ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

(Python Beginning)

Не бывает хороших или плохих языков программирования, бывают лишь плохие программисты, которые плохо подходят как к своей работе, так и к вопросу повышения своей квалификации.

Язык программирования – это инструмент. Если Вы не в состоянии выбрать наиболее подходящий инструмент для решения задачи, или не знаете всех возможностей данного инструмента, то это говорит лишь о вашей низкой квалификации! Нищий в рассказе Марка Твена использовал большую золотую печать для того, чтобы колоть орехи, для лучшего её использования у него не хватало знаний и жизненного опыта…

Урок 2. (Lesson 2) Основы программирования

Вспомним об основной ключевой особенности языка *Python* – динамической типизация. Она обеспечивается за счет нескольких механизмов, одним из которых является ***отказ от традиционного использования переменных имеющих некоторый тип***, и использование переменных–ссылок, которые указывают на объект в памяти. Но, в отличие от *C++*, где ссылка используется в качестве некоего постоянного псевдонима переменной, её второго имени, которое не может быть связано с другой переменной, ссылки в *Python* могут быть изменены, то есть можно изменять эту связь между ссылкой и объектом по ходу выполнения программы…

В *Python*, как и в *C++*, можно создавать анонимные объекты, но в отличие от *C++* в *Python* другой «***механизм сборки мусора***», поэтому ***время жизни*** у *Python*-объектов отличается от времени жизни *C++*-объектов. Имеется возможность удалить ссылку, но оставить объект в памяти. В *Python* даже исполняемый программный код рассматривается как объект, содержащий в себе другие объекты, использующий вызовы методов и передачу сообщения другим объектам. На низкоуровневом подходе к программированию объектам в памяти ставится в соответствие указатель, адрес, а при высокоуровневом подходе – ссылка. Ссылка должна иметь имя. Правила формирования имен в *Python* совпадают с правилами языков *C/C++*: имена переменных-ссылок могут содержать буквы латинского алфавита, арабские (индийские) цифры и символ нижнего подчеркивания. Кроме того, имена переменных-ссылок не могут совпадать с ключевыми словами языка Python. Чтобы получить список ключевых слов, введите в среде Python IDLE следующие инструкции.

```
import keyword  # имортировать модуль keyword
keyword.klist   # получить атрибут модуля klist – keyword list
```

Ниже приведен результат работы данного кода в среде *Python IDLE*.

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with',
'yield']
>>>
```

Для того чтобы сделать сценарий, выводящий на окне терминала такой же список ключевых слов, необходимо в среде *Python IDLE* (далее просто *IDLE*) выбрать команду меню *File*, опцию *New File* (комбнация клавишь *Ctrl - N*) ввести в следующие инструкции.

```
#!/usr/bin/python3
# Filename: keywordlist01.py
"""Программа для вывода списка ключевых слов языка Python"""
import keyword  # имортировать модуль keyword
keyword.klist   # получить атрибут модуля klist – keyword list
input()         # ожидание нажатия клавиши <Enter>
```

Сохраните данный файл на жестком диске. Для запуска данного сценария из оболочки *IDLE* нажмите клавишу *F5*.

Первая строка данного сценария соответствует созданному нами атрибуту документации модуля *__doc__* , доступ к которому возможен программно. Данный механизм ***немного похож*** на средства *javadoc*, но является *Python*-технологией. Не стоит забывать, что языки *Java* и *Python* являются ровесниками и явились ответом на потребность в разработке нового языка общего назначения вместо *C++*. И дело не в сложности *C++*, который предъявляет более высокие требования к уровню квалификации разработчика, основная задача была в необходимости создания инструмента значительно упрощающего процесс разработки ПО, с чем *Java* и *Python* блестяще справились. Да, изначально у этих языков были свои сегменты рынка: *Java* был ориентирован на *ActiveX*-технологии, а *Python* – на ***десктоп-решения***. По мере развития оба языка стали претендовать на универсальности, включают поддержку программирования для Интернет, поддержку клиент-серверных архитектур и возможность работы с микроконтроллерными платформами. Сейчас они конкуренты, хотя мы имеем возможность комбинировать их, используя их технологии одновременно. Но все это зависит только от квалификации разработчика!

Как уже стало очевидно, инструкция *import* используется для включения в модуль информации из других модулей, написанных на языке *Python*. Она похожа на директиву препроцессора #*include* из языка *C*.

Надеюсь, что уже обратили внимание на тот факт, что в конце строки *Python*-выражения не ставится оператор «точка с запятой» ‘;’. Её, конечно, можно использовать, но необходимости в этом нет. Данная операция может быть использована для объединения нескольких выражений, указанных в одной строке.

```
>>> A = 10;
>>> B = 12; C = 13
```

Но вернемся к именам переменных-ссылок. Помимо совпадения с ключевыми словами для них настоятельно рекомендуется избегать совпадения со встроенными идентификаторами, определяющими функциональные средства языка *Python*. Это опять же связано с динамической типизацией, принятой в *Python*.

Рассмотрим следующий пример (инструкции вводятся в окне *IDLE* после символов приглашения >>>, результат работы выводится с новой строки).

```
>>> print("Hello")  # вывод сообщения в стандартный поток вывода
Hello
>>> A = print       # создать ссылку и связать ее с методом print()
>>> A("Hello")      # вызвать метод через ссылку
Hello
>>> print = 2       # использовать ссылку – имя метода как переменную
>>> print           # получить доступ к объекту в памяти
2
>>>
```

После выполнения инструкции `print = 2` доступ к методу в данной программе становится невозможен, ***связь с объектом-методом разрушена*** (***утеряна***)! Это приводит к тому, что дальнейшее привычное использование метода становится невозможным! В подтверждение данных слов рассмотрим следующий пример.

```
>>> print = 2       # использовать ссылку – имя метода как переменную
>>> print           # получить доступ к объекту в памяти
2
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print("Hello")
TypeError: 'int' object is not callable
>>>
```

Ура, получили ***ошибку времени исполнения***, исключение ***TypeError***, которое говорит о том, что целочисленный объект в памяти не может быть ***callable*** (исполняемым), ***он не похож на функцию***.

Таких вольностей, связанных с переопределением встроенных идентификаторов, языки *C++* и *Java* нам не позволяют, но *Python* оставляет за собой контроль над нашими действиями в отношении объектов в памяти.

При составлении имен переменных важно учитывать тот факт, что имя должно нести информацию о назначении этой переменной. Кроме того, нельзя создать переменную и не связать её с объектом. Если все-таки нужна такая, явно не связанная с объектом в памяти переменная, то её связывают с объектом *None*.

```
>>> Var1 = None
>>> type(Var1)
<class 'NoneType'>
```

Метод *type()* используется для получения информации о типе объекта по ссылке не него. Получили сообщение, что объект, на который указывает ссылка Var1 относится к специальному типу *NoneType*.

Рассмотрим следующий дополненный пример.

```
>>> Var1 = None
>>> type(Var1)
<class 'NoneType'>
>>> type(type(Var1))
<class 'type'>
>>> type(type(type(Var1)))
<class 'type'>
>>>
```

Получили сообщение, что метод *type()* возвращает объект типа '*type*', который тоже относится к ***специальному встроенному типу*** (классу '*type*'). Прежде чем продолжим, еще раз вспомним, что в языке Python встроенные типы данных делятся на изменяемые и неизменяемые. К изменяемым относятся и числовые типы: целочисленные 'int', длинный 'long', логический ‘bool’, вещественный 'float', комплексно сопряженный 'complex'. Очень сложно определить разницу между типами 'int' и 'long', так как в памяти программы целочисленные данные хранятся в виде строковой записи. Это позволяет очень просто работать с ***длинной арифметикой***. Например, для того, чтобы на C/C++ или Java вычислить значение 100^{100} придется либо воспользоваться специальной библиотекой, либо самостоятельно её реализовывать. В Python это делается встроенными средствами, включенными в состав стандартной поставки. Рассмотрим следующий код.

```
>>> 100**100  # возвести 100 в 100-ую степень
```

```
10000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
>>> type((100**100))  # получить тип результата
<class 'int'>
>>> len(str(100**100)) # узнать длину результата вычисления в цифрах
201       # результат представляет собой одну единицу и двести нулей
>>>
```

Задачи

Для начинающих изучать язык программирования Python рекомендуется решать эти задачи в консольном варианте. Тем, кто уже отчасти знает его, не возбраняется выполнять решение задач в виде программ с графическим интерфейсом. Программирование графического интерфейса средствами библиотеки Tkinter будет представлено несколько позже. «Десерт» из первого урока не в счет.

Для решения следующих задач Вам могут потребоваться средства модуля *math*, в котором, как и в библиотеке языка *C++ <cmath>* содержатся основные математические функции и константы. Для подключения данного модуля в начале скрипта нужно указать следующую инструкцию: *import math*. Чтобы получить информацию об объектах, содержащихся в модуле, нужно использовать инструкцию *dir* (сокращение от *direction* – ‘***правила использования***’)

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']
>>>
```

То есть, если необходимо, например, вычислить длину окружности, необходимо импортировать модуль *math* и воспользоваться значением переменной *math.pi*.

```
>>> import math  # импортировать модуль math полностью
>>> R = 12.7
>>> Len = math.pi * R * R
>>> print("Длина окружности радиуса %f равна %f" % (R, Len))
Длина окружности радиуса 12.700000 равна 506.707479
```

Данный пример может быть несколько оптимизирован по памяти, так как инструкция *import* позволяет импортировать не весь модуль, а только отдельные его компоненты. Для этого используется следующая форма.

from math import <имя_компонента>

В нашем случае необходимо импортировать только значение константы Пи (π). И выше приведенная задача принимает следующий вид.

```
>>> from math import pi
>>> Len = pi * R * R
```

Модифицируем данный код, введя возможность генерации случайного значения радиуса в интервале (0.0, 1.0).

```
>>> import math, random  # импорт двух модулей
>>> R = random.random()  # сгенерировать случайное значение радиуса
>>> Len = pi * R * R     # вычислить длину окружности
>>> print("Длина окружности радиуса %f равна %f :" % (R, Len))
Длина окружности радиуса 0.348308 равна 0.381133 :
>>>
```

Настоящее руководство не претендует на роль полноценного учебника по языку программирования Python версии 3.х. За полной информацией обращайтесь к учебникам и справочным ресурсам.

Предлагаемые задачи могут быть решены в виде отдельных программ-скриптов, так и в виде инструкций, вводимых в окне среды *IDLE*.

Рассмотрим решение следующей задачи. Вычислить расстояние между двумя точками с координатами A(x1, y1) и B(x2, y2).

Вариант №1.

```
import math
x1 = 1
y1 = 1
x2 = 4
y2 = 5
AB = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
print("Расстояние между точками равно :", AB)
```

Вариант №2.

```
import math
x1 = int(input("Введите значение координаты x1 "))
y1 = int(input("Введите значение координаты y1 "))
x2 = int(input("Введите значение координаты x2 "))
y1 = int(input("Введите значение координаты y2 "))
AB = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
print("Расстояние между точками равно :", AB)
```

Вариант №3.

```
import math
coord = [1, 2, 3, 4]  # список значений координат точек
x1 = coord[0]  # обращение к элементы списка по индексу
y1 = coord[1]
x2 = coord[2]
y1 = coord[3]
AB = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
print("Расстояние между точками равно :", AB)
```

Как можете видеть, у вас достаточно возможностей для организации процесса ввода исходных данных для задачи.

Перейдем непосредственно к задачам. Предлагается решить относительно простые и, возможно, не очень интересные задачи, которые позволят полноценно освоить данный раздел программирования на Python. Часть задач была заимствована из задачника М.Э Абрамян «1000 задач по программированию» и других источников, и уже знакома нашим студентам из курса «Программирование».

1. Записать по правилам языка Python следующие выражения:

1) $2x$-$3y$	2) $\|n-m\|$	3) $3\sin\alpha\cos\beta+4\cos\alpha\sin\beta$
4) $\sin x$	5) $5\cos y$	6) $a\sqrt{12-b}$
7) a^2-$2b$	8) -81,05a^2	9) $7{,}63\sin(2\alpha+\varphi)\cos(3\beta-\varphi)$
10) $\sqrt{x}$	11) $y\sqrt{ax}$	12) $\sqrt[y]{x}$
13) $\|x-y\|$	14) $\sqrt{\|x-y\|}$	15) $\left\|a\sin x^2+b\cos x^2\right\|$
16) $-2b+\sqrt{b^2-4ac}$	17) $\sqrt[3]{1+x+x^2}$	18) $\sqrt[5]{3{,}5x-\sqrt[3]{x-y}}$
19) $\sqrt{x_1^2+x_2^2-x_3^2}$	20) $x_1x_2\sin(x_3+\varphi)$	21) $\sqrt{2(A+B)-A\sin^2 x-B\cos^2 x}$

2. Дана сторона квадрата a. Найдите его периметр $P = 4{\cdot}a$.
3. Дана сторона квадрата a. Найдите его площадь $S = a^2$.
4. Даны стороны прямоугольника a и b. Найти его площадь $S = a{\cdot}b$ и периметр $P = 2{\cdot}(a + b)$.
5. Дан диаметр окружности d. Найти ее длину $L = \pi{\cdot}d$. В качестве значения π использовать `math.pi`.
6. Составить программу для вычисления значения функции $y = 7x^2 - 3x + 6$ при любом значении x.

7. Составить программу для вычисления значения функции $x = 12a^3 - 4a^2 + 7a - 17$ при любом значении a.
8. Составить программу для вычисления значения функции $\frac{a^2 + 12ab + 3b^2}{\sqrt{a^2 + b^2 + 1}}$ при любом значении a и b.
9. Считая, что Земля – идеальная сфера с радиусом $R \approx 6350$ км, определить расстояние до линии горизонта от точки с заданной высотой над Землей (*не забудь нарисовать схему*).
10. Даны объем и масса тела. Определить плотность материала этого тела, считая его однородным.
11. Даны два целых числа, найти их среднее арифметическое и среднее геометрическое.
12. Известны количество жителей в государстве и площадь его территории. Определить плотность населения в этом государстве.
13. Найти площадь кольца по заданному внешнему и внутреннему радиусам.
14. Даны два числа. Найти их сумму, разность, произведение, а также частное от деления первого числа на второе.
15. Даны стороны прямоугольника. Найти его периметр и длину диагонали.
16. Даны длины сторон прямоугольного параллелепипеда. Найти его объем и площадь боковой поверхности.

Строковый тип данных

В языке Python существует несколько классов для представления строк, но мы начнем работать с классом `str()`. С некоторой натяжкой можно сказать, что данный класс является наследником класса `std::string` из языка *C++*.

Как и в языке *C++*, для создания строки, объекта класса `str`, вызывается конструктор `str()`. При этом конструктор может быть вызван явно и неявно. Например:

```
name = "Bob Smith"  # неявный вызов конструктора
book = str("Python: How to Program")  # явный вызов контсруктора
```

В отличие от языка *C++* в *Python* нет необходимости думать о том, как вызывается конструктор, явно или неявно! В *Python* нет таких возможностей для управления доступом конструктору, как `explicit` в *C++*. Хотя *Python* и поддерживает вызов *C/C++* - кода и, как следствие, можно ***«наступить на детские грабли»*** в унаследованном коде, но в *Python* этого пока нет… Не будем впадать в ностальгию по ***великому и могучему*** *C++*, а сосредоточимся на *Python*.

Строки в *Python* относятся к категории неизменяемых типов данных, это значит, что при выполнении операций над строкой создается новая строка, а старая остается без изменения и занимает память.

```
Str1 = 'Jolly'  # создан один объект
Str1 += ' '     # создано еще два объекта. Количество объектов - три
Str1 += 'Roger' # Создано еще два объекта. Количество объектов - пять
```

На таком простом примере можем проследить ***один из недостатков языка*** *Python*, унаследованный от языка *C++*, связанный с работой «***сборщика мусора***»: Встроенный «***сборщик мусора***» использует ***механизм подсчета активных ссылок***, но у вас нет информации, когда выполняется эта ***волшебная сборка мусора*** – удаление не используемых объектом. В *Java*, например, очень высока вероятность, что объекты будут удалены из памяти только один раз за время работы приложения и произойдет это уже после завершения работы java-приложения. Считается, что памяти на современных компьютерах уже достаточно много и заботиться о её экономии нет необходимости…

Вспомним варианты объявления строк в Python, приведенные на первом занятии.

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

Для нас строка - это некоторый текст, заключенный в парные кавычки, для кодирования которого используется ***unicode – кодировка***. Кроме того, строки относятся к классу ***последовательностей***, ранее знакомых в языке C++ как контейнеры последовательного доступа (*std::vector*, *std::array*, *std::string*). Но в *Python*, кроме того, есть ***хорошие методы для работы с текстом***, содержащихся в этих самых строках.

Рассмотрим следующий пример.

```
>>> myString = "Hello World"
>>> print(myString.center(40, '*'))
**************Hello World***************
>>> newString = " " + myString + " "
>>> newString = newString.center(50, '*')
>>> newString
'****************** Hello World ******************'
>>>
```

Использованный метод *center()* создает новую строку, длиной не менее чем значение первого аргумента символов, в середину которой он вставляет строковый объект, для которого вызывается данный метод, а «***свободные позиции***» будут заполнены символом, указанным в значении второго аргумента.

```
>>> a1 = 'aaaaaaaaaa'  # создан объект, содержащий 10 символов 'a'
>>> # создать новый объект, длиной не менее 5 символов
>>> a1.center(5, '*')  # свободные позиции заполнить символом '*'
```

```
'aaaaaaaaaa'
>>> len(a1)
10
>>>

>>> str1 = 'Jolly Roger'
>>> str2 = str1.center(40, '*')
>>> print(str2)
**************Jolly Roger***************
>>> print("Lenght str1 =", len(str1), "lenght str2 =", len(str2))
Lenght str1 = 11 lenght str2 = 40
>>>
```

Метод *center*(<*ширина*> [<*символ*=' '>]) позволяет управлять процессом заполнения строк по заданному шаблону и может быть использован для ***декорирования*** потокового вывода. Значением по умолчанию для параметра <символ> является символ пробела.

Метод *len*(), как можете догадаться, возвращает длину строки в символах.

Метод *ljust*(<*ширина* > [,<*символ*=' '>]) выполняет выравнивание формируемой строки по левому краю внутри поля указанной ширины. Значением по умолчанию для параметра <символ> является символ пробела.

```
>>> s1 = "String"
>>> s2 = s1.ljust(12); s3 = s1.ljust(12, '*')
>>> s2, s3
('String      ', 'String******')
>>>
```

Метод *rjust*(<*ширина* > [,<*символ*=' '>]) выполняет выравнивание формируемой строки по правому краю внутри поля указанной ширины. Значением по умолчанию для параметра <символ> является символ пробела.

```
>>> s2 = s1.rjust(12); s3 = s1.rjust(12, '*')
>>> s2, s3
('      String', '******String')
>>>
```

Метод *zfill*(<*ширина*>) выолняет выравнивание формируемой строки внутри поля указанной ширины. Слева от фрагмента будут добавлены символы нуля. Если длина форматируемой строки превышает ширину поля, то значение ширины игнорируется.

```
>>> s2 = s1.zfill(12); s3 = s1.zfill(6); s2, s3
('000000String', 'String')
>>>
```

Форматирование строк с помощью % (устаревший подход)

Язык *Python*, как и *Java*, унаследовал механизм форматирования в стиле языка *C++*. Предполагается, что в последующих верст Python оператор форматирования % может быть удален. Вместо него в новом коде рекомендуется использовать метод *format*(), который будет рассмотрен ниже.

Форматирование имеет следующий синтаксис.

```
<Строка_специального_сормата> % <Значения>
```

Внутри параметра *<Строка_специального_сормата>* , как и в *ANSI C* могут быть символы, отображаемые без изменения, управляющие символы и спецификаторы формата, имеющие следующий синтаксис (вспоминаем горячо любимую функцию *printf*() и не особо жалуемую *sprintf*()):

```
% [(<ключ>)][<флаг>][<ширина>][.<точность>]<спецификатор_формата>
```

Почти родной синтаксис вывода строки в Python.

```
>>> name = "Bob Smith"
>>> print("Hello, %s" % name)
Hello, Bob Smith
```

Метод *print*(), в отличие от *C*-шной функции *printf*(), выводит в стандартный поток вывода уже подготовленный строковый объект

```
>>> type("%s" % name)
<class 'str'>
>>> "%s" % name
'Bob Smith'
>>>
```

Отрадно, что данная возможность по манипуляции над строками выделена в самостоятельный метод. Конечно, в *C++* есть строковые потоки (*std::basic_stringstream<char>* и *std::stringstream*), имеется и замечательное средство *boost::format*, но работа над строками в *Python* для программиста сделана значительно проще.

Рассмотрим следующие примеры форматирования строк

```
>>> "%s" % 10  # формат строки
'10'  # результат - строка
>>> "%d" % 10  # формат 10-чного целого
'10'  # результат - строка

>>> "%x" % 10  # формат 16-ричного целого
'a'  # результат - строка
>>> "%o" % 10  # формат 8-ричного целого
'12'  # результат - строка
>>> "%s %s" % ("Hello", "World")
'Hello World'
```

```
>>>
```

Если необходимо встроить в строку несколько объектов, в ней используется необходимое количество спецификаторов преобразования, сами объекты упаковываются в кортеж (*tuple*) – помещаются в круглые скобки и разделяются запятыми.

```
>>> "масса тела %s кг" % 16  # один элемент
'масса тела 16 кг'
>>> "Дата: %02d/%02d/%4d" % (18, 7, 2017)  # несколько элементов
'Дата: 18/07/2017'
>>>
```

Следует учесть, что при выполнении такого рода преобразований выполняется проверка на возможность осуществляемого преобразования, и если оно невозможно, то генерируется соответствующее исключение.

```
>>> "%d" % "one"
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    "%d" % "one"
TypeError: %d format: a number is required, not str
>>>
```

Ещё раз вспомним правила работы со спецификатором форматного преобразования.

```
% [(<ключ>)][<флаг>][<ширина>][.<точность>]<спецификатор_формата>
```

Поле ***<ключ>*** используется в случае, если необходимо вывести значение элемента контейнера ассоциативного доступа – словаря, то выполняется следующее преобразование

```
>>> "%(name)s - %(year)s" % {"name": "Миг-35", "year": 2017}
'Миг-35 - 2017'
>>>
```

Поле ***<флаг>*** позволяет управлять форматом отображения целочисленный констант, задает выравнивание, заполняет неиспользуемые символьные позиции пробелами и задает обязательный вывод знака для чисел. Поле флаг может принимать следующие значения:
- вывод префикса системы счисления для 8-ричных, 16-ричных целочисленных значений.

```
>>> "%#o %#d %#x" % (10, 10, 10)
'0o12 10 0xa'
>>> "%#x %#X" % (10, 10)
'0xa 0XA'
>>> "%d %x %#x" % (0xFF, 0xff, 15)
```

```
'255 ff 0xf'
>>>
```

0 (ноль) – задает наличие ведущих нулей для числового значения, используется совместно со спецификатором <*ширина*>.

```
"'  12' '0012'"
>>>
```

- (знак минус) – определяет выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе. Если данный спецификатор используется одновременно со спецификатором `0`, то действие спецификатора `0` будет отменено.

```
>>> "'%5d' '%-5d'" % (12, 12)
"'   12' '12   '"
>>> "'%05d' '%-05d'" % (12, 12)
"'00012' '12   '"
>>>
```

пробел – вставляет пробелы перед положительным числом. Перед отрицательным числом будет стоять знак минус. Используется для выравнивания ширины поля, занимаем числовыми значениями, когда нет необходимости в выводе знака числа.

```
>>> "'% d' '% d'" % (9, -9)
"' 9' '-9'"
>>>
```

+ (знак плюс) – задаёт обязательный вывод знака, как для положительных, так и для отрицательных чисел. Если спецификатор **+** используется одновременно со спецификатором **пробел**, то действие спецификатора пробел будет отменено.

```
>>> "'%+d' '% +d' '%+d'" % (9, -9, -9)
"'+9' '-9' '-9'"
>>>
```

Поле ***<ширина>*** определяет минимальную ширину поля, используемую для символьного представления объекта. Если длина объекта в символах превышает указанное значение, то оно игнорируется.

```
>>> "'%15s' '%s'" % ("Python rules", "Python rules")
"'   Python rules' 'Python rules'"
>>>
```

Имеется возможность для передачи значения поля <*ширина*> в списке параметров, в этом случае в форматируемой строке оно заменяется символом '*'.

```
>>> "'%*s' '%15s'" % (15, "Python rules", "Python rules")
"'   Python rules' '   Python rules'"
```

```
>>>
```

Поле ***<.точность>*** задает количество знаков после точки для вещественных чисел. Перед данным параметром обязательно должна стоять точка. Символы – спецификаторы типа для формата отображения для числовых значений отчасти совпадают с символам – спецификаторами в языке *ANSI C*, да и в *Java* используются такой же подход.

```
>>> import math                    # импортировать данные модуля math
>>> math.cos(0.12)                 # вывод числового значения функции
0.9928086358538663
>>> type(math.cos(0.12))           # вывод информации о типе объекта
<class 'float'>
>>>
>>> type("%f" % math.cos(0.12))    # вывод информации о типе объекта
<class 'str'>
>>>
>>> "%s %f %.0f %.4f" % (math.cos(0.12), math.cos(0.12),\
                 math.cos(0.12), math.cos(0.12))
'0.9928086358538663 0.992809 1 0.9928'
>>>
```

Обратите внимание на третье значение в последней строке вывода: вещественное значение при преобразовании было округлено в большую сторону. С точки зрения человека, это может быть и правильно, но с математической точки зрения при формировании результата, путем преобразования *float*()→*int*(), результат получен неверный! Объяснить это можно отчасти тем, что в результате наших действий получается не числовой объект, а строка. Данный факт необходимо учитывать при разработке расчетных программных систем, где используются механизмы долговременного хранения числовых данных в базах данных (БД), так как в большинстве БД числовые данные хранятся в виде строкового строки.

Поле ***<спецификатор_формата>*** задает тип преобразования при формировании строкового представления объекта. Данное поле является обязательным. Могут быть использованы следующие значения:

s – объект преобразуется в строку с помощью метода-конструктора *str*().

r – преобразует объект в строку с помощью метода *repr*().

```
>>> "'%s' '%r'" % ("Hello", "Dummy")
"'Hello' ''Dummy'"
>>>
>>> print("Hello"); "%r" % "Hello"; print("%r" % "Hello")
Hello
"'Hello'"
'Hello'
>>>
```

Обратите внимание на следующий пример, каким образом *Python* реагирует на синтаксическую ошибку – использование знака «запятая» вместо «точки с запятой».

```
>>> "%r" % "Hello", print("%r" % "Mr.Smith")
'Mr.Smith'
("'Hello'", None)
>>>
```

Инструкции, разделенные знаком «запятая», Python рассматривает как кортеж (*tuple*) – контейнер последовательного доступа. Поэтому сначала осуществляется вывод в стандартный поток вывода результатов метода *print*(), а затем во второй строке отображается сам кортеж, где вместо метода фигурирует «заглушка», соответствующая уже вызванному методу. В таких вот «косяках» и проявляется С-шная сущность языка *Python* и модификатор стека *cdecl*, используемый у С-шных функций по умолчанию (учи матчасть ☺)…

a – преобразует объект в строку с помощью метода *ascii*().
c – преобразует объект в одиночный символ или преобразует числовое значение в символ.

```
>>> "%a" % "this is a string"
"'this is a string'"
>>> "%a" % ("string")
"'string'"
>>> "%a" % "строка"
"'\\u0441\\u0442\\u0440\\u043e\\u043a\\u0430'"
>>>
```

Последняя строка вывода демонстрирует факт использования unicode-кодировки для отображения кириллицы в строковых объектах.

```
>>> "%c" % 1234
'Ä'
>>> "%c" % 9876
'□'
>>> "%c" % 12345
'卅'
>>> "%c" % 23451
'宛'
>>>
```

d и **i** – формируют строковое отображение для числового объекта в 10-чном формате.

```
>>> s1 = "%d %d %d" % (12, -13, 14.56)
```

```
>>> print(s1)
12 -13 14
>>> s1 = "%i %i %i" % (12, -13, 14.56)
>>> print(s1)
12 -13 14
>>>
```

o – формирует строковое отображение объекта в 8-ричном формате.

```
>>> s1 = "%o %o %o" % (7, 8, 9); print(s1)
7 10 11
>>>
>>> s1 = "%o %o %o" % (12, -13, 14.56); print(s1)
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    s1 = "%o %o %o" % (12, -13, 14.56); print(s1)
TypeError: %o format: an integer is required, not float
```

Обратите внимание, что при попытке форматирования значения типа *float* в *str* было сгенерировано исключение.

```
>>> s1 = "%o %o %o" % (0o7, 7, 0o77); print(s1)
7 7 77
>>>
```

x – формирует строковое отображение объекта в 16-ричном формате в нижнем регистре.
X – формируется строковое отображение объекта в 16-ричном формате в верхнем регистре.

```
>>> "%x %x %X %X" % (10, 0xA, 15, 0xf)
'a a F F'
>>>
```

```
>>> "%x" % 12.34
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    "%x" % 12.34
TypeError: %x format: an integer is required, not float
>>>
```

Обратите внимание на выброс объекта исключения при попытке форматирования вещественного значения в строковое представление: в старых версия языка *Python* данный пример был допустим.

f и **F** – формируется строковое отображение для вещественных чисел в 10-чном формате.

e – формируется строковое отображение для вещественных чисел в экспоненциальном формате.
g – равноценно спецификаторам f или e (выбирается более короткая, округленная запись числа).
G – равноценно спецификаторам F или G (выбирается более короткая, округленная запись числа).

```
>>> import math; "%f %e %g" % (math.pi, math.pi, math.pi)
'3.141593 3.141593e+00 3.14159'
>>> "%F %E %G" % (math.pi, math.pi, math.pi)
'3.141593 3.141593E+00 3.14159'
>>>
```

Если внутри строки нужно поместить символ '%', то этот символ необходимо удвоить.

```
>>> "%%"
'%%'
>>> "%% %s" % ("знак процента")
'% знак процента'
>>>
```

Рассмотрим пример использования форматирования строк, заимствованный из книги «Прохоренок Н.А. Python 3 и PyQt. Разработка приложений. – 2012г.»

```
# -*- coding: utf-8 -*-
html = """<html>
<head><title>%(title)s</title>
</head>
<body>
<h1>%(h1)s</h1>
<div>%(content)s</div>
</body>
</html>"""
arr = {"title": "Это название документа",
       "h1": "Это заголовок первого уровня",
       "content": "Это основное содержание страницы"}
print(html % arr)  # Подставляем значения и выводим шаблон
input()
```

Результат выполнения сценария.

```
<html>
<head><title>Это название документа</title>
</head>
<body>
<h1>Это заголовок первого уровня</h1>
<div>Это основное содержание страницы</div>
</body>
```

```
</html>
```

Форматирование строк с помощью метода *format()*

Вспомним фразу из фильма Гайдая «Приключения Шурика» - «Надо, Вася. Надо.», произнесенную Шуриком перед экзекуцией. И напишем инструкцию, формирующую данную фразу.

```
>>> "Надо, {}. Надо.".format("Вася")
'Надо, Вася. Надо.'
>>>
```

```
>>> "{2} + {1} = {0}".format("Дружба", "Саша", "Маша")
'Маша + Саша = Дружба'
>>>
```

Метод *format()* позволяет управлять процессом формирования строк на основании некоего шаблона, который также является строкой. Позиция, в которую осуществляется вставка необходимого блока, размещается внутри фигурных скобок. Данный метод можно считать развитием С-подобного метода форматирования строк, но он имеет значительные преимущества перед %-форматированием. Синтаксис метода *format()*:

```
<строка> = <управляющая_строка>.format(*args, **kwargs)
```

Где *<управляющая_строка>* представляет собой строку, содержащую специальные управляющие поля, заключеные в фигурные скобки. Управляющие поля имеют следующий формат:

```
{[<Поле>][!<Функция>][:<Формат>]}
```

Параметр *<Поле>* может содержать значение индекса позиции (нумерация начинается с нуля, как для поля *enum* в *ANSI C*) или ключ. Рассмотрим следующие примеры использования метода *format()*.

```
>>> "{1}, {2}, {0}".format("1","2","3")
'2, 3, 1'
>>>
>>> "{color} {model} =).".format(color="Черный", model="Бумер")
'Черный Бумер =).'
>>> "{event}-{0}".format(2017, event="IВолга")
'IВолга-2017'
>>> "{event}-{date}".format(date=2017, event="IВолга")
'IВолга-2017'
```

Приведенные примеры соответствуют аргументу метода формат **args*. В нашем случае это просто набор ссылок на простые объекты. Если объекты

представляют собой контейнеры, например, кортежи, то используется вариант аргумента ***kwargs*. Например:

```
>>> str1 = ["мама", "мыла", "раму"]
['мама', 'мыла', 'раму']
>>> "{0[0]} {0[1]} {0[2]}".format(str1)
'мама мыла раму'
>>>
>>> str1 = ["Мама", "мыла", "Милу", "мылом"]
>>> str2 = ("Мила", "мыло", "не", "любила")
>>> "{0[0]} {0[1]} {0[2]} {0[3]}\n\
{1[0]} {1[1]} {1[2]} {1[3]}".format(str1, str2)
'Мама мыла Милу мылом\nМила мыло не любила'
>>> str3 = "{0[0]} {0[1]} {0[2]} {0[3]}\n\
\n{1[0]} {1[1]} {1[2]} {1[3]}".format(str1, str2)
>>> print(str3)
Мама мыла Милу мылом
Мила мыло не любила
>>>
```

В приведенных примерах поле аргументов метода *format*() содержит кортеж аргументов-контейнеров, доступ к которым осуществляется по индексу. Внутри объекта-контейнера доступ к его элементам также осуществляется по индексу. Ограничения на количество вложений, то есть на структуру составного объекта нет.

Измененный пример, отчасти заимствованный из книги Марка Лутца – в своей книге он так «не накручивал».

```
>>> bob = ["Bob Smith", 42, 30000, "software"]
>>> sue = ["Sue Jones", 45, 40000, "hardware"]
>>> people = [1, bob, 2, sue]
>>> "{0[0]} : {0[1][0]}, {0[1][3]}\n\
{0[2]} : {0[3][0]}, {0[3][3]}".format(people)
'1 : Bob Smith, software\n2 : Sue Jones, hardware'
>>> print("{0[0]} : {0[1][0]}, {0[1][3]}\n\
{0[2]} : {0[3][0]}, {0[3][3]}".format(people))
1 : Bob Smith, software
2 : Sue Jones, hardware
>>>
```

Для доступа к атрибута объекта используется точечная нотация:

```
>>> class Event: name, date = "IВолга", "2017"

>>> event = Event() # вызов конструктора класса Event
>>> "{0.name}-{0.date}".format(event)
'IВолга-2017'
>>>
```

В случае, если необходимо передать в качестве параметров контейнер ассоциативного доступа, то самому параметру метода формат предшествует удвоенный символ * (звездочка), а постановка полей контейнера осуществляется по имени ключа. Например:

```
>>> it_forum={"name":"IВолга", "year": 2017}
>>> "{name}-{year}".format(**it_forum)
'IВолга-2017'
>>>
>>> ngtu={"Adress":"Minin St., 24, Nizhny Novgorod, 603950, Russia",\
     "name" : "Nizhny Novgorod State Technical University n.a. R.E.
Alekseev",\
     "phone" : "(831) 436-23-25", "fax" : "(831) 436-94-75"}
>>> print("{Adress}\nPhone:{phone};
Fax:{fax}\n{name}".format(**ngtu))
Minin St., 24, Nizhny Novgorod, 603950, Russia
Phone:(831) 436-23-25; Fax:(831) 436-94-75
Nizhny Novgorod State Technical University n.a. R.E. Alekseev
>>>
```

Существует краткая форма записи <*управляющей_строки*>, в которой параметр <*Поле*> не указывается. В этом случае используется порядок подстановки параметров, соответствующий порядку следования аргументов метода *format*().

```
>>> "{} {} {} {} {}".format(1,2,3,4,5)
'1 2 3 4 5'
>>> "{n} {} {} {} {}".format(1,2,3,4,n=0)
'0 1 2 3 4'
>>>
>>> first = "John"; last = "Smith"
>>> "Good morning, {0} {1}".format(first, last)
'Good morning, John Smith'
>>> "Good morning, {} {}".format(first, last)
'Good morning, John Smith'
>>>
>>> names = (first, last)
>>> "Good morning, {} {}".format(*names)
'Good morning, John Smith'
>>>
>>> "Good morning, {first} {last}".format(first="John", last="Smith")
'Good morning, John Smith'
>>>
>>> "Good {0}, {first} {last}".format('morning', **person)
'Good morning, John Smith'
>>> "Good {}, {first} {last}".format('morning', **person)
'Good morning, John Smith'
>>>
```

Параметр <*Функция*> метода *format*() задает функцию, используемую для создания строкового объекта при подстановке объекта в формируемую строку. Доступны следующие функции-конструкторы: *str*(), *repr*() и *ascii*(). Имя функции сокращается до одного символа и ему предшествует восклицательный знак. Если выполняется «встраивание» нескольких объектов, перечисляемых в списке параметров, то функции предшествует «номер» объекта .

```
>>> "{0!s} {1!s} {2!s}".format("this", "is", "a string")
'this is a string'
>>> "{0!s} {0!r}".format("Hello")
"Hello 'Hello'"
>>> "{0!a}".format(23541)
'23541'
>>> "{!a}".format("Ы")
"'\\u042b'"
>>> "{!a}".format("МИРУ МИР")
"'\\u041c\\u0418\\u0420\\u0423 \\u041c\\u0418\\u0420'"
>>>
```

Использование параметра <*Формат*> отчасти напоминает спецификатор форматирования, используемый с %-форматом. Он имеет следующий синтаксис:

```
[[<Символ_заполнения>]<Выравнивание>][<Знак>][#][0][<Ширина>][,]
[.<Точность>][<Формат_отображения>]
```

Параметр <Ширина>, как можно догадаться, задает минимальную ширину поля, используемую для строкового представления объекта. Если длина строки превышает указанное значение, то данный параметр игнорируется и строка выводится полностью.

```
>>> "{0:10} - {1:15}".format("Знание", "сила")
'Знание     - сила           '
>>>
>>> "Питон - это {value:*^10}!".format(value="прикольно")
'Питон - это прикольно*!'
>>> "Питон - это {value:*^13}!".format(value="прикольно")
'Питон - это **прикольно**!'
>>>
```

Ширину поля можно передать в качестве параметра метода format(), для этого вместо числового значения внутри фигурных скобок указывается в качестве индекса.

```
>>> "Питон - это {0:*^{1}}!".format("прикольно", 15)
'Питон - это ***прикольно***!'
>>>
>>> "{0:{1}}".format("Hello", 9)
'Hello    '
>>> "{0:>{1}}".format("Hello", 9)
```

```
'    Hello'
>>> "{0:^{1}}".format("Hello", 9)
'  Hello  '
>>>
```

Как можете заметить, по умолчанию используется выравнивание по правому краю поля формируемой строки. Управлять выравниванием позволяет параметр <*Выравнивание*>. Доступны следующие значения данного параметра:
< – выравнивание по левому краю поля;
> – выравнивание по правому краю поля;
^ – выравнивание по центру поля;
= – используется для вывода числовых значений: знак числа выравнивается по левому краю, а число по правому краю.

```
>>> "{0:<10}|{1:>10}|{2:^10}".format("Xa","Xa","Xa")
'Xa        |        Xa|    Xa    '
>>>
>>> "'{0:=10}' '{1:=10}'".format(-15, 15)
"'-       15' '        15'"
>>> "'{0:=010}' '{1:=010}'".format(-15, 15)
"'-000000015' '0000000015'"
>>>
```

Из вышеприведенного примера видно, что по умолчанию пустые позиции заполняются пробельными символами. Для того чтобы вместо пробелов вывести нули, перед шириной поля необходимо указать нуль. Такой же результат можно получить и при использовании параметр <*Символ_заполнения*> со значением, равным нулю.

```
>>> "'{0:0=10}' '{1:0=10}'".format(-15, 15)
"'-000000015' '0000000015'"
>>>
```

В качестве параметра <*Символ_заполнения*> могут быть использованы любые отображаемые символы.

```
>>> "'{0:$=10}' '{1:@=10}'".format(-15, 15)
"'-$$$$$$$15' '@@@@@@@@15'"
>>>
>>> "Стоимость изделия - ${price:0=10}".format(price=12345)
'Стоимость изделия - $0000012345'
>>> "Стоимость изделия - ${price:010}".format(price=12345)
'Стоимость изделия - $0000012345'
>>>
```

Кроме этого, при помощи параметра <*Знак*> метод *format*() позволяет управлять выводом знака числа. Для этого могут быть использованы следующие значения:

+ – отображение знака числа как для отрицательных, так и для положительных чисел;
- – (знак минус) отображение знака только для отрицательных чисел;
пробел – вставлять пробельный символ только перед положительным числом, перед отрицательным числом отображается знак минус.

```
>>> "'{0:+}' '{1:+}' '0:-' '{1:-}'".format(-15, 15)
"'-15' '+15' '0:-' '15'"
>>> "'{0:+3}' '{1:+3}' '0:-3' '{1:-3}'".format(-15, 15)
"'-15' '+15' '0:-3' ' 15'"
>>> "'{0:+5}' '{1:+5}' '0:-5' '{1:-5}'".format(-15, 15)
"'  -15' '  +15' '0:-5' '   15'"
>>> "{0:*>-10}".format(-12)
'*******-12'
>>> "'{0:0>+5}' '{1:0>+5}' '0:0>-5' '{1:0>-5}'".format(-15, 15)
"'00-15' '00+15' '0:0>-5' '00015'"
>>> "'{0:*>+5}' '{1:*>+5}' '0:$>-5' '{1:$>-5}'".format(-15, 15)
"'**-15' '**+15' '0:$>-5' '$$$15'"
>>>
```

Для отображения целых числе могут быть использованы следующие значения параметра <*Формат_отображения*>
b – вывод целых чиселв двоичном формате;
o – (*oct*) вывод целых чисел в 8-ричном формате;
x – (*hex*) вывод целых чисел в 16-ричном формате в нижнем регистре;
X – (*hex*) вывод целых чисел в 16-ричном формате в верхнем регистре;

```
>>> "{0:d}(10-чное) = {0:b}(2-чное)".format(15)
'15(10-чное) = 1111(2-чное)'
>>> "int: {0:d}, hex: {0:x}, oct: {0:o}, bin: {0:b}".format(33)
'int: 33, hex: 21, oct: 41, bin: 100001'
>>> "int: {0:d}, hex: {0:#x}, oct: {0:#o}, bin: {0:#b}".format(33)
'int: 33, hex: 0x21, oct: 0o41, bin: 0b100001'
>>>
```

c – вывод символа, соответствующего ASCI – коду подставляемого параметра;

```
>>> "E-mail: JohnSmith{:c}yandex.ru".format(64)
'E-mail: JohnSmith@yandex.ru'
```

d – отображение целого значения в 10-чном формате;

```
>>> "'{0:}' '{0:d}'".format(123456789)
"'123456789' '123456789'"
>>> "{0:,d}".format(100000000000)
'100,000,000,000'
>>> "{:,}".format(9876543210)
'9,876,543,210'
>>>
```

n – аналогично значению `d` , но вывод осуществляется с учетом настройки локали

Имеется возможность для вывода целых чисел триадами, то есть тысячными разрядами. Для этого символу *d* предшествует запятая.

Формат отображения вещественных чисел также совпадает со спецификаторами %-форматирования (был унаследован из *ANSI C*). Используются спецификаторы *f*, *F*, *e*, *E*, *g*, *G*.

Отдельный интерес представляет собой формат перевода вещественного значения в проценты, для этого используется спецификатор %.

```
>>> "'{0:%}' '{1:.4%}'".format(0.03, 1.23)
"'3.000000%' '123.0000%'"
>>>
>>> points, total = 13, 25
>>> "Correct answer: {:.2%}.".format(points/total)
'Correct answer: 52.00%.'
>>>
```

Также имеются спецификаторы для работы с формата даты и времени.

```
>>> import datetime
>>> d = datetime.datetime(2017, 7, 27, 13, 28, 00)
>>> "{:%Y-%m-%d %H:%M:%S}".format(d)
'2017-07-27 13:28:00'
>>> "{:%Y/%m/%d %H:%M:%S}".format(d)
'2017/07/27 13:28:00'
>>> "{:%d/%m/%Y %H:%M:%S}".format(d)
'27/07/2017 13:28:00'
>>>
```

Замечание. *Средства форматирования метода `format()` несколько уступают в производительности средствам %-форматирования.*

Задачи на управление выводом на экран (в стандартный поток вывода)

Используя средства %-форматирования и метода format(), составьте программы для решения следующих задач.

1. Составить программу для вывода на экран даты рождения в формате ДД:ММ:ГГГГ, пустые позиции в записи даты заполнять нулями.

2. Составить программу, выводящую на экран приветствие пользователя, имя которого задается значением программной переменной.

3. Составить программу, запрашивающую у пользователя его имя, и выводящую на экран приветствие в формате «Здравствуй, » <Имя_пользователя>

«, разлюбезный друг!». Замечание - форма вежливого приветствия может быть изменена по желанию программиста.

4. Составить программу, выводящую на экран краткую справку о себе в следующем формате:

```
**********************************************************
* <Фамилия Имя>, <количество полных лет> «лет от роду»,   *
* <отношение к браку> [женат|не женат|замужем|не замужем] *
* <Краткая характеристика о себе, как о программисте>     *
* <Контактная информация>                                 *
**********************************************************
```

5. Составить программу, выводящую на экран краткую справку о любом вашем знакомом в формате «краткой справки, дававшейся героям фильма «17 мгновений весны»

6. Составить программу, запрашивающую у пользователя данные, согласно требованиям краткой справки, представленной в предыдущей задаче, и выводящую на экран сформированную справку. Обратите внимание на правила форматирования, текст должен быть вписан в рамку из символов псевдографики.

7. Составить программу, выводящую на экран значение константы «Пи» с точностью до 5 знаков после «плавающей точки».