsc();
      cs = rdtsc();
      sum += (uint32_t)( cs - cf );
   }
   return sum / m;
}

int main( int argc, char **argv, char **envp ) {
   printf( "число процессорных тактов = %llu\n", rdtsc() );
   printf( "калибровка последовательных вызовов:" );
   printf( " %lu(0)", calibr( 0 ) );
   int n;
   for( n = 10; n <= 100000; n*=10 )
      printf( " %lu(%d)", calibr( n ), n );
   printf( "\n");
   exit( EXIT_SUCCESS );
}
```

Проверим работу этой реализации (тесты выполнялись последовательно друг за другом,

поэтому значения счётчика процессорных тактов имеют сходный порядок):

```
$ ./ctest
число процессорных тактов = 32794140948560
калибровка последовательных вызовов: 193(0) 191(10) 191(100) 191(1000) ... 142(100000)
```

Имея два различных варианта реализации (на Python и на C), мы можем сравнить время выполнения функции `rdtsc()` в Python – порядка 440 тактов с реализацией на C – около 190 тактов (все абсолютные цифры этой главы, естественно, радикально зависят от испольуемого процессора). Разница всего в 2.3 раза — это очень неплохой результат для интерпретирующего языка.

Пакет `distutils`

Следующим в рассмотрении инструментов должен стать пакет `distutils`, входящий в состав **стандартной** библиотеки Python. Этот пакет является вообще важным компонентом инфраструктуры Python, так как служит стандартным инструментом для распространения (дистрибуции) собственных пакетов Python. Но с его помощью можно наладить и взаимодействие между C и Python кодом, как будет показано в данной статье.

Для демонстрации `distutils` мы воспользуемся примером, который будет похож на модули из предыдущей статьи, правда исходный код модуля будет значительно упрощен для простоты изложения. В листинге представлен исходный модуль на языке C (файл `c_interaction/distutils/ownmod.c`).

Код модуля (язык C):

```
#include <Python.h>


static PyObject* py_echo( PyObject* self, PyObject* args ) {
   printf( "вывод из экспортированного кода!\n" );
   return Py_None;
}


static PyMethodDef ownmod_methods[] = {
   { "echo", py_echo, METH_NOARGS, "echo function" },
   { NULL, NULL }
};


PyMODINIT_FUNC initownmod() {
   (void)Py_InitModule( "ownmod", ownmod_methods );
}
```

Алгоритм работы `distutils` определяется небольшим конфигурационным файлом `setup.py` (который сам пишется на Python), представленным в листинге ниже.

Конфигурационный файл:

```
from distutils.core import setup, Extension


module1 = Extension( 'ownmod', sources = ['ownmod.c'] )


setup( name = 'ownmod',
       version = '1.1',
       description = 'This is a first package',
```

```
      ext_modules = [module1]
    )
```

Используя эти 2 файла процесс создания модуля можно запустить, как показано ниже (в каталоге `c_interaction/distutils/` эти действия включены в сценарий сборки `Makefile`):

```
$ python setup.py build
running build
running build_ext
building 'ownmod' extension
creating build
creating build/temp.linux-i686-2.7
gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 \
    ... \
    -fPIC -I/usr/include/python2.7 -c ownmod.c -o build/temp.linux-i686-2.7/ownmod.o
creating build/lib.linux-i686-2.7
gcc -pthread -shared -Wl,-z,relro build/temp.linux-i686-2.7/ownmod.o \
    -L/usr/lib -lpython2.7 -o build/lib.linux-i686-2.7/ownmod.so
$ cp build/lib.linux-i686-2.7/ownmod.so ./
```

В процессе сборки создаётся подкаталог `build/lib.linux-i686-2.7/`, в котором и собирается файл модуля `ownmod.so`, а вторая команда (копирования) используется только для переноса модуля в более удобное место при тестировании.

(В примере сборка показана в 32-бит системе. В 64-бит системе итоговый создаваемый каталог будет иметь имя `build/lib.linux-x86_64-2.7`, и именно такое имя нужно использовать в сценариях сборки.)

Теперь мы можем запустить простейшее тестовое приложения, импортирующее созданный **модуль** `ownmod`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import ownmod
ownmod.echo()
```

Как и ожидалось, всё работает без ошибок:

```
$ ./test.py
вывод из экспортированного кода!
```

В показанном примере, с использованием `distutils`, написание самого кода модуля практически ничем не отличается от ручного способа сборки, который демонстрировался ранее, но сам процесс сборки упрощается на порядок.

Проект `Cython`

На самом деле, проект `Cython` является своеобразным клоном языка Python с совместимым синтаксисом, но позволяющий, с некоторыми ограничениями на статические определения типов, **компилировать** программу сразу в исполняемый код (в отличие от интерпретации байт-кода в Python). В зависимости от класса задач, утверждается что это позволяет ускорить выполнение в 100 или, иногда, даже 1000 раз!

Но кроме этого, `Cython` — язык программирования, упрощающий написание C/C++ модулей для Python-приложений. В коде, написанном на `Cython`, кроме стандартного синтаксиса Python поддерживается и прямой вызов функций и методов C/C++. Код `Cython`

преобразуется в C/C++ код для последующей компиляции, и впоследствии может использоваться либо как расширение стандартного Python, либо как независимое приложение со встроенной библиотекой выполнения `Cython`.

`Cython` потребуется установить дополнительно, так как по умолчанию он, скорее всего, будет отсутствовать в системе. Это можно сделать из исходных кодов на сайте проекта, но в данный момент `Cython` настолько распространён, что его инсталляционный пакет присутствует практически во всех дистрибутивах Linux, например:

```
$ dnf list cython*
Доступные пакеты
Cython.x86_64
```

И, конечно, вы должны его установить:

```
$ sudo dnf install Cython
...
Package      Архитектура         Версия           Репозиторий                    Размер
================================================================================
=
Установка:
 Cython    x86_64      0.23.4-3.fc23             updates                        2.5 M
...
Объем загрузки: 2.5 M
Объем изменений: 11 M
Установлено:
  Cython.x86_64 0.23.4-3.fc23
Выполнено!
```

То же самое будет и в DEB дистрибутивах: Debian, Ubuntu, Mint, … (только при использовании их собственного пакетного менеджера `apt`).

Сначала рассмотрим, как в `Cython` выполнить создание модуля из Python-совместимого кода. При построении модуля средствами `Cython` используется уже рассмотренный пакет `distutils`, но конфигурационный файл `setup.py` будет иметь уже другое содержание.

Конфигурационный файл:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext


ext_modules = [ Extension( "ownmod", [ "ownmod.pyx"] ) ]


setup(
  name = 'first app',
  cmdclass = { 'build_ext': build_ext },
  ext_modules = ext_modules
)
```

Реализация самого модуля останется практически без изменений, как можно увидеть в листинге (исходный код в файле `c_interaction/cython/pfirst/ownmod.pyx` — `Cython` использует расширение `.pyx`, чтобы обозначить, что этот код подлежит компиляции, в отличие от расширения `.py` самого Python).

Код модуля:

```
def echo():
   print "вывод из компилированного (cython) кода!"
```

Как уже было сказано, код на Cython требует **компиляции** (новый для нас термин в контексте Python). Этот процесс состоит из двух последовательных этапов:

- Cython компилирует .pyx-файл в .c-файл, содержащий код модуля расширения Python;
- Компилятор языка C компилирует далее полученный .c-файл в .so-файл разделяемой (динамической) библиотеки, который можно импортировать обычным образом как модуль, прямо из Python-кода.

Обратите внимание, что благодаря обобщённой схеме (независимо от того, модуль написан на C или на Cython) с библиотекой модуля можно связать любой код на языке C. Выполняется это так:

```
$ python setup.py build_ext --inplace
running build_ext
cythoning ownmod.pyx to ownmod.c
building 'ownmod' extension
creating build
creating build/temp.linux-i686-2.7
gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 ...\
    -fPIC -I/usr/include/python2.7 -c ownmod.c -o build/temp.linux-i686-2.7/ownmod.o
gcc -pthread -shared -Wl,-z,relro build/temp.linux-i686-2.7/ownmod.o
                                          -L/usr/lib           -lpython2.7           -o
/home/olej/2013_WORK/Python/PythonC/cython/pfirst/ownmod.so


$ ls -l ownmod.*
-rw-rw-r-- 1 olej olej 72054 ноя 17 23:38 ownmod.c
-rw-rw-r-- 1 olej olej    91 авг 16  2013 ownmod.pyx
-rwxrwxr-x 1 olej olej 43032 ноя 17 23:38 ownmod.so
```

Модуль можно вызывать из кода, написанного на Python, невзирая на то, что сам модуль был написан на Cython:

```
$ python
Python 2.7.3 (default, Jul 24 2012, 10:05:39)
[GCC 4.7.0 20120507 (Red Hat 4.7.0-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ownmod
>>> ownmod.echo()
вывод из компилированного (cython) кода!
```

В следующем примере мы реализуем математические вычисления (многократное вычисление функции sin(x)) и добавим в Cython-код статические определения типов в силе C (cdef), а также подключим функцию вычисления sin(x) из стандартной математической библиотеки C (libm.so). Два варианта реализации функции sin(x) на Cython показаны в листинге ниже (файл c_interaction/cython/math/csin.pyx).

Математические вычисления на Cython:

```
import time
import math
```

```
def sin1( int n ):
    cdef int i
    cdef double x, clc
    x = 0.0
    clc = time.time()
    for i in xrange( n ):
        x += math.sin( x )
    return time.time() - clc

cdef extern from "math.h":
    double sin( double )

def sin2( n ):
    x = 0.0
    clc = time.time()
    for i in xrange( n ):
        x += sin( x )
    return time.time() - clc
```

А в следующем листинге показана эквивалентная реализация той же функции на Python (файл `c_interaction/cython/math/psin.py`).

Альтернатива — математические вычисления на Python :

```
import time
import math

def sin0( n ):
    x = 0.0
    clc = time.time()
    for i in xrange( n ):
        x += math.sin( x )
    return time.time() - clc
```

А затем мы напишем конфигурационный файл `setup.py` пакета `distutils` для сборки Cython-реализации модуля (обратите внимание на обязательное подключение библиотеки `libm.so` и как она указывается).

Конфигурационный файл:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [ Extension( "csin", [ "csin.pyx" ], libraries=["m"] ) ]

setup(
  cmdclass = { 'build_ext': build_ext },
  ext_modules = ext_modules
)
```

Теперь всё готово для сборки компилированной библиотеки `csin.so` для нашего модуля:

```
$ python setup.py build_ext --inplace
running build_ext
cythoning csin.pyx to csin.c
building 'csin' extension
creating build
creating build/temp.linux-i686-2.7
gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 ...\
    -fPIC -I/usr/include/python2.7 -c csin.c -o build/temp.linux-i686-2.7/csin.o
gcc -pthread -shared -Wl,-z,relro build/temp.linux-i686-2.7/csin.o -L/usr/lib ... \
    -lm -lpython2.7 -o /home/olej/2013_WORK/Python/PythonC/cython/math/csin.so
```

Здесь показано тестовое приложение, сравнивающее 3 варианта реализации по затратам на время выполнения (файл `c_interaction/cython/math/test.py`):

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from sys import argv
from psin import sin0 as t1
from csin import sin1 as t2
from csin import sin2 as t3

n = ( len( argv ) > 1 and int( argv[ 1 ] ) ) or 5000000
print "время %i вычислений %5.2f секунд" % ( n, t1( n ) )
print "время %i вычислений %5.2f секунд" % ( n, t2( n ) )
print "время %i вычислений %5.2f секунд" % ( n, t3( n ) )
```

В итоге интересно получается, что реализация на Python оказалась быстрее реализации на Cython, но в разы самой быстрой оказалась Cython-версия с использованием C-функции `sin(x)` из библиотеки `libm.so`:

```
$ ./test.py
время 5000000 вычислений  2.40 секунд
время 5000000 вычислений  2.66 секунд
время 5000000 вычислений  0.15 секунд
```

Вопреки, возможно, ожиданиям, компилированная Cython-версия уступила традиционной реализации на Python по производительности. Но это и понятно, так как большую долю времени в вычислениях занимает вычисление `sin(x)`, производимое **модулем** `math` из библиотеки модулей Python. Но как только мы присоединили к модулю Cython C-реализацию функции `sin(x)` из библиотеки `libm.so`, то скорость вычислений выросла в **18 раз**.

В этом примере было показано использование в модуле C-реализации из собранной библиотеки `libm.so`, без доступа к исходному коду реализации. Очевидно, что прикомпоновать к DLL модуля (подобной `csin.so`) объектный код, полученный компиляцией собственного исходного кода на C (или C++) ещё проще. В архиве примеров показаны несколько вариантов такого подключения (подкаталог `c_interaction/cython/xfirst`).

Особенности применительно к C++

Всё, что сказано выше про техники взаимодействия кода Python с кодом C, в равной мере относится и к языку C++, поскольку C++ является совместимым надмножеством C. Но для

C++ разработчики предлагают ещё больше специфических интерфейсных технологий.

Библиотека Boost.Python

Проект Boost — это шаг на пути к мульти-платформенной разработке для C++ проектов, так как код, построенный на Boost, работает без изменений на платформах Windows, Linux, Solaris, FreeBSD и др. Именно этим объясняются пристрастия компаний, разрабатывающих программное обеспечение, к использованию Boost (или отдельных частей из Boost).

На данный момент полный Boost представляет собой **набор** из нескольких десятков разнообразных инструментов от **различных** разработчиков, иногда очень слабо связанных между собой. Некоторые из составных частей Boost используются повсеместно (например, `Thread, Regex, Boost Filesystem Library, Signals,` ...). Другие составные части Boost являются, скорее, экзотикой, и используются только узкими специалистами (`Lambda, Identity Type, Meta State Machine,` ...). Конечно, при такой вольной философии наполнения Boost не мог обойти стороной и Python. Эта часть Boost известна как Boost Python Library (автор Dave Abrahams).

Подготовим тестовый проект Python, в котором мы будем использовать Boost для вызова из него C++ кода. В листингах ниже представлены все необходимые файлы (которые найдёте в каталоге `c_interaction/boost/first`).

Исходный код вызываемого модуля (файл `ownmod.cc`):

```
#include <iostream>
#include <iomanip>
using namespace std;


void echo( void ) {
   cout << "вывод из экспортированного кода!" << endl;
}
```

Общий заголовок (файл `ownmod.h`)

```
void echo( void );
```

Файл интерфейса Boost (файл `ownmod_wrap.cc`)

```
#include <boost/python.hpp>
#include "ownmod.h"


BOOST_PYTHON_MODULE( ownmod ) {
    using namespace boost::python;
    def( "echo", echo );
}
```

Соберём модуль, как показано ниже:

```
$ c++ -c -fpic ownmod.cc
$ c++ -c -fpic ownmod_wrap.cc `python-config --includes`
$ ls *.o
ownmod.o  ownmod_wrap.o
$ c++ -shared ownmod_wrap.o ownmod.o -l boost_python -o ownmod.so
$ ls *.so
ownmod.so
```

Для тестирования используем уже знакомое приложение:

```
$ python test.py
```

```
вывод из экспортированного кода!
```

Это только самая общая схема использования `Boost.Python`. На практике возможно возникновение сложностей при необходимости передавать параметры и возвращать значения. Но все особенности реализации этих действий подробно описаны в документации `Boost.Python` (см. библиографию в конце текста). В каталоге `c_interaction/boost/libra` находится отдельный тестовый пример с передачей параметров и возвратом значений (но здесь мы не станем его детально рассматривать).

Из всех рассмотренных способов взаимодействия с кодом, написанным на языке C++, `Boost.Python` является самым простым в использовании.

Примечание: К сожалению, `Boost.Python` — это, возможно, уже устаревший проект, так как разработка его основных компонентов была завершена к 2003 году. Хотя он замечательно работает с Python 2, но с Python 3 всё не так просто, и вопрос адаптации `Boost.Python` к Python 3 требует дополнительного изучения.

Проект SWIG

Проект SWIG предоставляет универсальный интерфейс к C/C++ не только из языка Python, но и других языков. В его документации говорится, что SWIG – это инструмент разработки программного кода, позволяющий интегрировать программы, написанные на языках C и C++, с другими высокоуровневыми языками программирования. SWIG может использоваться с различными типами языков, включая такие скриптовые языки, например, Perl, PHP, Python, Tcl и Ruby. Также поддерживаются такие языки как C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), D, Go, язык Java, включая платформу Android и др.

Но повышенная универсальность и гибкость, как всегда, ведёт к увеличению сложности, поэтому при освоении SWIG возможно возникновение досадных и неожиданных ошибок. Зато освоив этот инструмент, вы получите возможность создавать интерфейсы к любому из указанных языков.

Проект SWIG активно развивается, так его последняя, на сегодня, версия 3.0.12 была выпущена 28 января 2017 года. Так как SWIG является мультиплатформенным проектом, то пользователям предлагаются версии для платформ Windows, Linux, Solaris, OS-X/Darwin (но в дальнейших примерах будет рассматриваться версия только для ОС Linux). Актуальную версию SWIG можно собрать из исходных кодов (см. ссылку конце текста). Но его можно найти практически в любом дистрибутиве Linux и установить с помощью менеджера используемой пакетной системы, как показано ниже:

```
$ sudo dnf install swig
...
Объем загрузки: 1.5 M
Объем изменений: 5.4 M
...
Установлено:
  swig.x86_64 3.0.7-8.fc23
Выполнено!

$ swig -version
SWIG Version 3.0.7
Compiled with g++ [x86_64-redhat-linux-gnu]
Configured options: +pcre
Please see http://www.swig.org for reporting bugs and further information
```

Базовая идея SWIG заключается в том, что:

1. готовится реализация будущего **модуля** на языке C (`.c`) или C++ (`.cc`);
2. создаётся интерфейсный файл (`.i`) на специальном **макроязыке** SWIG;
3. SWIG генерирует код для интерфейсного файла (`.c`), который далее собирается вместе с реализацией модуля (`.c`) в разделяемую динамическую библиотеку модуля;

При этом язык программирования (например, Python), для которого генерируется интерфейсный файл, указывается в качестве **входного параметра** команды SWIG, а сам вид интерфейсного файла (`.i`) **не зависит** от того, для какого конкретного языка будет генерироваться интерфейс.

Далее будет представлен простой пример использования SWIG, который может послужить отправной точкой для последующих разработок. В листинге представлен исходный модуль на языке C (сама степень сложности кода нас не интересует), реализующий требуемую функцию (см. файл `c_interaction/swig/first/ownmod.c`).

Код реализации модуля:

```
#include <stdio.h>
void echo( void ) {
   printf( "вывод из экспортированного кода!\n" );
}
```

К этому модулю следует подготовить файл с описанием **интерфейса** на макроязыке SWIG. В листинге представлен пример такого файла для нашего модуля (см. файл `ownmod.i` из того же каталога) — определение интерфейса:

```
%module ownmod
%{
extern void echo( void );
%}
```

Этих **двух** файлов достаточно для начала работы с SWIG. Далее выполняются следующие шаги:

- генерация кода интерфейса (результатом будет файл, по умолчанию с именем `ownmod_wrap.c`, но название этого файла можно переопределить опциями команды `swig`):

```
$ swig -python -module ownmod ownmod.i
```

- компиляция всех исходных файлов `.c` в соответствующие объектные файлы `.o`:

```
$ gcc -c -fpic ownmod_wrap.c ownmod.c -DHAVE_CONFIG_H -I/usr/include/python2.7
```

- сборка их в единую библиотеку:

```
$ gcc -shared ownmod.o ownmod_wrap.o -o ownmod.so
```

- также сборку в библиотеку можно выполнить следующей командой:

```
$ ld -shared ownmod.o ownmod_wrap.o -o ownmod.so
```

Ещё раз обратим внимание на то, что генерация командой `swig` может производиться не только под язык Python, но и ещё в добрых два десятка языков программирования, обслуживаемых SWIG.

В примерах кодов, в каталоге `c_interaction/swig/first`, все команды, требуемые для сборки, собраны в единый скрипт `Makefile`:

```
$ make
swig -python -module ownmod ownmod.i
gcc   -c   -fpic     ownmod_wrap.c   ownmod.c   -DHAVE_CONFIG_H   -I/usr/include/python2.7
```

```
-I/usr/include/python2.7
ownmod_wrap.c: В функции «_wrap_echo»:
ownmod_wrap.c:2986:3: предупреждение: неявная декларация функции «echo» [-Wimplicit-function-declaration]
   echo();
   ^
gcc -shared ownmod.o ownmod_wrap.o -o _ownmod.so
rm *.o

$ ls *ownmod*
ownmod.c  ownmod.i  ownmod.py  ownmod.pyc  _ownmod.so  ownmod_wrap.c
```

Теперь мы можем использовать созданный модуль из кода Python, как показано в файле `c_interaction/swig/first/test.py` — тестовое приложение:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import ownmod
ownmod.echo()
```

Запустим тестовый Python-сценарий:

```
$ python test.py
вывод из экспортированного кода!
```

Хотя показанный здесь пример крайне схематичен, но он показывает общую логику использования SWIG для разработки модулей любой степени сложности. Однако, в отдельных случаях описание интерфейсов (`.i`) на макроязыке SWIG может оказаться сложной задачей. В архив кодов примеров `python_examples/c_interaction/swig` включен подкаталог `c_interaction/swig/exampl`, построенный на основе заимствования с домашней страницы проекта SWIG, демонстрирующий передачу параметров и возврат результатов. Ниже показаны только ключевые файлы и результирующее исполнение (из Python) — этого достаточно для понимания:

- файл `example.c`:

```
#include <time.h>
double My_variable = 3.0;

int fact( int n ) {
   if( n <= 1 ) return 1;
   else return n * fact( n - 1 );
}

int my_mod( int x, int y ) {
   return ( x % y );
}

char *get_time( void )  {
   time_t ltime;
   time( &ltime );
   return ctime( &ltime );
}
```

- файл `example.i`:

```
%module exampl
%{
/* Put header files here or function declarations like below */
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

- `Makefile` :

```
INCLUDE = $(shell python-config --includes)
all:    exampl.i exampl.c
        swig -python exampl.i
        gcc $(INCLUDE) -fPIC -c exampl.c exampl_wrap.c
        ld -shared exampl.o exampl_wrap.o -o _exampl.so
```

- вызывающая программа `test.py` :

```
import exampl
print( exampl.fact( 5 ) )
print( exampl.my_mod( 7, 3 ) )
print( exampl.get_time() )
```

И выполнение всего этого хозяйства:

```
$ ./test.py
120
1
Sat Nov 18 15:52:28 2017
```

Ещё один представленный пример `freq.c` (каталог `c_interaction/swig/frequen`) — подсчёт числа вхождений каждого из 256 символов ASCII в тексте, переданном программе в качестве параметра запуска.

Внимание: показанный вариант программы замечательно работает в 32-бит архитектуре, но не будет работать в 64-бит архитектуре (ошибка сегментирования из-за приведение к типу указателя от целого другого размера). Адаптация примера под 64-бит оставлена для несложной самостоятельной проработки.

Исходная функция на языке C, требующая динамического выделения буфера:

```
#include <stdlib.h>


int* frequency( char s[] ) {
   int *freq;
   char *ptr;
   freq = (int*)( calloc( 256, sizeof( int ) ) );
   if( freq != NULL )
```

```
      for( ptr = s; *ptr; ptr++ )
         freq[ *ptr ] += 1;
   return freq;
}
```

Подобная функция потребует более сложного файла с описанием интерфейса, так как она динамически выделяет память буфера, которая должна освобождаться **вызывающим** кодом Python. Ниже представлен пример такого файла (см. файл `freq.i`).

Описание интерфейса для функции `frequency()` :

```
%module freq
%typemap(out) int* {
   int i;
   $result = PyTuple_New( 256 );
   for( i = 0; i < 256; i++ )
      PyTuple_SetItem( $result, i, PyLong_FromLong( $1[ i ] ) );
   free( $1 );
}
extern int* frequency( char s[] );
```

Сборка модуля выполняется также как и в предыдущем примере. В листинге представлено Python-приложение, использующее модуль с данной функцией.

Программа подсчёта частоты вхождений символов:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from sys import argv
import freq

sarg = lambda : ( len( argv ) > 1 and argv[ 1 ] ) or str( input( "string?: " ) )

f = freq.frequency( sarg() )

s = ""
for i in range( 256 ):
    if f[ i ] != 0:
        s = s + "'%c'[0x%x]:%d " % ( i, i, f[ i ] )

print s
```

Запустим эту программу:

```
$ python test.py "1234123121 abc ab a"
' '[0x20]:3 '1'[0x31]:4 '2'[0x32]:3 '3'[0x33]:2 '4'[0x34]:1 'a'[0x61]:3 'b'[0x62]:2 ...
```

Ещё одним преимуществом использования SWIG является простота создания **интерфейсных модулей** к уже существующим DLL библиотекам, в частности, к системным библиотекам и библиотекам сторонних открытых проектов GNU. В этом случае не требуется создавать вообще никаких реализаций на C (они уже имеются), достаточно будет подготовить только файл описания интерфейса, как показано ниже (файл `c_interaction/swig/fileio/fileio.i`).

Интерфейс к файловым операциям:

```
%module fileio
extern FILE *fopen(char *, char *);
extern int fclose(FILE *);
extern unsigned fread(void *ptr, unsigned size, unsigned nobj, FILE *);
extern unsigned fwrite(void *ptr, unsigned size, unsigned nobj, FILE *);
extern void *malloc(int nbytes);
extern void free(void *);
```

Остаётся выполнить сборку разделяемой библиотеки `_fileio.so`, реализующей файловый интерфейс для нового модуля. Обратите внимание, что в сборку включена стандартная C-библиотека `libc.so`, в которой и находится реализация файловых операций:

```
$ swig -python fileio.i
$ gcc -c -fpic fileio_wrap.c -DHAVE_CONFIG_H -I/usr/include/python2.7 \
      -I/usr/lib/python2.7/config
$ ld -shared fileio_wrap.o -lc -o _fileio.so
```

Теперь в коде Python можно напрямую использовать файловые операции POSIX, как показано в листинге файла `fiotest.py`.

Копирование файлов из Python с использованием POSIX операций:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from fileio import *
from sys import argv

def filecopy( source, target ):           # копирование файла
    sum = long( 0 )
    f1 = fopen( source, "r" )
    if f1 == None: return -1
    f2 = fopen( target, "w" )
    buffer = malloc( 8192 )
    nbytes = fread( buffer, 1, 8192, f1 )
    while( nbytes > 0 ):
        fwrite( buffer, 1, nbytes, f2 )
        sum = sum + nbytes
        nbytes = fread( buffer, 1, 8192, f1 )
    free( buffer )
    fclose( f1 )
    fclose( f2 )
    return sum

n = filecopy( ( len( argv ) > 1 and argv[ 1 ] ) or "in.txt", \
              ( len( argv ) > 2 and argv[ 2 ] ) or "out.txt" )
print "скопировано %d байт" % n
```

Запустим данное приложение и посмотрим на результаты его работы:

```
$ python fiotest.py in.txt out.txt
```

```
скопировано 178 байт
$ ls -l *.txt
-rw-rw-r-- 1 olej olej 178 июня  21 23:48 in.txt
-rw-rw-r-- 1 olej olej 178 июня  22 00:14 out.txt
```

Обратная интеграция Python-кода в C/C++ приложения

Интеграция Python-кода в проект, написанный на C или C++, представляет собой обратную задачу, которая также может возникнуть в различных областях. Многие крупные проекты (например, телефонные коммутаторы для IP-телефонии — Asterisk и FreeSWITCH) используют встраивание одного или нескольких интерпретирующих языков (Python, JavaScript, Lua, ...) для быстрой реализации сценариев, позволяющих управлять основной функциональностью (например, осуществлять разнообразную конфигурацию, гибко и «на ходу»). Также подобные языки могут применяться для конфигурации ваших проектов без нужды в их повторной компиляции. В листинге показан простейший интерпретатор, встроенный в C-приложение (см. файл `interpr.c` в каталоге `c_interaction/embedding`).

Интерпретатор Python, встроенный в C-приложение:

```
/* Пример встраивания интерпретатора Python в другую программу */
#include "Python.h"

main( int argc, char **argv ) {
   Py_SetProgramName( argv[ 0 ] ); // Передает argv[ 0 ] интерпретатору Python
   Py_Initialize();                // Инициализация интерпретатора
   /* ... */
   while( 1 ) {                    // Выполнение операторов Python
      char buf[ 120 ];
      fprintf( stdout, ">>> " );
      fflush( stdout );
      fgets( buf, sizeof( buf ) - 2, stdin );
      buf[ strlen( buf ) - 1 ] = '\n';
      buf[ strlen( buf ) ] = '\0';
      if( 0 == strcmp( buf, "quit\n" ) || 0 == strcmp( buf, "exit\n" ) ) break;
      PyRun_SimpleString( buf );
   }
   Py_Finalize();           // Завершение работы интерпретатора
}
```

Пример запуска данного приложения:

```
$ ./interpr
>>> print 2+2
4
>>> import sys
>>> print sys.version_info
sys.version_info(major=2, minor=7, micro=3, releaselevel='final', serial=0)
>>> quit
```

Хотя этот пример и сильно похож на использование стандартного интерпретатора `CPython`, (того, который мы и запускаем командой `python`), но на самом деле это gjkyjcnm.

самостоятельное приложение, убедиться в этом вы можете тем, что в нём не обрабатываются клавиши редактирования вводимой строки, и в тоже время из него можно выйти, введя команду `quit` (стандартный интерпретатор не завершается подобным образом).

Важное преимущество подобного встраивания Python-функций в C-приложения — это возможность **доступа** к переменным и объектам C-кода из Python (параметризация приложения). Таким образом вы можете быстрым написанием Python-скрипта радикально менять поведение крупного программного проекта. Эта возможность обеспечивается специальным API для встраивания, входящим в состав Python, (дополнительную информацию об этом API можно найти в ссылках библиографии в конце текста: «Extending and Embedding the Python Interpreter»).

Разработка GUI-приложений

Ещё одной привлекательной особенностью Python является простота, скорость и гибкость в создании приложений с графическим интерфейсом пользователя (GUI) — это целая отдельная область приложений. Это преимущество связано не только с большим количеством поддерживаемых графических библиотек: Tkinter, PyQt, PyGTK, wxPython, Pygames и др. Основная причина заключается в интерпретирующей природе платформы Python, почему, из-за доступности Python-кода, внешний вид графического приложения всегда можно легко изменить или дополнить. А весь интерфейс из Python-кода к фактической реализации GUI скрыт внутри модулей библиотеки Python.

Однако этим преимущества разработки GUI-приложений именно на Python не исчерпываются, так как Python предлагает:

- независимость от платформы: графическое приложение, разработанное на Python в одной ОС (например, Linux) будет с большой степенью вероятности адекватно работать в любой другой среде (Windows, MacOS, Solaris, FreeBSD, …) или потребует для этого незначительных доработок;
- GUI-приложения в основном являются диалоговыми, т.е. предназначенными для взаимодействия с пользователем, при этом скорость работы приложения определяется действиями пользователя, и здесь исчезает один из основных формальных недостатков Python — его замедленность по сравнению с C или C++;
- из-за простоты интеграции Python с C/C++, визуальные компоненты проекта (GUI) могут быть написаны на Python с учётом скорости разработки (frontend), а внутренние процедуры для обработки данных — на C/C++ (backend).

В этой и последующей статьях мы обсудим различные мульти-платформенные технологии создания GUI-приложений, доступные для Python-программистов.

GUI-приложения

Целью этого раздела будет **беглый** (иначе это сделать невозможно в сжатый объём) обзор различных вариантов, доступных при выборе инструментария для разработки GUI-приложений. Таким образом программист сможет определить, какой инструмент наиболее подходит для решения конкретной задачи. Подобная постановка задачи связана ещё и с тем, что для всех из многочисленных инструментов для разработки GUI-приложений существуют (более или менее удачные и полные) руководства, но практически отсутствуют материалы, в которых бы выполнялся сравнительный анализ разных подходов.

При всём разнообразии, громоздкости и кажущейся сложности различных инструментальных средств создания графических приложений, их общая схема однообразна и проста:

- создаётся бесконечный цикл опроса (ожидания) событий (**главный цикл**), порождаемых действиями пользователя (ввод с клавиатуры, перемещение указателя или нажатие кнопки мыши и т.д.);

- для каждого потенциально возможного (**обрабатываемого**) события назначается функция обратного вызова (**callback функция**), которая будет вызываться при наступлении этого события (**обработчик события**);

- при возникновении событий, для которых не были назначены обработчики, они будут либо игнорироваться, либо использоваться умалчиваемые обработчики;

- на каждом «витке» главного цикла приложения анализируются наступившие (со времени предыдущего «витка») события, и для каждого такого события вызывается его обработчик, если ожидающих событий несколько, то они обрабатываются в порядке очереди;

– всякий GUI-инструмент имеет некоторый набор графических компонентов (виджетов - widget) и средства для компоновки таких виджетов в окне приложения, а каждый из компонентов имеет свой специфический набор событий, на которые он может реагировать;

Такой подход называется событийно управляемым программированием (**event driven programming**). Все GUI приложения, независимо от используемой графической технологии, будут использовать этот общий шаблон.

Исходя из подобной сжатой формулировки, в дальнейшем нас будут интересовать только вопросы применимости, выбора, установки и начала работы с инструментом. Но иногда мы будем обращаться и к специфическим требованиям для определённых инструментов по оформлению функций обработчиков событий. Вопросы компоновки виджетов в конкретных средах рассматриваться не будут.

Примечание. Внешний вид одного и того же GUI-приложения может отличаться при запуске в разных средах (операционных системах, менеджерах рабочего стола, оконных менеджерах). Это обусловлено тем, что фактическую прорисовку виджетов обеспечивают не графические библиотеки, а оконный менеджер графической системы.

Tkinter

Tkinter является стандартным модулем Python и входит в состав его стандартной библиотеки. Модуль `Tkinter` позиционируется для **быстрого** написания GUI-приложений. Его описание в достаточной мере детализации доступно в встроенной справочной системе `pydoc`.

Примечание. Справочная система `pydoc` доступна только в операционных системах семейства UNIX (Linux и др., о `pydoc` мы поговорим отдельно в другой раз). Для обращения к ней используется команда:

```
$ pydoc -p8080
```

Значение опции `-p` определяет порт TCP, через который доступна система и который можно переопределить. После этого любым Web-браузером можно подключиться к этой справочной системе, набрав в адресной строке браузера: `http://localhost:8080`. В реализации Python для ОС Windows есть своя справочная система, устанавливаемая вместе с Python.

Хотя модуль `Tkinter` считается входящим в состав библиотеки стандартных модулей Python, может потребоваться его отдельная инсталляция (или, как минимум, проверка его наличия), так как обычно `Tkinter` не устанавливается при установке Python. Кроме того существуют его различные версии специально для Python 2 и Python 3.

- в .rpm дистрибутивах (Fedora, CentOS, Scientific и др.):

```
# yum list tkinter.*
...
Установленные пакеты
tkinter.i686                                    2.7.3-7.2.fc17
```

```
# yum install python3-tkinter.i686
...
Установлено:
  python3-tkinter.i686 0:3.2.3-7.fc17
```

- в .deb дистрибутивах (Debian, Ubuntu, Mint и др.):

```
$ aptitude search python-tk
p   python-tk           - Tkinter - написание Tk программ на Python
p     python-tk-dbg           - Tkinter - Writing Tk applications with Python (debug
extension)
p   python-tksnack      - Sound extension to Tcl/Tk and Python/Tkinter - Python library
# apt-get install python-tk
...
Настраивается пакет python-tk (2.7.3-1) …
$ aptitude search python3-tk
p   python3-tk          - Tkinter - Writing Tk applications with Python 3.x
p    python3-tk-dbg         - Tkinter - Writing Tk applications with Python 3.x (debug
extension)
# apt-get install python3-tk
...
Настраивается пакет python3-tk (3.2.3-1) ...
```

Как видно из примера, иногда могут возникнуть сложности из-за **разных наименований пакетов** в различных дистрибутивах, но эта проблема легко решается.

Примечание: Существует ещё одна особенность использования стандартного модуля `Tkinter`, так в версиях Python 2.X он должен записываться в написании `Tkinter`:

```
from Tkinter import Label
```

Но в версиях Python 3.X он должен записываться в написании как `tkinter`, и та же строка будет выглядеть по-другому:

```
from tkinter import Label
```

Если же необходимо обеспечить совместимость между версиями, то строки импорта могут быть записаны как-то так:

```
try:
    from tkinter import *
except ImportError:
    from Tkinter import *
```

В качестве иллюстрации будет использована не слишком сложная (но и не тривиальная) задача угадывания пользователем случайного числа, генерируемого программой. Вариант кода для Python 3, реализующего задачу при использовании `Tkinter`, показан в листинге ниже (см. файл `GUI/Tkinter/gntm.py`).

Оконное приложение, использующее Tkinter:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-


from sys import argv
import random
import math
```

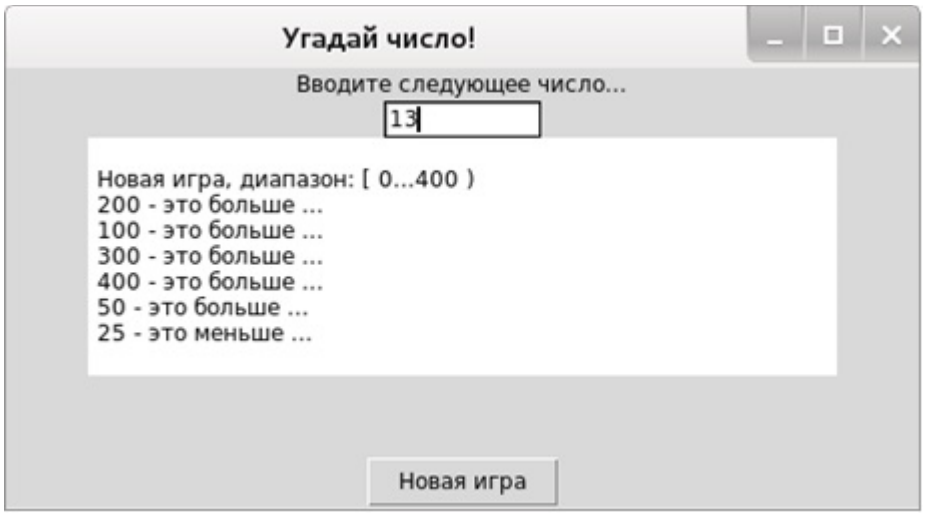
```
from tkinter import *

val_range = 0    # диапазон числа
secret = 0       # угадываемое число
guesses = 0      # число попыток
limits = 0       # макс. число попыток

def new_game():
    global secret, guesses, limits
    msg[ 'text' ] = ' ' * 100 + \
        '\nНовая игра, диапазон: [ 0...{} )\n'.format( val_range )
    secret = random.randrange( 0,val_range )
    guesses = 0
    limits = int( math.ceil( math.log( val_range, 2 ) ) )

def input_guess( event ):
    global guesses
    if guesses < 0 :            # признак завершённой игры
        msg[ 'text' ] += 'Начните новую игру...\n'
        ent.delete( 0, END )
        return
    try :
        value = int( ent.get() )
    except ValueError:
        msg[ 'text' ] += 'Ошибка: значение должно быть целочисленным!\n'
        ent.delete( 0, END )
        return;
    ent.delete( 0, END )
    guesses += 1
    if value < secret :
         msg[ 'text' ] += '{} - это меньше ...\n'.format( value )
    elif value > secret :
        msg[ 'text' ] += '{} - это больше ...\n'.format( value )
    else :
        msg[ 'text' ] += 'Игрок выиграл!\n'
        guesses = -1             # признак завершённой игры
        return
    if guesses >= limits :
        msg[ 'text' ] += 'Компьютер выиграл! Загадно было {}\n'.format( secret )
        guesses = -1             # признак завершённой игры
        return

root = Tk()
root.title( 'Угадай число!' )    # окно пиложения
root.geometry( '500x240' )
```

```
Label( root, text='Вводите следующее число...' ).pack( side=TOP )
ent = Entry( root, width=10 )    # поле ввода
ent.pack( side=TOP )
ent.focus() # избавить от необходимости выполнять щелчок мышью для фокуса
ent.bind( '<Return>', input_guess )
Button( root, text=' Новая игра ', command=new_game ).pack( side=BOTTOM )
msg = Message( root, bg='white', fg='black', width=400, borderwidth=0 )
msg.pack( side=TOP )             # окно результата

val_range = ( len( argv ) > 1 and int( argv[ 1 ] ) ) or 100 # параметр - диапазон
new_game()
root.mainloop()
```

Здесь с помощью менеджера компоновки `Tkinter` при вызове `pack()` в приложение было добавлено несколько виджетов: поле ввода, кнопка, текстовые поля вывода...

На рисунке показан результат запуска данного приложения:

```
$ ./gntm.py 400
...
```



Руководство по использованию `Tkinter` более чем детально изложено в 1-м томе книги Марка Лутца (см. библиографию). Представленной информации будет вполне достаточно для квалифицированной работы с `Tkinter`.

Следует также иметь в виду некоторые ограничения этого способа построения графических приложений, так как `Tkinter` основан на инструментарии `Tcl/Tk`, который поддерживает несколько устаревший вид виджетов, как считают многие. По умолчанию, этот GUI-инструментарий не поддерживает некоторые сложные виджеты, например, деревья или окна с закладками и др.

PyQt

Пакет (несколько различных модулей) PyQt являются интерфейсом из Python к графической системе Qt. Для использования PyQt потребуется установить дополнительные модули Python.

Для Python 2:

```
$ aptitude search python-qt
p   ipython-qtconsole           - enhanced interactive Python shell - Qt console
```

```
p   python-qt4               - Python bindings for Qt4
p   python-qt4-dbg           - Python bindings for Qt4 (debug extensions)
p   python-qt4-dbus          - D-Bus Support for PyQt4
p   python-qt4-dbus-dbg      - D-Bus Support for PyQt4 (debug extensions)
p   python-qt4-dev           - Development files for PyQt4
p   python-qt4-doc           - документация и примеры для PyQt4
p   python-qt4-gl            - Python bindings for Qt4's OpenGL module
p   python-qt4-gl-dbg        - Python bindings for Qt4's OpenGL module (debug extension)
p   python-qt4-phonon        - Python bindings for Phonon
p   python-qt4-phonon-dbg    - Python bindings for Phonon (debug extensions)
p   python-qt4-sql           - интерфейс Python к модулю Qt4 SQL
p   python-qt4-sql-dbg       - Python bindings for PyQt4's SQL module (debug extension)

$ sudo apt-get install python-qt4 python-qt4-dev python-qt4-doc
...
```

Для Python 3:

```
$ aptitude search  python3-pyqt*
p   python3-pyqt4              - Python3 bindings for Qt4
p   python3-pyqt4-dbg          - Python3 bindings for Qt4 (debug extensions)
p   python3-pyqt4.phonon       - Python3 bindings for Phonon
p   python3-pyqt4.phonon-dbg   - Python3 bindings for Phonon (debug extensions)
p   python3-pyqt4.qsci         - Python 3 bindings for QScintilla 2
p   python3-pyqt4.qtopengl     - Python 3 bindings for Qt4's OpenGL module
p   python3-pyqt4.qtopengl-dbg - Python 3 bindings for Qt4's OpenGL module (debug...
p   python3-pyqt4.qtsq         - Python3 bindings for PyQt4's SQL module
p   python3-pyqt4.qtsql-dbg    - Python3 bindings for PyQt4's SQL module (debug...

# apt-get install python3-pyqt4
...
```

Далее будут использоваться простые примеры, которых, тем не менее, вполне достаточно для понимания логики приложения и начала работы с этими инструментами (дальше всё становится проще). Ниже в листинге показано простейшее приложение, написанное с использованием PyQt (файл `GUI/PyQt/hw3pyqt.py`).

Пример простейшего PyQt-приложения:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import sys
from PyQt4.QtGui import *

# Каждое приложение должно создать объект QApplication
# sys.argv - список аргументов командной строки
application = QApplication(sys.argv)

# QWidget - базовый класс для всех объектов интерфейса пользователя;
# если использовать для виджета конструктор без родителя, такой виджет станет окном
```

```
widget = QWidget()


widget.resize( 320, 240 )                # изменить размеры виджета
widget.setWindowTitle( "Hello, World!" ) # установить заголовок
widget.show()                            # отобразить окно на экране


sys.exit( application.exec_() )          # запуск основного цикла приложения
```

В этом примере присутствуют все показанные ранее признаки и структура событийно-управляемого приложения, а главное отличие между листингами 1 и 2 исключительно внешнее и заключается в именах вызовов API, но сама логика построения приложения остаётся неизменной. На рисунке показан результат запуска этого приложения:

```
$ python3 hw3pyqt.py
...
```

Проект PyQt имеет параллельно развивающееся ответвление PySide, которое некоторые разработчики считают более удобным, чем оригинальный PyQt.

PyGTK

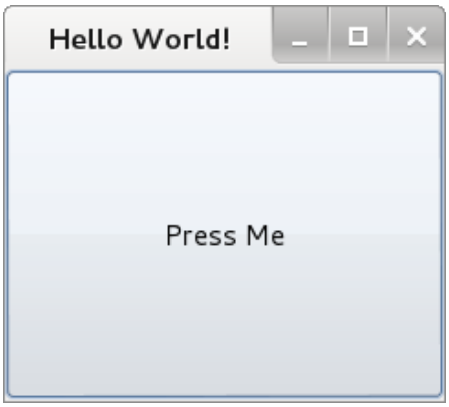
Проект PyGTK – это ещё один мульти-платформенный GUI инструмент, использующий библиотеки GTK+. В листинге ниже показан пример PyGTK-приложения (файл GUI/PyGTK/hw3gtk.py), демонстрирующий что PyGTK также полагается на событийно-управляемый подход.

Пример PyGTK-приложения:

```
#!/usr/bin/env python
#-*- coding: UTF-8 -*-


import gtk


def button_clicked( button ):
    print 'Hello World!'


def main():
    window = gtk.Window()
    window.set_default_size( 240, 180 )
```

```
    window.set_title( 'Hello World!' )
    window.connect( 'destroy', lambda w: gtk.main_quit() )

    button = gtk.Button( 'Press Me' )
    button.connect( 'clicked', button_clicked )
    button.show()

    window.add( button )
    window.present()

    gtk.main()

if __name__ == '__main__':
    main()
```

На рисунке показан результат запуска этого примера:



На каждое событие нажатия кнопки мыши (в поле окна) запущенное приложение выводит в терминал строку «Hello World!»:

```
$ python hw3gtk.py
Hello World!
Hello World!
Hello World!
```

В библиографии приведены ссылки на подробную документацию по PyGTK, в том числе и на русскоязычные материалы.

Начиная с 2012 года (с версии 2.8), обёртки объектов Glib были вынесены в отдельную библиотеку — PyGObject, которая должна будет полностью вытеснить PyGTK при использовании GTK+ версии 3:

```
$ aptitude search python3-gi
p   python3-gi       - Python 3 bindings for gobject-introspection libraries
p   python3-gi-cairo - Python 3 Cairo bindings for the GObject library
p   python3-gi-dbg   - Python 3 bindings for gobject-introspection libraries (debug ...
```

После установки пакета python3-gi (в RPM дистрибутивах нужный пакет называется python-gobject) можно воспроизвести предыдущий пример, но уже с использованием Python 3. В листинге ниже показан обновлённый вариант кода (см. файл GUI/PyGTK/hw4gtk.py). В данном случае изменения носят скорее косметический характер, хотя иногда в ходе миграции из PyGTK в PyGObject могут возникнуть определённые проблемы.

Пример PyGTK-приложения для Python 3 с использованием PyGObject:

```
#!/usr/bin/python3
#-*- coding: UTF-8 -*-
from gi.repository import Gtk

def button_clicked(button):
    print( 'Hello World!' )

def main():
    window = Gtk.Window()
    window.set_default_size( 240, 180 )
    window.set_title( 'Hello World!' )
    window.connect( 'destroy', lambda w: Gtk.main_quit() )

    button = Gtk.Button( 'Press Me' )
    button.connect( 'clicked', button_clicked )
    button.show()

    window.add( button )
    window.present()

   Gtk.main()

if __name__ == '__main__':
    main()
```

Исполнение этого примера в Python 3 создаст аналогичное окно.

```
$ python3 hw4gtk.py
Hello World!
Hello World!
Hello World!
...
```

В таком виде приложение может выполняться и в Python 2:

```
$ python hw4gtk.py
hw4gtk.py:3: PyGIWarning: Gtk was imported without specifying a version first. Use gi.require_version('Gtk', '3.0') before import to ensure that the
  from gi.repository import Gtk
Hello World!
```

WxPython

Проект wxPython происходит от известного проекта wxWidgets. В основе wxWidgets лежит GUI-библиотека, реализованная на языке C++, также проект содержит набор классов C++ с богатыми возможностями (высокоуровневые структуры данных и др.). А wxPython в свою очередь предлагает «обертки» для классов wxWidgets связанных с GUI.

Примечание: Графическая система wxWidgets декларирует только мульти-платформенный API верхнего уровня, а для непосредственного отображения используется низкоуровневый

(native) API для конкретной платформы. Основная задача библиотеки - это обеспечить работоспособность приложений, написанных с её помощью, на различных операционных системах. Поэтому параллельно существуют несколько различных реализаций wxWidgets: wxGTK, wxMSW, wxMac и т.д. WxWidgets — это более высокоуровневая абстракция, чем графические базисы, над которыми она надстраивается.

Библиотека wxWidgets не настолько популярна в Linux, как GTK+ или Qt, потому всё необходимое для использования wxPython потребуется установить отдельно:

```
$ aptitude search python-wx
p   python-wxglade      - GUI designer written in Python with wxPython
p   python-wxgtk2.8     - wxWidgets Cross-platform C++ GUI toolkit (wxPython binding)
p   python-wxgtk2.8-dbg - wxWidgets Cross-platform C++ GUI toolkit (wxPython binding,
debug version)
p   python-wxmpl        - Painless matplotlib embedding in wxPython
p   python-wxtools      - wxWidgets Cross-platform C++ GUI toolkit (wxPython common...
p   python-wxversion    - wxWidgets Cross-platform C++ GUI toolkit (wxPython version...
```

Достаточно будет установить пакет python-wxgtk2, и все остальные пакеты он инсталлирует самостоятельно, по зависимостям:

```
$ sudo apt-get install python-wxgtk2.8
...
```

Примечание: В RPM дистрибутивах Linux требуемый пакет будет называться wxPython:

```
$ dnf list wxPython*
Установленные пакеты
wxPython                  3.0.2.0-8.fc23                                  @updates
Доступные пакеты
wxPython-devel.i686       3.0.2.0-8.fc23                                  updates
wxPython-devel.x86_64     3.0.2.0-8.fc23                                  updates
wxPython-docs.noarch      3.0.2.0-8.fc23                                  updates
```

В листинге показано простое приложение, использующее wxPython (см. файл GUI/wxPython/bare.py):

Пример wxPython-приложения:

```
import wx

class App( wx.App ):

    def OnInit(self):
        frame = wx.Frame( parent=None, title='Bare' )
        frame.Show()
        return True

app = App()
app.MainLoop()
```

В этом примере создаётся пустое окно GUI-приложения, которое под контролем оконного менеджера Linux может перемещаться, менять размер, сворачиваться и разворачиваться во весь экран:

В следующем листинге показано ещё одно wxPython-приложение, использующее возможности API `wxPython` в терминологии классов Python. Это приложение позволяет из Python-кода перенаправить потоки SYSOUT и SYSERR в созданное окно.

Перенаправление потоков вывода (файл `GUI/wxPython/sysout.py`):

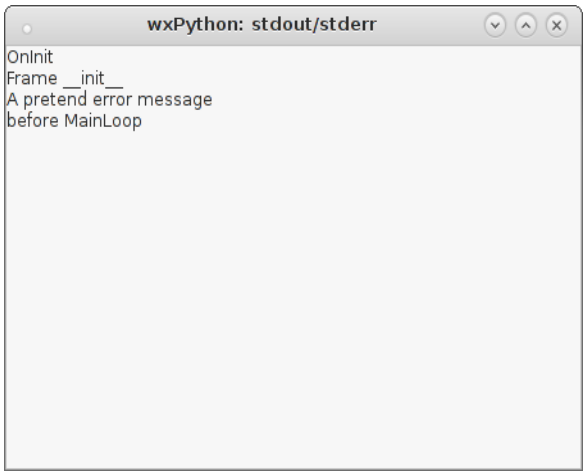
```
import wx
import sys

class Frame(wx.Frame):

    def __init__(self, parent, id, title):
        print "Frame __init__"
        wx.Frame.__init__(self, parent, id, title)

class App(wx.App):

    def __init__(self, redirect=True, filename=None):
        print "App __init__"
        wx.App.__init__(self, redirect, filename)

    def OnInit(self):
        print "OnInit"                          # вывод в поток stdout
        # создание окна
        self.frame = Frame(parent=None, id=-1, title='Startup')
        self.frame.Show()
        self.SetTopWindow(self.frame)
        print >> sys.stderr, "A pretend error message"  # вывод в поток stderr
        return True

    def OnExit(self):
        print "OnExit"

if __name__ == '__main__':
    app = App(redirect=True) # здесь происходит перенаправление вывода
    print "before MainLoop"  # здесь запускается основной цикл обработки событий
    app.MainLoop()
```

```
    print "after MainLoop"
```

Запустим приложение и посмотрим, какие сообщения `print` будут отправлены в **окно** приложения, а какие — в **терминал** запуска:

```
$ python sysout.py
App __init__
after MainLoop
```

На рисунке показано окно при работе приложения:



Примечание. Все примеры, показанные в этой статье и в руководстве для `wxPython`, используют синтаксис Python 2. Это объясняется тем, что только в середине 2012 года создатель `wxPython` Робин Данн (Robin Dunn) в своём блоге сообщал только об **экспериментах** по сборке `wxPython` для Python 3. На сайте проекта в августе 2013 г, в качестве последних стабильных версий указывались реализации для Python 2.7.

Высказываются мнения, что реализация `wxPython` могла бы стать стандартной GUI-библиотекой для Python, если бы эта позиция раньше уже не была занята `Tkinter`. Тем не менее `WxPython` используется во многих открытых проектах.

Pygame

В отличие от ранее рассмотренных библиотек, предназначенных для создания прикладных GUI программ, `Pygame` — это инструмент для реализации разнообразных игровых стратегий внутри графических окон. Поэтому, помимо инструментов для отображения GUI, он содержит и некоторые средства для рисования в этих окнах, создания анимации и ограниченные возможности для воспроизведения аудио-потоков.

Этот пакет не относится к стандартной библиотеки модулей Python, поэтому его потребуется установить отдельно:

```
$ sudo dnf install pygame
...
Зависимости разрешены.
================================================================================
 Package     Архитектура          Версия               Репозиторий            Размер
================================================================================
Установка:
 SDL_mixer      x86_64         1.2.12-10.fc23          fedora                  97 k
 SDL_ttf        x86_64         2.0.11-7.fc23           fedora                  27 k
```

```
 fluidsynth     x86_64         1.1.6-6.fc23            fedora                  29 k
 libmikmod      x86_64         3.3.8-1.fc23            updates                152 k
 pygame         x86_64         1.9.1-19.fc23.20150926  fedora                 3.2 M
Результат операции
================================================================================
Установка  5 Пакетов
Объем загрузки: 3.5 M
Объем изменений: 9.0 M
...
Установлено:
...
pygame.x86_64 1.9.1-19.fc23.20150926
Выполнено!
```

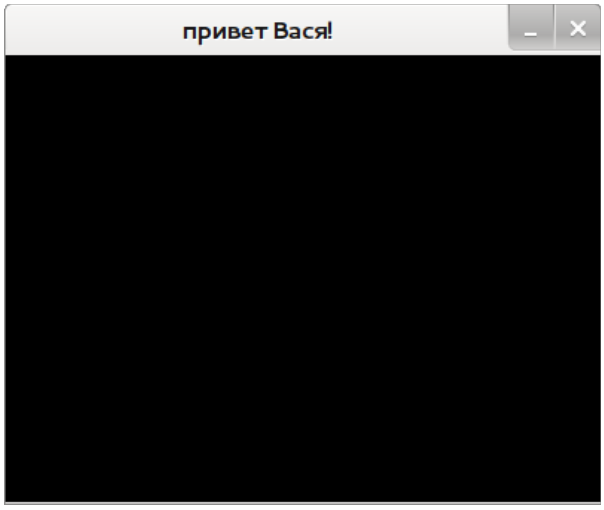
Примечание. Из-за указанных зависимостей, инсталляция данного пакета приведёт к последующей установке большого числа зависимых пакетов. Но это типовое поведение для всех GUI-инструментов, так как они используют много дополнительных пакетов (главным образом, библиотек).

В листинге представлен простейший пример использования PyGame (файл GUI/Pygame/hwpg.py):

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import pygame, sys
from pygame.locals import *

pygame.init()
DISPLAYSURF = pygame.display.set_mode((400, 300))
pygame.display.set_caption( 'привет Вася!' )

while True: # main game loop
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

Здесь в очередной раз демонстрируется классическая схема GUI-приложения, так явно видно как поступившие события, в порядке очереди, обрабатываются в цикле. Запустим приложение в среде Python 2 (автор нигде не встречал упоминаний о использовании Pygame с Python 3). Результат должен выглядеть как на рисунке (здесь не делается ничего сверх создания окна и присвоения ему заголовка).

```
$ python hwpg.py
```

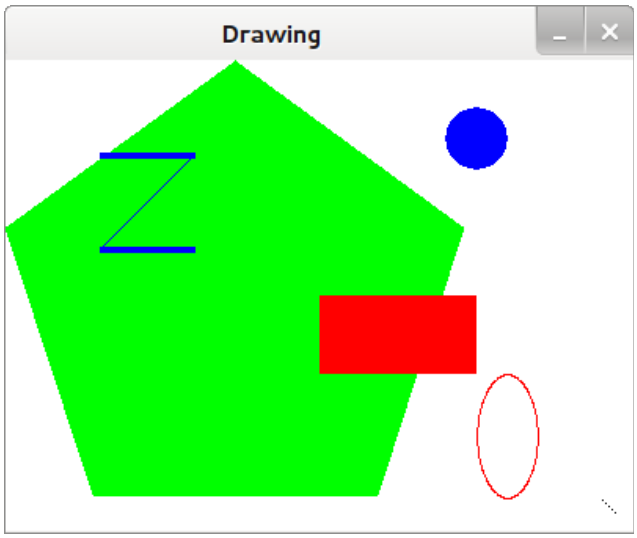


В следующем листинге показан пример рисования в поле окна, точно также рисовать можно не только в самом окне, но и в разнообразных оконных фреймах (файл `GUI/Pygame/drawing.py`):

```
import pygame, sys
from pygame.locals import *

pygame.init()

DISPLAYSURF = pygame.display.set_mode((400, 300), 0, 32)   # настройка окна
pygame.display.set_caption('Drawing')

BLACK = (  0,   0,   0)    # определение цветов
WHITE = (255, 255, 255)
RED   = (255,   0,   0)
GREEN = (  0, 255,   0)
BLUE  = (  0,   0, 255)

DISPLAYSURF.fill(WHITE)    # рисование
pygame.draw.polygon(DISPLAYSURF, GREEN, ((146, 0), (291, 106), (236, 277),
                                         (56, 277), (0, 106)))
pygame.draw.line(DISPLAYSURF, BLUE, (60, 60), (120, 60), 4)
pygame.draw.line(DISPLAYSURF, BLUE, (120, 60), (60, 120))
pygame.draw.line(DISPLAYSURF, BLUE, (60, 120), (120, 120), 4)
pygame.draw.circle(DISPLAYSURF, BLUE, (300, 50), 20, 0)
pygame.draw.ellipse(DISPLAYSURF, RED, (300, 200, 40, 80), 1)
pygame.draw.rect(DISPLAYSURF, RED, (200, 150, 100, 50))

pixObj = pygame.PixelArray(DISPLAYSURF)
pixObj[380][280] = BLACK
pixObj[382][282] = BLACK
pixObj[384][284] = BLACK
pixObj[386][286] = BLACK
```

```
pixObj[388][288] = BLACK
del pixObj

while True:      # основной цикл обработки событий
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

На рисунке показан результат запуска этого приложения:



Также PyGame, как инструмент для написания игровых приложений, обладает ограниченной функциональностью для работы с некоторыми распространёнными аудио-форматами (запуск, остановка, фоновое воспроизведение). Например, вот как можно воспроизвести `.wav`-файл:

```
import sys
import pygame
import time

file = ( lambda : ( len( sys.argv ) > 1 and str( sys.argv[ 1 ] ) ) or 'badswap.wav' )()
pygame.init()
soundObj = pygame.mixer.Sound( file )
soundObj.play()
time.sleep( 1 )   # проигрывать звуковой файл в течение 1 секунды.
soundObj.stop()
```

В простых сценариях подобная аудио-функциональность может оказаться востребованной. Но в более сложных случаях для подобных задач стоит использовать специализированные свободные проекты, например, SoX, Ogg, Vorbis, Speex, FLAC и их кодеки.

Автоматизация GUI-скриптов с помощью `PyZenity`

Иногда приложению требуется простейший GUI-интерфейс, например, набор некоторых диалоговых окон, в которых запрашивается имя пользователя и пароль или выбирается имя файла. Это трудно назвать программами, приложениями, скорее это — некоторые достаточно простые скрипты. В подобных сценариях возможности описанных ранее библиотек являются избыточными. Но специально для задач подобного рода существует

консольная утилита `Zenity`, предназначенная для формирования GUI-интерфейсов из языка командного интерпретатора. Конечно же, к ней существует интерфейс и из Python, реализуемый в проекте `PyZenity`.

Архив с модулем `PyZenity` необходимо скачать с сайта автора этого модуля (см. ссылку в библиографии, в конце текста, этот архив включен в каталог примера) и установить библиотеку в Python 2:

```
$ tar -xzvf PyZenity-0.1.7.tar.gz
PyZenity-0.1.7/
PyZenity-0.1.7/PyZenity.py
PyZenity-0.1.7/PKG-INFO
PyZenity-0.1.7/setup.py
$ cd PyZenity-0.1.7
# sudo python setup.py install
running install
running build
running build_py
running install_lib
copying build/lib.linux-i686-2.7/PyZenity.py -> /usr/local/lib/python2.7/dist-packages
byte-compiling /usr/local/lib/python2.7/dist-packages/PyZenity.py to PyZenity.pyc
running install_egg_info
Writing /usr/local/lib/python2.7/dist-packages/PyZenity-0.1.7.egg-info
```

Этот же архив можно использовать и для Python 3:

```
# sudo python3 setup.py install
...
```

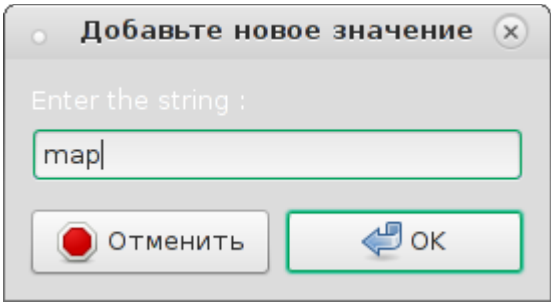
После установки мы сможем создать простейший GUI скрипт (см. файл `GUI/PyZenity/enter.py`):

```
import PyZenity
a = PyZenity.GetText( text="Enter the string : ", entry_text="", password=False )
print( a )
```

И запустить его:

```
$ python enter.py
...
map
```

В результате получим такое вот окно:



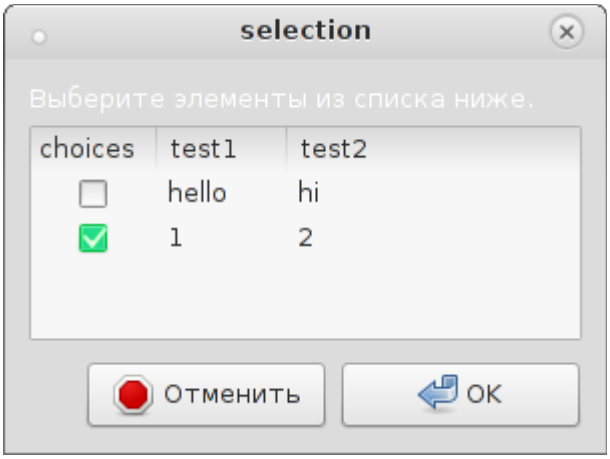
Текстовая строка, вводимая пользователем в этом диалоговом окне, поступит (после «OK») в вызывающий скрипт на Python (что мы и видим в терминале).

Также можно подготовить сценарий для выбора некоторого значения из списка нескольких предложенных (см. файл `GUI/PyZenity/select.py`).

```
import PyZenity

x = PyZenity.List( ["choices","test1","test2"], title="selection",
                   boolstyle="checklist",
                   editable=False, select_col="ALL", sep='|',
                   data=[["","hello","hi"],["","1","2"]])
print( x )
```

Результат запуска этого приложения показан на рисунке:

```
$  python select.py
...
['1', '2']
```



Дополнительные инструменты

Онлайн документирование

В Linux инфраструктуре Python есть такое замечательное средство как онлайновый документатор pydoc, входящий в стандартную установку Python, позволяющий генерировать справочную документацию по установленным в вашей системе пакетам Python (а значит и встроенным терминам и ключевым словам Python). Изучите это:

```
$ pydoc -h
pydoc - the Python documentation tool

pydoc <name> ...
    Show text documentation on something.  <name> may be the name of a
    Python keyword, topic, function, module, or package, or a dotted
    reference to a class or function within a module or module in a
    package.  If <name> contains a '/', it is used as the path to a
    Python source file to document. If name is 'keywords', 'topics',
    or 'modules', a listing of these things is displayed.

pydoc -k <keyword>
    Search for a keyword in the synopsis lines of all available modules.
```

```
pydoc -p <port>
    Start an HTTP server on the given port on the local machine.  Port
    number 0 can be used to get an arbitrary unused port.

pydoc -w <name> ...
    Write out the HTML documentation for a module to a file in the current
    directory.  If <name> contains a '/', it is treated as a filename; if
    it names a directory, documentation is written for all the contents.
```

Например так:

```
$ pydoc range
Help on built-in function range in module __builtin__:

range(...)
    range(stop) -> list of integers
    range(start, stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.
```

Примечательно, что `pydoc` **извлекает** информацию из установленных в вашей системе пакетов, а это означает, что:

- устанавливая в системе **новый** пакет Python, например `NumPy`, вы получаете справочную систему, в том числе, и по установленному пакету;
- справочная информация `pydoc` относится всегда **к актуальным версиям** пакетов, установленных в вашей системе, что делает её, в некотором смысле, интереснее, чем многочисленные справочные ресурсы в Интернет;

Утилита `pydoc` может генерировать справочный документ в HTML формате. Но самой замечательной его особенностью является то, что она может работать как HTTP-сервер, что позволяет дальше открыть сколько угодно вкладок в любом WEB-браузера и просматривать документацию прямо по ходу написания своего кода. В таком режиме программа запускается так:

```
$ pydoc -p 40000 &
[1] 26668
pydoc server ready at http://localhost:40000/
```

Если вас интересует Python 3, то для него есть такая же утилита:

```
$ ls -l /usr/bin/py*doc*
-rwxr-xr-x 1 root root 78 сен 29  2016 /usr/bin/pydoc
lrwxrwxrwx 1 root root  8 авг  9  2016 /usr/bin/pydoc3 -> pydoc3.4
-rwxr-xr-x 1 root root 78 авг  9  2016 /usr/bin/pydoc3.4
```

Как понятно, вы можете запустить **одновременно** справочные системы по версиям 2 и 3, запустив их как сервера на разных TCP портах.

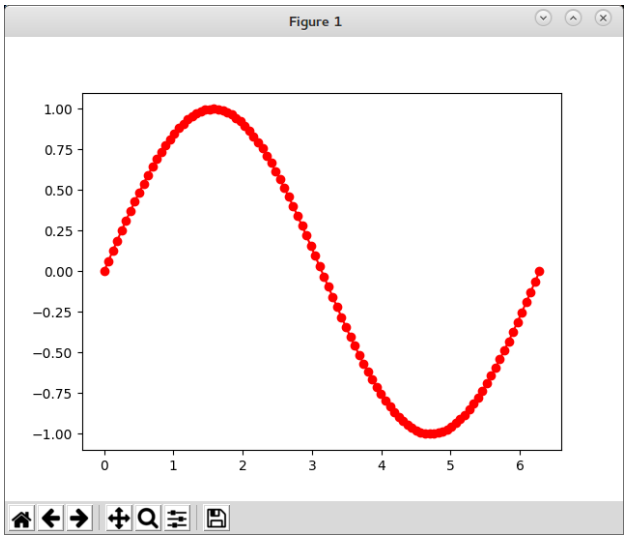
Научная графика в Python

Графическое представление данных в самых разнообразных формах обеспечивает такой пакет как `Matplotlib`, который, кроме исключительно полно описан в документации, и, кроме того, существует перевод этой документации с примерами использования (см. ссылки в библиографии). Рассмотрим только простейший пример, который понятен и в пояснениях не нуждается (файл `analys_data/matplotlib/gsin.py`):

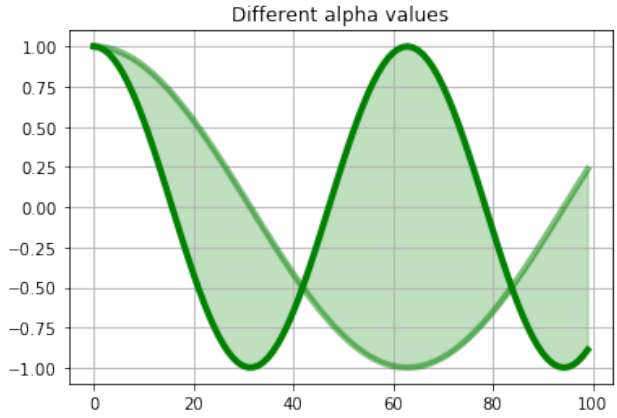
```
# -*- coding: utf-8 -*-
import sys
import numpy as np
from math import *
import matplotlib.pyplot as plt

def main():
   x = np.linspace( 0, 2 * pi, 100 )
   y = np.sin( x ) # numpy.sin() но не math.sin() !
   plt.plot( x, y, 'ro-' )
   plt.show()

if __name__ == '__main__':
    try:
        main()
    except Exception as exc:
        sys.stderr.write( u"{}\n{}\n".format( -1, exc ) )
        sys.exit( 1 )
```

Выполнение:

```
$ python gsin.py
```



Это совершенно примитивное изображение, показанное для упрощения иллюстрирующего кода. Но этот пакет позволяет создавать самые изощрённые графические представления … типа вот такого (из числа многих показанных в упоминавшемся переводе документации):



Машинное обучение и обработка данных

Здесь я имею в виду пакет Scikit-learn :

David Cournapeau изначально разработал библиотеку scikit-learn в рамках проекта Google Summer of Code в 2007 году. Позднее Matthieu Brucher присоединился к проекту и использовал scikit-learn в своей дипломной работе. В 2010 году INRIA подключилась к работе над библиотекой и первая версия (v0.1 beta) была выпущена в конце января 2010.

```
$ dnf info python-scikit-learn
Доступные пакеты
Имя         : python-scikit-learn
Архитектура : x86_64
Эпоха       : 0
Версия      : 0.17
Релиз       : 1.fc23
Размер      : 3.9 M
Репозиторий : updates
Краткое опи : Machine learning in Python
URL         : http://scikit-learn.org/
Лицензия    : BSD
Описание    : Scikit-learn integrates machine learning algorithms in the tightly-knit
            : scientific Python world, building upon numpy, scipy, and matplotlib.
            : As a machine-learning module, it provides versatile tools for data
            : mining and analysis in any field of science and engineering. It strives
            : to be simple and efficient, accessible to everybody, and reusable in
            : various contexts.
```

Пакет этот, естественно, потребует отдельной установки:

```
$ sudo dnf install python-scikit-learn
...
Установка  6 Пакетов
Объем загрузки: 21 M
Объем изменений: 67 M
...
Установлено:
blas.x86_64 3.5.0-12.fc23
```

```
lapack.x86_64 3.5.0-12.fc23
numpy-f2py.x86_64 1:1.9.2-2.fc23
python-scikit-learn.x86_64 0.17-1.fc23
python2-joblib.noarch 0.9.3-2.fc23
scipy.x86_64 0.14.1-1.fc22
```

Хотя пакет назван «машинное обучение», но он гораздо шире и включает множество алгоритмов, необходимых в обработке данных.

Линейная регрессия

Линейная регрессия — общеизвестный способ приближения данных линейной функцией, минимизирующий среднеквадратичное отклонение от исходных данных. Сделаем пример …, но линейная одномерная регрессия — это слишком просто, поэтому будем делать приближение экспериментальных данных от **нескольких** переменных. Сначала проверим регрессионное приближение в ручном режиме:

```
$ python
Python 2.7.11 (default, Sep 29 2016, 13:33:00)
[GCC 5.3.1 20160406 (Red Hat 5.3.1-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from sklearn import linear_model
>>> clf = linear_model.LinearRegression()
>>> X = [[0, 0], [0, 1], [1, 0]]
>>> Y = [2, 5, 3]
>>> clf.fit( X, Y )
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
>>> clf.coef_
array([ 1.,  3.])
>>> clf.predict( X )
array([ 2.,  5.,  3.])
>>>
```

Напишем приложение, которое даёт регрессионное приближение линейной функцией от 3-х переменных экспериментальных данных (файл `analys_data/analys/lr.py`), зашумленных случайными данными (так, что соотношение сигнал/шум можно менять и наблюдать зависимость погрешностей восстановления данных):

```
# -*- coding: utf-8 -*-
import sys
import math
from sklearn import linear_model
from random import gauss

def func( x ) :                                 # линейная функция 3-х переменных
   f = 0.0
   for i in range( 0, n ) :
      f += k[ i ] * x[ i ]
   return f
r = 0.05                                        # отношение шум/сигнал
k = [ 0.33, 1.0, 2.5 ]
```

```
n = len( k )
x = [] ; y = []
if len( sys.argv ) > 1 : r = float( sys.argv[ 1 ] )
print( 'коэффициенты исходные: {}'.format( k ) )
for x1 in range( 0, n ) :
   for x2 in range( 0, n ) :
      for x3 in range( 0, n ) :
         x.append( ( float( x1 + 1 ), float( x2 + 1 ), float( x3 + 1 ) ) )
for z in x : y.append( func( z ) )
m = reduce( lambda x, y : x + y, y ) / len( y )  # среднее значение
if len( sys.argv ) > 2 : print( y )
for i in range( 0, len( y ) ) :
   y[ i ] += gauss( 0, m * r )
v = reduce( lambda x, y : x + y, y ) / len( y )  # среднее значение
regr = linear_model.LinearRegression()           # регрессионный объект
regr.fit( x, y )                                 # тренировать модель
print( 'коэффициенты регрессии: {}'.format( regr.coef_ ) )
z = regr.predict( x )
if len( sys.argv ) > 2 : print( z )
d = 0.0
for i in range( 0, len( y ) ) :
   d += ( y[ i ] - z[ i ] ) * ( y[ i ] - z[ i ] )
print( 'СКО восстановления = {}'.format( math.sqrt( d / len( y ) ) ) )
```

И вот как это выполняется:

```
$ python lr.py
коэффициенты исходные: [0.33, 1.0, 2.5]
коэффициенты регрессии: [ 0.37248396  0.90390247  2.34488441]
СКО восстановления = 0.353896669512
```

Мы можем изменять уровень зашумленности данных и наблюдать зависимость погрешности восстановления данных в зависимости от этого параметра:

```
$ python lr.py .1
коэффициенты исходные: [0.33, 1.0, 2.5]
коэффициенты регрессии: [ 0.4758855   1.20324846  2.22633023]
СКО восстановления = 0.695637600137


$ python lr.py .3
коэффициенты исходные: [0.33, 1.0, 2.5]
коэффициенты регрессии: [ 0.42850093  0.7844652   2.60845126]
СКО восстановления = 2.4273269973
```

Многомерная оптимизация

Оптимизация (нахождение минимумов или максимумов, и желательно глобальных) многомерных нелинейных (общего вида) функций нескольких переменных — очень сложная алгоритмически задача, которой посвящены целые книги. Для решения этой задачи предложено достаточно много (десятки) совершенно отличающихся алгоритмов.

Окончательная отработка численных алгоритмов нелинейной многомерной оптимизации сложилась только к концу 90-х годов.

Пакет SciPy предлагает на выбор много как градиентных (со знанием производных целевой функции), там и безградиентных алгоритмов. В качестве алгоритма оптимизации (параметра `method`) пакет предлагает (из документации):

```
method : str or callable, optional
   Type of solver. Should be one of
     - 'Nelder-Mead'
     - 'Powell'
     - 'CG'
     - 'BFGS'
     - 'Newton-CG'
     - 'Anneal (deprecated as of scipy version 0.14.0)'
     - 'L-BFGS-B'
     - 'TNC'
     - 'COBYLA'
     - 'SLSQP'
     - 'dogleg'
     -'trust-ncg'
     - custom – a callable object (added in version 0.14.0)
```

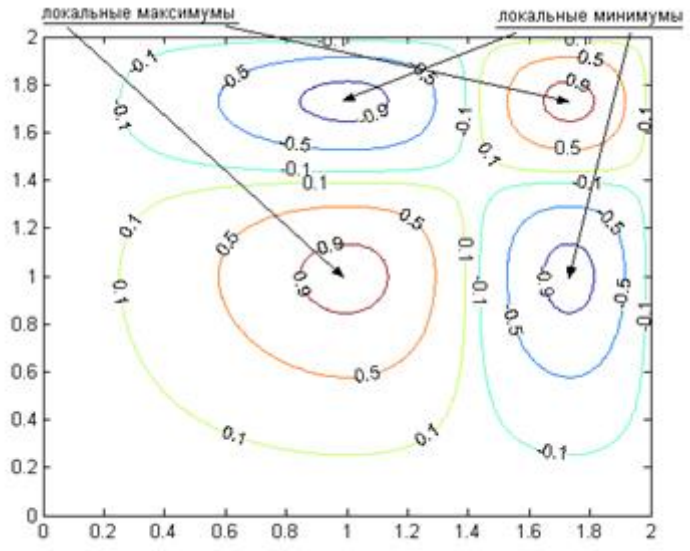
Примечание: Модуль SciPy у нас установился по зависимостям при установке пакета `Scikit-learn`, показанного ранее Хотя его, естественно, можно установить и автономно. Он часто используется совместно с модулем NumPy.

В библиотеке SciPy есть ещё множество интереснейших вещей, таких как (и это ещё не всё):

- Специальные математические функции (`scipy.special`);
- Фурье преобразования (`scipy.fftpack`);
- Цифровая сигнальная обработка (DSP — `scipy.signal`);
- Линейная алгебра (`scipy.linarg`);
- Статистика (`scipy.stats`);

Но мы не будем их рассматривать, за исключением задач оптимизации, всё остальное легко найти в объёмной документации библиотеки.

Для начала мы в ручном режиме поищем минимум такой «хитрой» функции 2-х переменных как $F(x1, x2) = \sin(\pi / 2 * x1^2) * \sin(\pi / 2 * x2^2)$, график поведения которой показан на рисунке:



Начнём поиск с точки (1.4, 1.4), используем метод Нелдера-Мида, известный ещё как метод деформируемого симплекса:

```
$ python
Python 2.7.11 (default, Sep 29 2016, 13:33:00)
[GCC 5.3.1 20160406 (Red Hat 5.3.1-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import sin, pi
>>> sin2 = lambda x: sin( pi / 2 * x[ 0 ] ** 2 ) * sin( pi / 2 * x[ 1 ] ** 2 )
>>> X = [ ( 1, 1 ), ( 1, 1.7 ), ( 1.7, 1 ), ( 1.7, 1.7 ) ]
>>> Y = [ sin2( x ) for x in X ]
>>> print( Y )
[1.0, -0.9851093261547738, -0.9851093261547738, 0.9704403844771124]
>>> from scipy.optimize import minimize
>>> res = minimize( sin2, ( 1.4, 1.4 ), method='Nelder-Mead' )
>>> print(  res.x )
[ 1.73201465  1.00003747]
>>> print( res.fun )
-0.999999973722
>>> print( '{} -> {}'.format( res.nit, res.nfev ) )
36 -> 71
>>>
```

Найден один из минимумов (минимумы симметричны относительно x1 и x2) с значением -1 в точке (1.732, 1.000), при этом было произведено 36 циклов итераций, в ходе которых значения целевой функции вычислялись 71 раз.

Все оптимизационные методы модуля возвращают объект класса `OptimizeResult` … структуру которого мы можем подсмотреть в онлайн-справке `pydoc` (**http://localhost:40000/scipy.optimize.html#-minimize**):

```
class OptimizeResult()
Represents the optimization result.
Attributes
----------
x : ndarray
```

```
   The solution of the optimization.
success : bool
   Whether or not the optimizer exited successfully.
status : int
   Termination status of the optimizer. Its value depends on the
   underlying solver. Refer to `message` for details.
Message : str
   Description of the cause of the termination.
fun, jac, hess, hess_inv : ndarray
   Values of objective function, Jacobian, Hessian or its inverse (if
   available). The Hessians may be approximations, see the documentation
   of the function in question.
nfev, njev, nhev : int
   Number of evaluations of the objective functions and of its
   Jacobian and Hessian.
nit : int
   Number of iterations performed by the optimizer.
maxcv : float
   The maximum constraint violation.
```

Теперь предыдущий пример мы можем записать в виде кода приложения (файл `analys_data/analys/min.py`):

```
# -*- coding: utf-8 -*-
from math import pi, sin
import matplotlib.pyplot as plt
from scipy.optimize import minimize


sin2 = lambda x: sin( pi / 2 * x[ 0 ] ** 2 ) * sin( pi / 2 * x[ 1 ] ** 2 );
X = [ ( 1, 1 ), ( 1, 1.7 ), ( 1.7, 1 ), ( 1.7, 1.7 ) ]
print( X )
Y = [ sin2( x ) for x in X ]
print( Y )
res = minimize( sin2, ( 1.4, 1.4 ), method='Nelder-Mead' )
print(  res.x )
print( res.fun )
print( '{} -> {}'.format( res.nit, res.nfev ) )
```

Выполняем:

```
$ python min.py
[(1, 1), (1, 1.7), (1.7, 1), (1.7, 1.7)]
[1.0, -0.9851093261547738, -0.9851093261547738, 0.9704403844771124]
[ 1.73201465  1.00003747]
-0.999999973722
36 -> 71
```

Нелинейная аппроксимация данных

Есть ещё одна задача из области обработки экспериментальных данных, которая не разрешима, в принципе, без (или до реализации) алгоритмов нелинейной оптимизации:

- Пусть у нас есть некоторая среда, состояние которой зависит от M параметров A[…], которые нужно измерить…
- И мы знаем характер функции зависимости некоторого измеряемого параметра среды Y = F() от нескольких переменных X[…] (в частном случае N может быть 1, переменная X — это шкала времени, а X[...] — это временной ряд измеренных значений Y(t))
- Но, зная численные значения Y[…] = F(), мы не можем по значениям F() восстановить массив параметров измеряемой среды … возможно, в силу сложности вида функции F() и невозможности решения для неё обратной задачи.

Это очень и очень частая задача … вообще то это общая постановка задачи в теории измерений. Но эта задача имеет общее **численное** решение: рассматривая зависимость F() от M измеряемых параметров как функцию M переменных F(A[1], A[2], … A[M]) — **минимизировать среднеквадратичное отклонение** всех измеренных точек Y[…] от предсказываемых функцией F(). Искомый вектор A и будет набором параметров среды.

Примечание: Это продолжение идеи линейной регрессии, известное в математике столетиями, но расширенное от линейной аппроксимации на **функции общего вида** произвольной сложности. Ничего подобного нельзя было предложить без соответствующих численных методов и адекватных вычислительных мощностей.

Продемонстрируем это примером и кодом, что станет гораздо понятнее (файл `analys_data/analys/exdata.py`):

```
# -*- coding: utf-8 -*-
import sys
from math import exp, sin, pi, sqrt
from random import gauss
from scipy.optimize import minimize

#------------------------------------ модедирование ----------------------------
a = [ 1.0, 2, .7, 4., .2 ]                      # 4 параметра
x = [] ; y = []
r = 0.05                                        # отношение шум/сигнал
opfunc = lambda k, x : k[ 0 ] + \
                       k[ 1 ] * exp( -k[ 2 ] * x[ 0 ] ) + \
                       k[ 3 ] * sin( 2. * pi * k[ 4 ] * x[ 1 ] )

print( 'коэффциенты модели: {}'.format( a ) )
if len( sys.argv ) > 1 : r = float( sys.argv[ 1 ] )
for x1 in range( 0, 4 ) :
   for x2 in range( 0, 5 ) :
      x.append( ( float( x1 ), float( x2 ) ) )
for z in x :
   y.append( opfunc( a, z ) )
m = reduce( lambda x, y : x + y, y ) / len( y )  # среднее значение
```

```
for i in range( 0, len( y ) ) :
   y[ i ] += gauss( 0, m * r )
#---------------------------------- восстановление -----------------------
def delt( a ) :
   global d
   d = [ opfunc( a, z ) for z in x ]
   sum = 0.0
   for i in range( 0, len( y ) ) :
      sum += ( d[ i ] - y[ i ] ) ** 2
   return sum

b = ( 3.0, 1.0, 2.5, 2.0, .3 )
print( 'начальное приближение: {}'.format( b ) )
res = minimize( delt, b, method='Nelder-Mead' )
print( 'восстановленные коэффциенты: {}'.format( res.x ) )
print( 'рассогласование данных = {}'.format( sqrt( res.fun / len( y ) ) ) )
print( 'итераций {} -> вычислений функции {}'.format( res.nit, res.nfev ) )
```

Измеряемый параметр имеет функциональную зависимость (достаточно произвольно выбранную нами для иллюстрации, но непростую в вычислительном смысле): y = opfunc(x1, x2) = a0 + a1 * exp(-a2 * x1) + a3 * sin(2. * pi * a4 * x2) …. По зашумленным данным (с разной интенсивностью — переменная r) восстанавливаются коэффициенты a[…]:

```
$ python exdata.py
коэффциенты модели: [1.0, 2, 0.7, 4.0, 0.2]
начальное приближение: (3.0, 1.0, 2.5, 2.0, 0.3)
восстановленные коэффциенты: [0.9452891  2.1316498  0.69098432  3.98081858  0.1993641]
рассогласование данных = 0.093108050073
итераций 418 -> вычислений функции 686

$ python exdata.py .1
коэффциенты модели: [1.0, 2, 0.7, 4.0, 0.2]
начальное приближение: (3.0, 1.0, 2.5, 2.0, 0.3)
восстановленные коэффциенты: [1.10580162  1.7768491  0.83597573  4.0930248   0.196962]
рассогласование данных = 0.166063658543
итераций 462 -> вычислений функции 734

$ python exdata.py .01
коэффциенты модели: [1.0, 2, 0.7, 4.0, 0.2]
начальное приближение: (3.0, 1.0, 2.5, 2.0, 0.3)
восстановленные коэффциенты: [ 1.0207835  1.97788697  0.7111184  4.00236494  0.2002597]
рассогласование данных = 0.0179642252942
итераций 477 -> вычислений функции 759
```

Для удовлетворительной аппроксимации данных требуется близкое к 1000 раз число вычислений достаточно непростой целевой функции. Очевидно, что никакое подобное приближение не может быть достигнуть никакими ручными способами счёта.

Онлайн ресурсы и литература

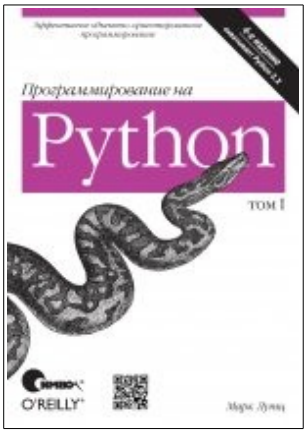
1. Марк Лутц, «Программирование на Python», 4-е издание, Спб. Символ-Плюс

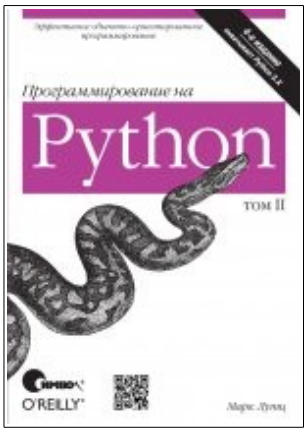
Том 1, 992 страницы, сентябрь 2011, ISBN: 978-5-93286-210-0

https://www.books.ru/books/programmirovanie-na-python-4-e-izdanie-i-tom-841171/

Том 2, 992 страницы, октябрь 2011, ISBN: 978-5-93286-211-7

https://www.books.ru/books/programmirovanie-na-python-4-e-izdanie-ii-tom-1532378/





Учебный курс для начального ознакомления. Можно свободно скачать здесь:

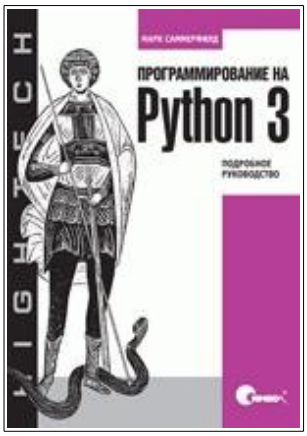
http://www.proklondike.com/books/python/lutcz_progr_python_tom1_2011.html

http://www.proklondike.com/books/python/lutcz_progr_python_tom2_2011.html

2. Марк Саммерфилд, Программирование на Python 3. Подробное руководство, Спб. Символ-Плюс, 608 страниц, июнь 2009, ISBN: 978-5-93286-161-5

https://www.books.ru/books/programmirovanie-na-python-3-podrobnoe-rukovodstvo-661501/

Это одна из лучших книг по Python, но рассчитана на программистов с опытом, для начального ознакомления она не годится. К книге прилагается архив интересных и сложных примеров.

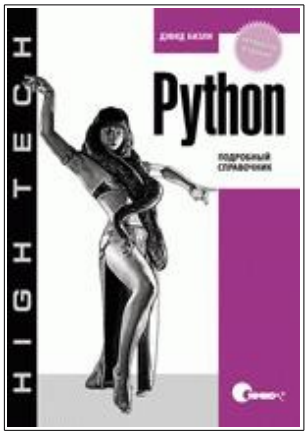


Книгу вместе с архивом примеров можно свободно скачать здесь:

https://rutracker.org/forum/viewtopic.php?t=3454925

3. Дэвид М. Бизли, Python. Подробный справочник, 4-е издание, Спб. Символ-Плюс, 864 страницы, ноябрь 2012, ISBN: 978-5-93286-157-8

https://www.books.ru/books/python-podrobnyi-spravochnik-4-e-izdanie-784182/



Справочник по Python, который всегда нужно держать под рукой при написании своего кода.

Можно свободно скачать здесь:

https://rutracker.org/forum/viewtopic.php?t=3454925

4. The Python Standard Library — онлайн справочник по библиотеке Python с возможностью выбора используемой версии: http://docs.python.org/3.3/library/index.html

5. Девид Мертц, Очаровательный Python: Функциональное программирование на языке Python, Часть 1:

http://www.ibm.com/developerworks/ru/library/l-prog/

6. Девид Мертц, Очаровательный Python: Еще о функциональном программировании на Python, Часть 2 :

http://www.ibm.com/developerworks/ru/library/l-prog2/

7. David Mertz, Charming Python: Functional programming in Python, Part 3 :

http://www.ibm.com/developerworks/linux/library/l-prog3/index.html

8. Peter Norvig, Python for Lisp Programmers : http://www.norvig.com/python-lisp.html

9. Дэвид Бизли, Как устроен GIL в Python - http://habrahabr.ru/post/84629/

10. ctypes — A foreign function library for Python : http://docs.python.org/3/library/ctypes.html

11. Boost 1.53.0 Library Documentation - http://www.boost.org/doc/libs/1_53_0/

12. Boost.Python : http://www.boost.org/doc/libs/1_53_0/libs/python/doc/index.html

13. About Cython : http://cython.org/

14. Документация по Cython 0.14 (перевод) : http://kostikvento.ru/cython/docs/index.html

15. Welcome to SWIG : http://www.swig.org/

16. Open source projects using SWIG : http://www.swig.org/projects.html

17. Extending and Embedding the Python Interpreter:
http://docs.python.org/3.3/extending/index.html

18. An Introduction to Tkinter : http://effbot.org/tkinterbook/

19. PyQt : http://ru.wikipedia.org/wiki/PyQt

20. Differences Between PySide and PyQt
http://qt-project.org/wiki/Differences_Between_PySide_and_PyQt

21. Учебник PyGTK 2.0 :
http://pygtk.ru/%D0%A3%D1%87%D0%B5%D0%B1%D0%BD%D0%B8%D0%BA-pygtk-2-0/

22. PyGObject - GLib/GObject/GIO Python bindings : https://wiki.gnome.org/PyGObject

23. Кратко о PyGI :
http://pygtk.ru/2012/02/%D0%9A%D1%80%D0%B0%D1%82%D0%BA%D0%BE-%D0%BE-pygi/

24. wxPython : http: http://www.wxpython.org/

25. WxPython in Action (перевод) :
http://wiki.python.su/%D0%9A%D0%BD%D0%B8%D0%B3%D0%B8/WxPythonInAction

25. Albert Sweigart, «Invent You Own Computer Games with Python» :
http://inventwithpython.com/

26. Albert Sweigart, «Making Games with Python & Pygames» :
http://inventwithpython.com/

27. Zenity - графический интерфейс для командной строки :
http://www.ibm.com/developerworks/ru/library/l-zenity/

28. PyZenity : http://brianramos.com/?page_id=38

29. Play with GUIs using Python : http://www.linuxforu.com/2011/03/quickly-write-gui-using-python/

30. Документируй это (о справочной документации Python) :

http://pyobject.ru/blog/2006/09/08/document-it/

31. Научная графика в Python, перевод Шабанов Павел Александрович :

http://nbviewer.jupyter.org/github/whitehorn/Scientific_graphics_in_python/blob/master/P1%20Chapter%201%20Pyplot.ipynb

32. Ликбез по Matplotlib (Python) : https://edunow.su/site/content/python-matplotlib

33. Введение в scikit-learn : http://igorsubbotin.blogspot.ru/2015/01/intro-to-scikit-learn.html

34. Линейная регрессия в Python (Scikit-learn) :

https://edunow.su/site/content/linear-regression-python-scikit-learn

35. SciPy Tutorial (October 25, 2017): https://docs.scipy.org/doc/scipy/reference/

36. NumPy v1.13 Manual (June 10, 2017) : https://docs.scipy.org/doc/numpy/

37. NumPy Reference (June 10, 2017) : https://docs.scipy.org/doc/numpy/reference/

38. Исходный цикл авторских статей на IBM developerWorks, 2013 г., послуживших основой данного текста:

Тонкости использования языка Python: Часть 1. Версии и совместимость. https://www.ibm.com/developerworks/ru/library/l-python_details_01/

Тонкости использования языка Python: Часть 2. Типы данных. https://www.ibm.com/developerworks/ru/library/l-python_details_02/

Тонкости использования языка Python: Часть 3. Функциональное программирование. https://www.ibm.com/developerworks/ru/library/l-python_details_03/

Тонкости программирования на языке Python: Часть 4. Параллельное исполнение. https://www.ibm.com/developerworks/ru/library/l-python_details_04/

Тонкости использования языка Python: Часть 5. Мульти-платформенные многопоточные приложения. https://www.ibm.com/developerworks/ru/library/l-python_details_05/

Тонкости использования языка Python: Часть 6. Способы интеграции Python и C/C++ приложений. https://www.ibm.com/developerworks/ru/library/l-python_details_06/

Тонкости использования языка Python: Часть 7. Особенности взаимодействия с C++. Пакет distutils, библиотека Boost.Python, проект Cython. https://www.ibm.com/developerworks/ru/library/l-python_details_07/

Тонкости использования языка Python: Часть 8. Особенности взаимодействия с C++. Проект SWIG и обратная интеграция Python в C/C++ приложения. https://www.ibm.com/developerworks/ru/library/l-python_details_08/

Программирование на Python: Часть 9. Разработка GUI-приложений. https://www.ibm.com/developerworks/ru/library/l-python_details_09/

Тонкости использования языка Python: Часть 10. 2D графика и GUI-сценарии. https://www.ibm.com/developerworks/ru/library/l-python_details_10/