

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ
И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Северо-Кавказский филиал
ордена Трудового Красного Знамени федерального государственного
бюджетного образовательного учреждения высшего образования
«Московский технический университет связи и информатики»

Методические указания для проведения
лабораторных работ и практических занятий
по дисциплине
**Языки программирования (Основы алгоритмизации и
программирования на основе языка PYTHON)**

(направление подготовки 10.03.01 Информационная безопасность)

Ростов-на-Дону
2022

Методические указания для проведения
лабораторных работ и практических занятий
по дисциплине
**Языки программирования (Основы алгоритмизации и программирования
на основе языка PYTHON)**

(направление подготовки 10.03.01 Информационная безопасность)

Составитель: И.А. Сосновский, доцент кафедры ИТСС

Рассмотрено и одобрено на заседании кафедры
Протокол от «29» августа 2022 г. № 1

Лабораторная работа 1. Введение в язык программирования Python.

Цель работы: Получить общие сведения о языке и среде разработки.

Python— это объектно-ориентированный, интерпретируемый, переносимый язык сверхвысокого уровня. Программирование на Python позволяет получать быстро и качественно необходимые программные модули.

В комплекте вместе с интерпретатором Python идет IDLE (интегрированная среда разработки). По своей сути она подобна интерпретатору, запущенному в интерактивном режиме с расширенным набором возможностей (подсветка синтаксиса, просмотр объектов, отладка и т.п.).

Для запуска IDLE в Windows необходимо перейти в папку Python в меню “Пуск” и найти там ярлык с именем “IDLE (Python 3.X XX-bit)”.

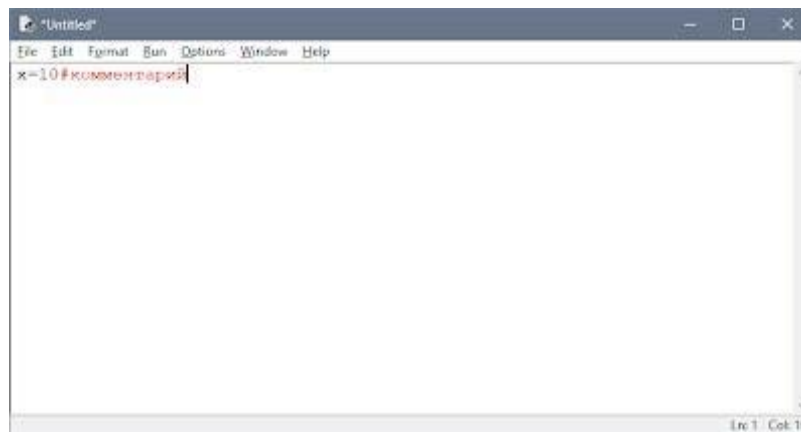
Для запуска редактора программы (кода) следует выполнить команду File->NewFile или сочетание клавиш Ctrl+N.

Любая Python-программа состоит из последовательности допустимых символов, записанных в определенном порядке и по определенным правилам.

Программа включает в себя:

- комментарии;
- команды;
- знаки пунктуации;
- идентификаторы;
- ключевые слова.

Комментарии в Python обозначаются предваряющим их символом # и продолжаются до конца строки (т.е. в Python все комментарии являются однострочными), при этом не допускается использование перед символом # кавычек:



Знаки пунктуации.

В алфавит Python входит достаточное количество знаков пунктуации, которые используются для различных целей. Например, знаки "+" или "*" могут использоваться для сложения и умножения, а знак запятой "," - для разделения параметров функций.

Идентификаторы.

Идентификаторы в Python это имена используемые для обозначения переменной, функции, класса, модуля или другого объекта.

Ключевые слова.

Некоторые слова имеют в Python специальное назначение и представляют собой управляющие конструкции языка.

Ключевые слова в Python:

['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']

Типы данных.

1. None (неопределенное значение переменной)
2. Логические переменные (Boolean Type)
3. Числа (Numeric Type)
4. int – целое число

5. float – число с плавающей точкой
6. complex – комплексное число
7. Списки (Sequence Type)
8. list – список
9. tuple – кортеж
10. range – диапазон
11. Строки (Text Sequence Type)
12. str

Ввод и вывод данных.

Ввод данных осуществляется при помощи команды **input**(список ввода):

```
a = input()
print(a)
```

В скобках функции можно указать сообщение - комментарий к вводимым данным:

```
a = input ("Введите количество: ")
```

Команда `input()` по умолчанию воспринимает входные данные как строку символов. Поэтому, чтобы ввести целочисленное значение, следует указать тип данных `int()`:

```
a = int (input())
```

Для ввода вещественных чисел применяется команда `a=float(input())`.

Вывод данных осуществляется при помощи команды **print**(список вывода): `a = 1.`

```
b = 2
print(a)
print(a + b)
print('сумма = ', a + b)
```

Существует возможность записи команд в одну строку, разделяя их через `;`. Однако не следует часто использовать такой способ, это снижает

удобочитаемость:

```
a = 1; b = 2; print(a)print (a + b)
print ('сумма = ', a + b)
```

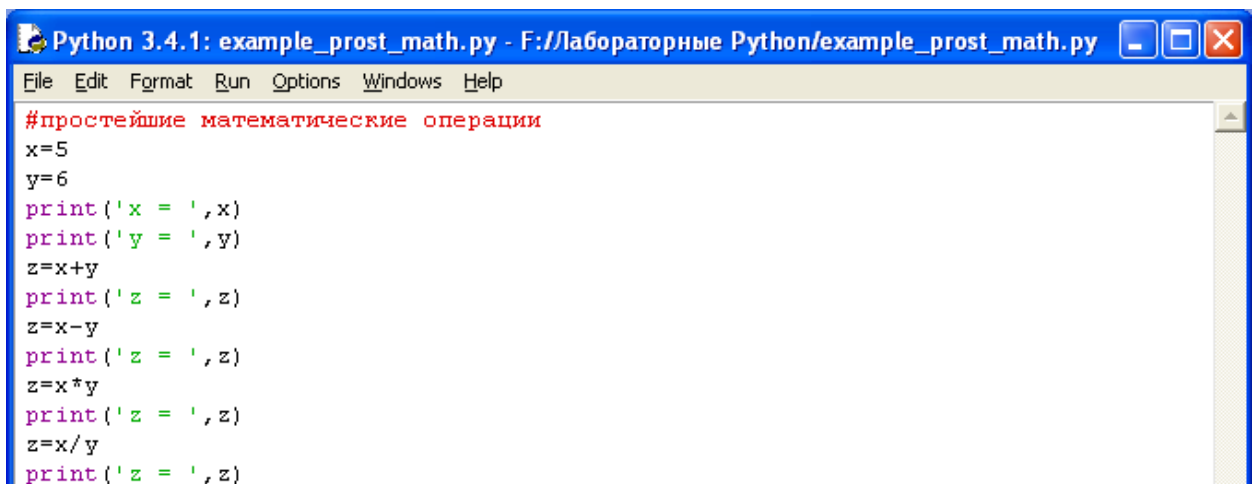
Для команды **print** может задаваться так называемый сепаратор — разделитель между элементами вывода:

```
x=2y=5
print ( x, "+", y, "=", x+y, sep = " " )
```

Результат отобразится с пробелами между элементами: $2 + 5 = 7$

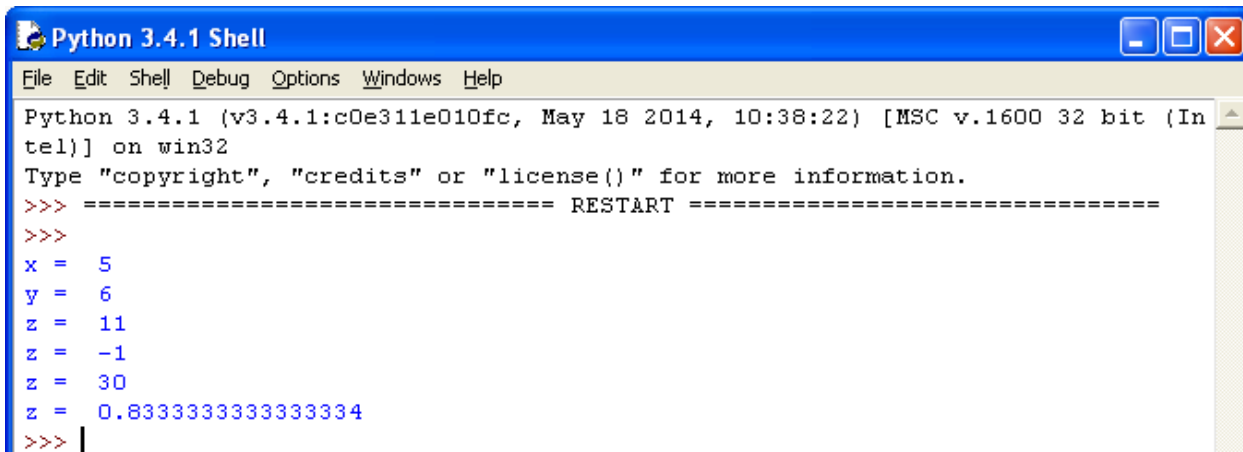
Простые арифметические операции над числами.

$x + y$	Сложение
$x - y$	Вычитание
$x * y$	Умножение
x / y	Деление



```
Python 3.4.1: example_prost_math.py - F://Лабораторные Python/example_prost_math.py
File Edit Format Run Options Windows Help
#простейшие математические операции
x=5
y=6
print ('x = ', x)
print ('y = ', y)
z=x+y
print ('z = ', z)
z=x-y
print ('z = ', z)
z=x*y
print ('z = ', z)
z=x/y
print ('z = ', z)
```

Пример программы на Python.



```
Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ----- RESTART -----
>>>
>>>
x = 5
y = 6
z = 11
z = -1
z = 30
z = 0.8333333333333334
>>> |
```

Результат выполнения программы с применением простых арифметических операций.

Для форматированного вывода используется **format**:

Строковый метод `format()` возвращает отформатированную версию строки, заменяя идентификаторы в фигурных скобках `{}`. Идентификаторы могут быть позиционными, числовыми индексами, ключами словарей, именами переменных.

Синтаксис команды **format**:

поле замены := "{" [имя поля] ["!" преобразование] [":" спецификация] "}"

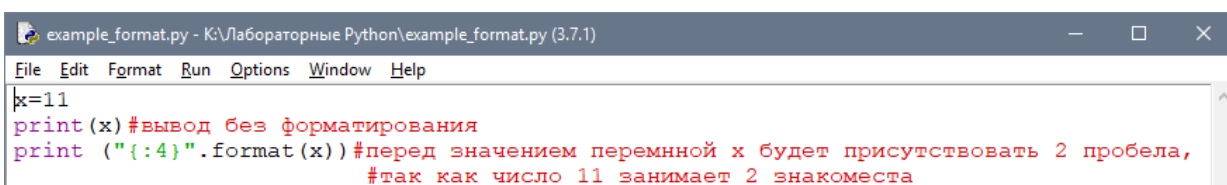
имя поля := `arg_name` (". имя атрибута | "[" индекс "]")*

преобразование := "r" (внутреннее представление) | "s" (человеческое представление)

спецификация := см. ниже

Аргументов в `format()` может быть больше, чем идентификаторов в строке. В таком случае оставшиеся игнорируются.

Идентификаторы могут быть либо индексами аргументов, либо ключами:



```
example_format.py - K:\Лабораторные Python\example_format.py (3.7.1)
File Edit Format Run Options Window Help
x=11
print(x) #вывод без форматирования
print("{:4}".format(x)) #перед значением переменной x будет присутствовать 2 пробела,
                        #так как число 11 занимает 2 знакоместа
```

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: K:\Лабораторные Python\example_format.py =====
11
    11
>>> |
```

В результате выведется число 11, а перед ним два пробела, так как указано использовать для вывода четыре знакоместа.

Или с несколькими аргументами:

```
example_format1.py - K:\Лабораторные Python\example_format1.py (3.7.1)
File Edit Format Run Options Window Help
x=2
print ("{:4d}{:4d}{:4d}".format (x, x, x))
```

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: K:\Лабораторные Python\example_format1.py =====
    2    2    2
>>>
```

спецификация	:= [[fill]align][sign][#][0][width][,][.precision][type]
заполнитель	:= символ кроме '{' или '}'
выравнивание	:= "<" ">" "=" "^"
знак	:= "+" "-" " "
ширина	:= integer
точность	:= integer
тип	:= "b" "c" "d" "e" "E" "f" "F" "g" "G" "n" "o" "s" "x" "X" "%" "

В итоге каждое из значений выводится из расчета 4 знакоместа.

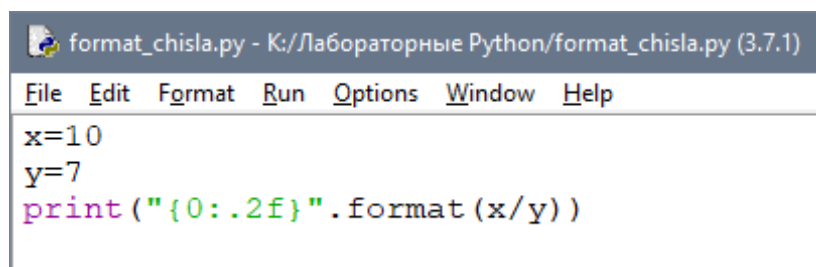
Спецификация формата:

Тип	Значение
'd', 'i', 'u'	Десятичное число.
'o'	Число в восьмеричной системе счисления.
'x'	Число в шестнадцатеричной системе счисления (буквы в нижнем регистре).

'X'	Число в шестнадцатеричной системе счисления (буквы в верхнем регистре).
'e'	Число с плавающей точкой с экспонентой (экспонента в нижнем регистре).
'E'	Число с плавающей точкой с экспонентой (экспонента в верхнем регистре).
'f', 'F'	Число с плавающей точкой (обычный формат).
'g'	Число с плавающей точкой. с экспонентой (экспонента в нижнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
'G'	Число с плавающей точкой. с экспонентой (экспонента в верхнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
'c'	Символ (строка из одного символа или число - код символа).
's'	Строка.
'%'	Число умножается на 100, отображается число с плавающей точкой, а за ним знак %.

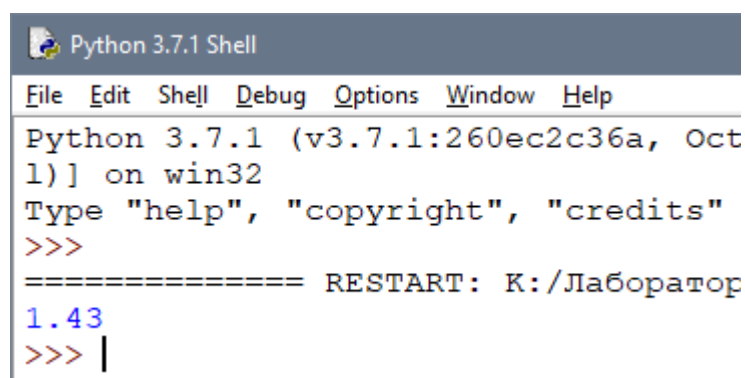
Для форматирования вещественных чисел с плавающей точкой используется следующая команда:

```
print('{0:.2f}'.format(вещественное число))
```



```
format_chisla.py - K:/Лабораторные Python/format_chisla.py (3.7.1)
File Edit Format Run Options Window Help
x=10
y=7
print("{0:.2f}".format(x/y))
```

В результате выведется число с двумя знаками после запятой.



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 1) on win32
Type "help", "copyright", "credits"
>>>
===== RESTART: K:/Лаборатор
1.43
>>> |
```

Пример.

Напишите программу, которая запрашивала бы у пользователя:

- ФИО ("Ваши фамилия, имя, отчество?")
- возраст ("Сколько Вам лет?")
- место жительства ("Где вы живете?")

После этого выводила бы три строки: "Ваше имя"

"Ваш возраст""Вы живете в"

Решение.

```
a=input('Введите ваши фамилию, имя, отчество ')
b=input('Сколько вам лет? ')
c=input('Где вы живёте? ')
print('Ваше имя ',a)
print('Ваш возраст ',b)
print('Вы живете в ',c)
```

```
Введите ваши фамилию, имя, отчество Иванов Иван Иванович
Сколько вам лет? 15
Где вы живёте? Уссурийск
Ваше имя Иванов Иван Иванович
Ваш возраст 15
Вы живете в Уссурийск
```

Задания для самостоятельной работы.

- 1) Установите Python <https://www.python.org/>
- 2) Напишите программу, которая запрашивала бы у пользователя: Имя,

Фамилия, Возраст, Место жительства

- фамилия, имя ("Ваши фамилия, имя?")
- возраст ("Сколько Вам лет?")
- место жительства ("Где вы живете?")

После этого выводила бы три строки: "Ваши фамилия, имя"

"Ваш возраст""Вы живете в"

Лабораторная работа 2. Математические операции в Python.

Цель работы: познакомиться с основными математическими операциями в Python.

Язык Python, благодаря наличию огромного количества библиотек для решения разного рода вычислительных задач, сегодня является конкурентом таким пакетам как Matlab и Octave. Запущенный в интерактивном режиме, он, фактически, превращается в мощный калькулятор. В этом уроке речь пойдет об арифметических операциях, доступных в данном языке. Арифметические операции изучим применительно к числам.

Если в качестве операндов некоторого арифметического выражения используются только целые числа, то результат тоже будет целое число. Исключением является операция деления, результатом которой является вещественное число. При совместном использовании целочисленных и вещественных переменных, результат будет вещественным.

В этом уроке речь пойдет об арифметических операциях, доступных в данном языке.

Если в качестве операндов некоторого арифметического выражения используются только целые числа, то результат тоже будет целое число. Исключением является операция деления, результатом которой является вещественное число. При совместном использовании целочисленных и вещественных переменных, результат будет вещественным.

Целые числа (int).

Числа в Python 3 поддерживают набор самых обычных математических операций:

$x + y$	Сложение
$x - y$	Вычитание
$x * y$	Умножение
x / y	Деление

<code>x // y</code>	Получение целой части от деления
<code>x % y</code>	Остаток от деления
<code>-x</code>	Смена знака числа
<code>abs(x)</code>	Модуль числа
<code>divmod(x, y)</code>	Пара (<code>x // y</code> , <code>x % y</code>)
<code>x ** y</code>	Возведение в степень
<code>pow(x, y[, z])</code>	<p><code>x</code> : Число, которое требуется возвести в степень.</p> <p><code>y</code> : Число, являющееся степенью, в которую нужно возвести первый аргумент. Если число отрицательное или одно из чисел "<code>x</code>" или "<code>y</code>" не целые, то аргумент "<code>z</code>" не принимается.</p> <p><code>z</code> : Число, на которое требуется произвести деление по модулю. Если число указано, ожидается, что "<code>x</code>" и "<code>y</code>" положительны и имеют тип <code>int</code>.</p>

Пример применения вышеописанных операций над целыми числами

```

x = 5
y = 2
z = 3
x+y = 7
x-y = 3
x*y = 10
x/y = 2.5
x//y = 2
x%y = 1
-x = -5
abs(-x) = 5
divmod(x,y) = (2, 1)
x**y = 25
pow(x,y,z) = 1

```

Вещественные числа (float).

Вещественные числа поддерживают те же операции, что и целые. Однако (из-за представления чисел в компьютере) вещественные числа неточны, и это может привести к ошибкам.

Пример применения вышеописанных операций над вещественными числами.

```

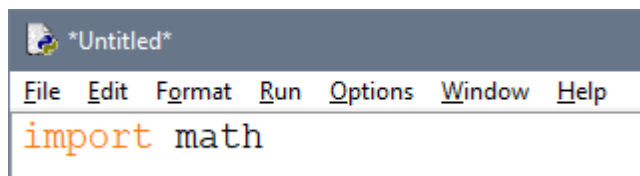
x = 5.5
y = 2.3
x+y = 7.8
x-y = 3.2
x*y = 12.649999999999999
x/y = 2.3913043478260874
x//y = 2.0
x%y = 0.9000000000000004
-x= -5.5
abs(-x) = 5.5
divmod(x,y) = (2.0, 0.9000000000000004)
x**y = 50.44686540422945

```

Библиотека (модуль) **math**.

В стандартную поставку Python входит библиотека **math**, в которой содержится большое количество часто используемых математических функций.

Для работы с данным модулем его предварительно нужно импортировать.



Рассмотрим наиболее часто используемые функции модуля **math**.

math.ceil(x)	Возвращает ближайшее целое число большее, чем x
math.fabs(x)	Возвращает абсолютное значение числа x
math.factorial(x)	Вычисляет факториал x
math.floor(x)	Возвращает ближайшее целое число меньшее, чем x
math.exp(x)	Вычисляет $e^{**}x$
math.log2(x)	Логарифм по основанию 2
math.log10(x)	Логарифм по основанию 10
math.log(x[,base])	По умолчанию вычисляет логарифм по основанию e, дополнительно можно указать основание логарифма
math.pow(x, y)	Вычисляет значение x в степени y
math.sqrt(x)	Корень квадратный от x

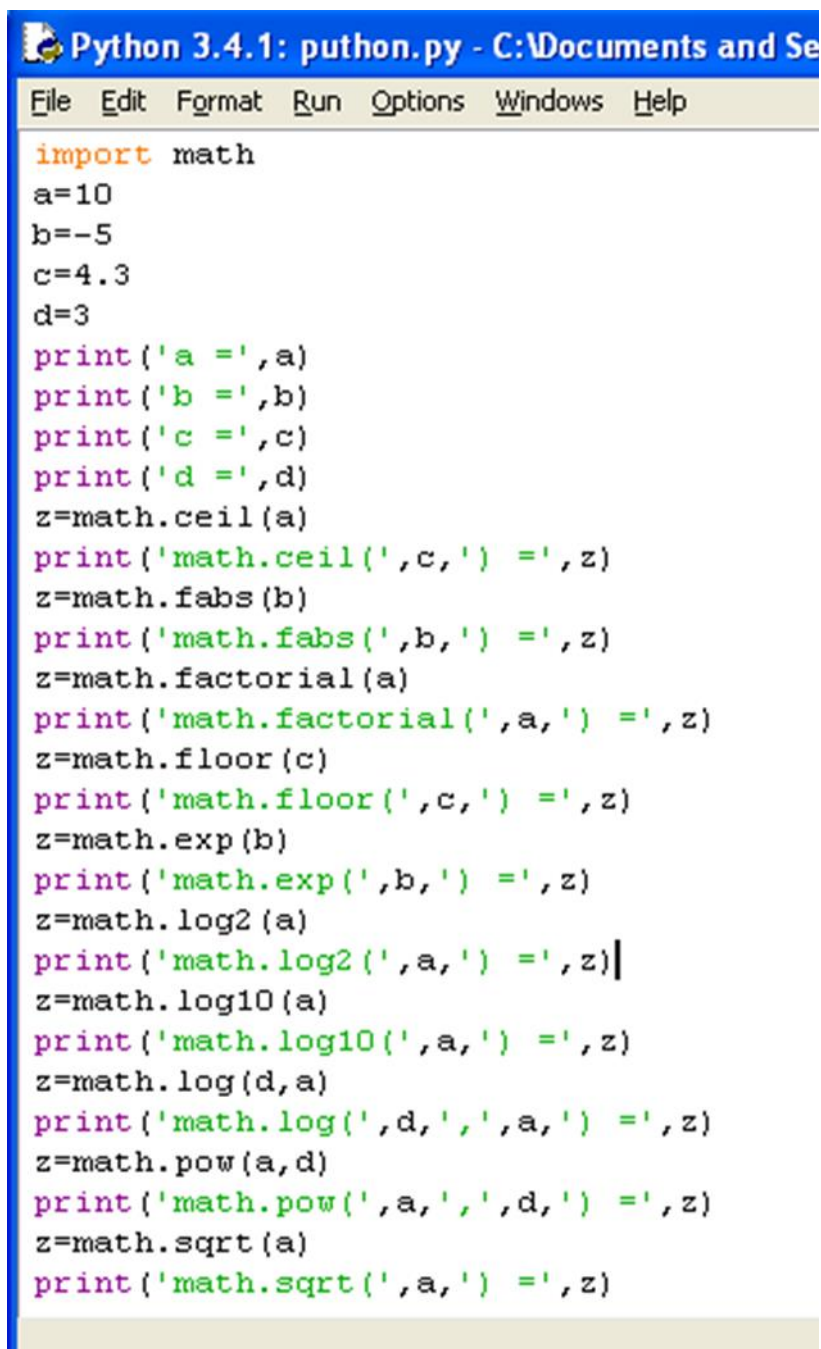
Пример применения вышеописанных функций над числами

В программе определены 4 переменные - a, b, c, d, каждая из которых является либо целым числом, либо вещественным, либо отрицательным.

Командой `print()` выводится значение каждой переменной на экран при выполнении программы.

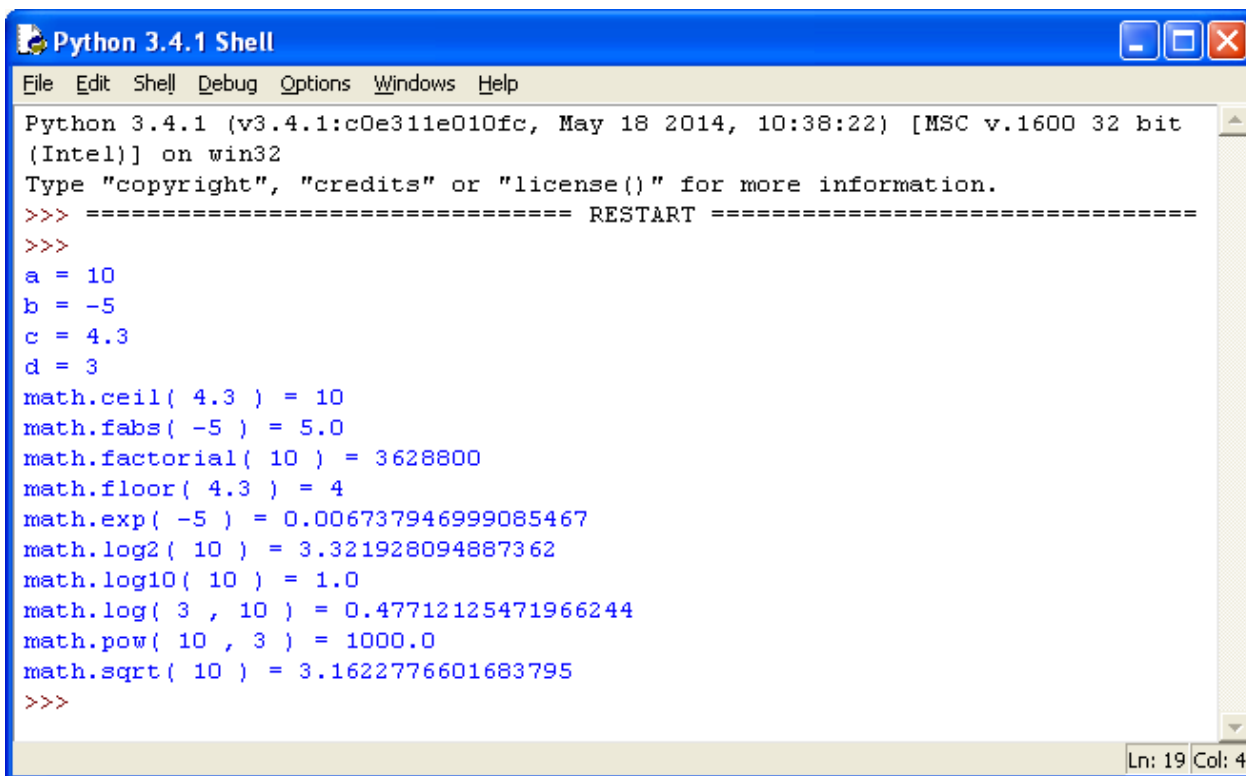
В переменную `z` помещается результат выполнения функции модуля `math`.

Затем командой `print()` выводится сообщение в виде используемой функции и её аргумента и результат её выполнения.



```
Python 3.4.1: puthon.py - C:\Documents and Se
File Edit Format Run Options Windows Help
import math
a=10
b=-5
c=4.3
d=3
print('a =',a)
print('b =',b)
print('c =',c)
print('d =',d)
z=math.ceil(a)
print('math.ceil(',c,') =',z)
z=math.fabs(b)
print('math.fabs(',b,') =',z)
z=math.factorial(a)
print('math.factorial(',a,') =',z)
z=math.floor(c)
print('math.floor(',c,') =',z)
z=math.exp(b)
print('math.exp(',b,') =',z)
z=math.log2(a)
print('math.log2(',a,') =',z)|
z=math.log10(a)
print('math.log10(',a,') =',z)
z=math.log(d,a)
print('math.log(',d,',',a,') =',z)
z=math.pow(a,d)
print('math.pow(',a,',',d,') =',z)
z=math.sqrt(a)
print('math.sqrt(',a,') =',z)
```

Пример программы на Python.



```
Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
a = 10
b = -5
c = 4.3
d = 3
math.ceil( 4.3 ) = 10
math.fabs( -5 ) = 5.0
math.factorial( 10 ) = 3628800
math.floor( 4.3 ) = 4
math.exp( -5 ) = 0.006737946999085467
math.log2( 10 ) = 3.321928094887362
math.log10( 10 ) = 1.0
math.log( 3 , 10 ) = 0.47712125471966244
math.pow( 10 , 3 ) = 1000.0
math.sqrt( 10 ) = 3.1622776601683795
>>>
```

Результат выполнения программы с применением функций модуля math

Тригонометрические функции модуля math.

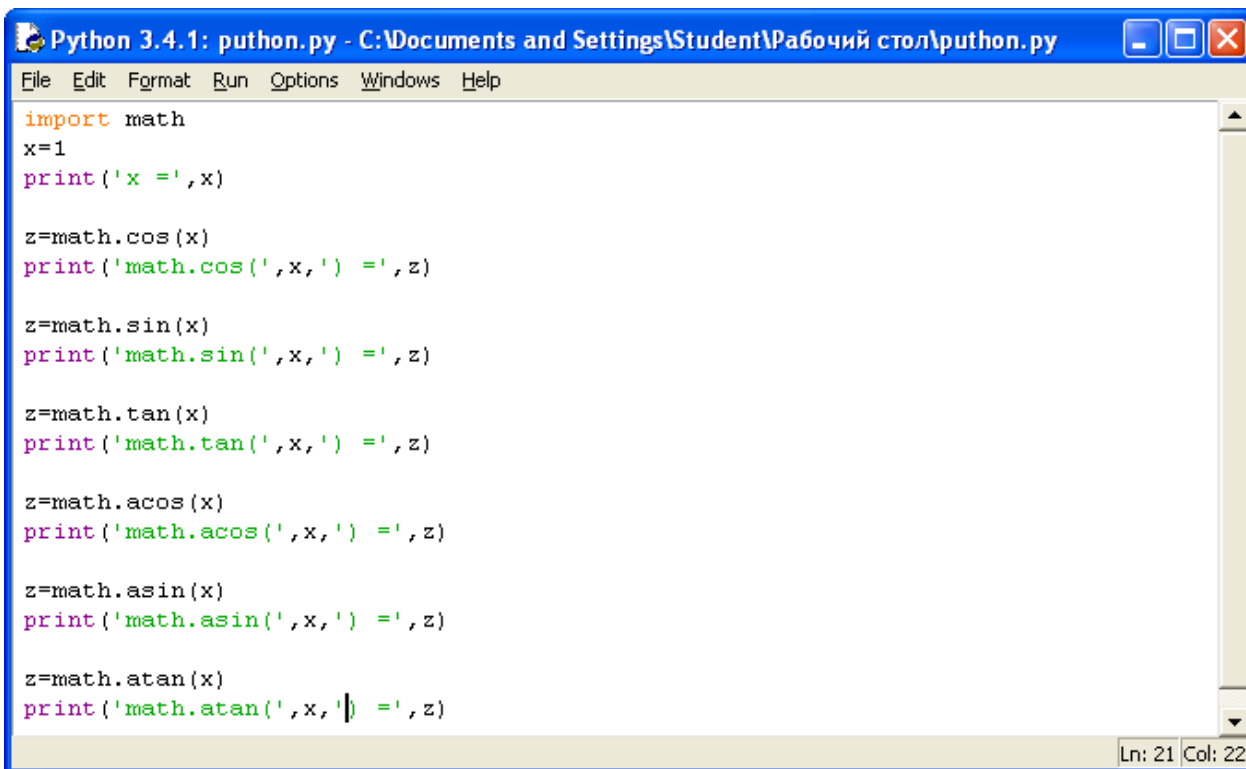
math.cos(x)	Возвращает cos числа X
math.sin(x)	Возвращает sin числа X
math.tan(x)	Возвращает tan числа X
math.acos(x)	Возвращает acos числа X
math.asin(x)	Возвращает asin числа X
math.atan(x)	Возвращает atan числа X

1 Пример применения вышеописанных функций над числами

В программе определена переменная x, содержащая целое число. Значение переменной выводится командой print() на экран.

В переменную z помещается результат выполнения тригонометрической функции модуля math.

Затем командой `print()` выводится сообщение в виде используемой функции и её аргумента и результат её выполнения.



```
Python 3.4.1: puthon.py - C:\Documents and Settings\Student\Рабочий стол\puthon.py
File Edit Format Run Options Windows Help
import math
x=1
print('x =', x)

z=math.cos(x)
print('math.cos(', x, ') =', z)

z=math.sin(x)
print('math.sin(', x, ') =', z)

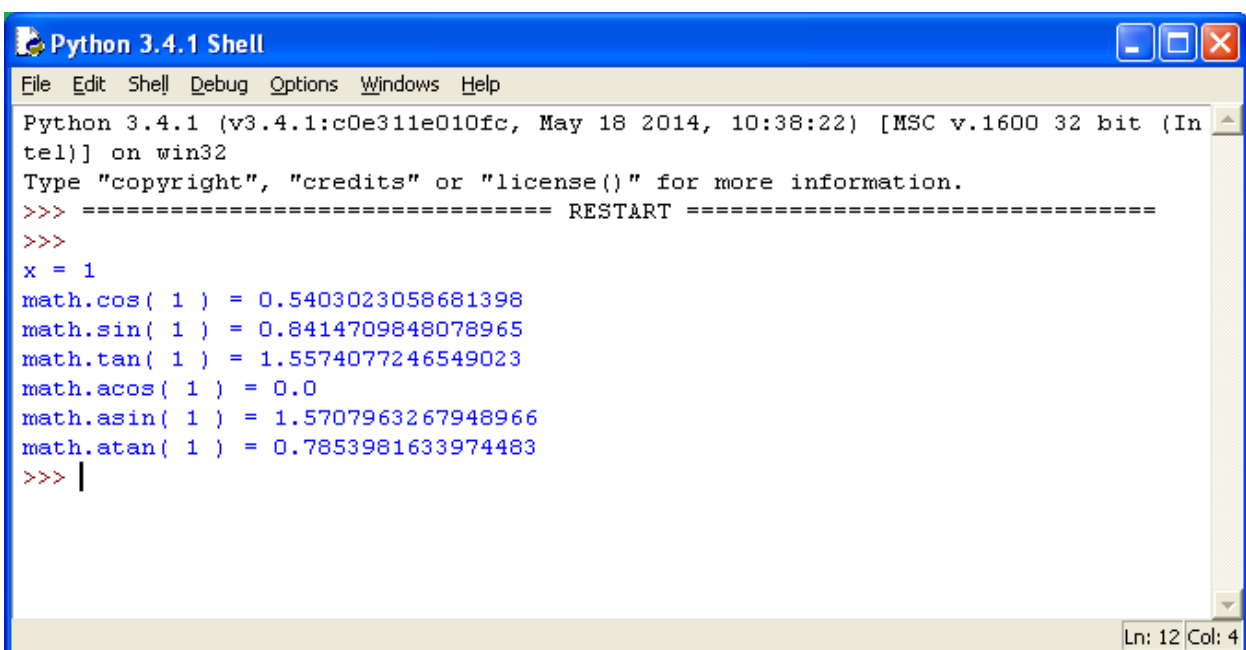
z=math.tan(x)
print('math.tan(', x, ') =', z)

z=math.acos(x)
print('math.acos(', x, ') =', z)

z=math.asin(x)
print('math.asin(', x, ') =', z)

z=math.atan(x)
print('math.atan(', x, ') =', z)
Ln: 21 Col: 22
```

Пример программы с использованием тригонометрических функций модуля `math`.



```
Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
x = 1
math.cos( 1 ) = 0.5403023058681398
math.sin( 1 ) = 0.8414709848078965
math.tan( 1 ) = 1.5574077246549023
math.acos( 1 ) = 0.0
math.asin( 1 ) = 1.5707963267948966
math.atan( 1 ) = 0.7853981633974483
>>> |
Ln: 12 Col: 4
```

Результат выполнения программы с применением тригонометрических функций модуля `math`

Константы:

math.pi - число Pi.

math.e - число e (экспонента).

Пример.

Напишите программу, которая бы вычисляла заданное арифметическое выражение при заданных переменных. Ввод переменных осуществляется с клавиатуры. Вывести результат с 2-мя знаками после запятой.

Задание.

$$Z = \frac{9\pi t + 10 \cos(x)}{\sqrt{t} - |\sin(t)|} * e^x$$

x=10; t=1

Решение.

Сначала импортируем модуль math. Для этого воспользуемся командой `import math`.

Затем следует ввести значения двух переменных целого типа x и t.

Для ввода данных используется команда `input`, но так как в условии даны целые числа, то нужно сначала определить тип переменных: `x=int()`, `t=int()`.

Определив тип переменных, следует их ввести, для этого в скобках команды `int()` нужно написать команду `input()`.

Для переменной x это выглядит так: `x=int(input("сообщение при вводе значения"))`.

Для переменной t аналогично: `t=int(input("сообщение при вводе значения"))`. Следующий шаг - это составление арифметического выражения, результат которого поместим в переменную z.

Сначала составим числитель. Выглядеть он будет так: `9*math.pi*t+10*math.cos(x)`.

Затем нужно составить знаменатель, при этом обратим внимание на то,

что числитель делится на знаменатель, поэтому и числитель и знаменатель нужно поместить в скобки (), а между ними написать знак деления /.

Выглядеть это будет так: $(9*\text{math.pi}*t+10*\text{math.cos}(x))/(\text{math.sqrt}(t)-\text{math.fabs}(\text{math.sin}(t)))$.

Последним шагом является умножение дроби на экспоненту в степени x.

Так как умножается вся дробь, то следует составленное выражение поместить в скобки (), а уже потом написать функцию $\text{math.pow}(\text{math.e},x)$.

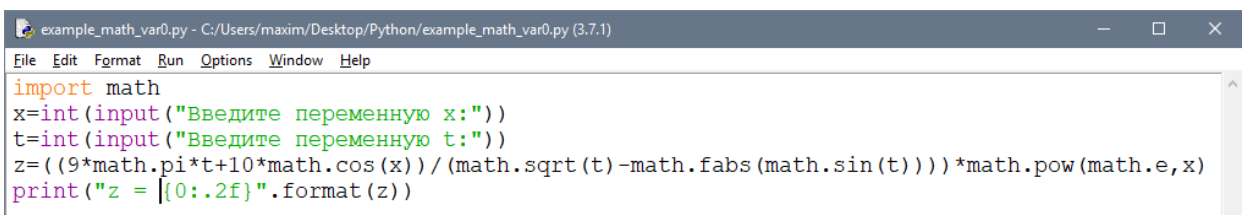
В результате выражение будет иметь вид:
 $z=((9*\text{math.pi}*t+10*\text{math.cos}(x))/(\text{math.sqrt}(t)-\text{math.fabs}(\text{math.sin}(t))))*\text{math.pow}(\text{math.e},x)$.

При составлении данного выражения следует обратить внимание на количество открывающихся и закрывающихся скобок.

Командой `print()` выведем значение переменной, отформатировав его командой `format`.

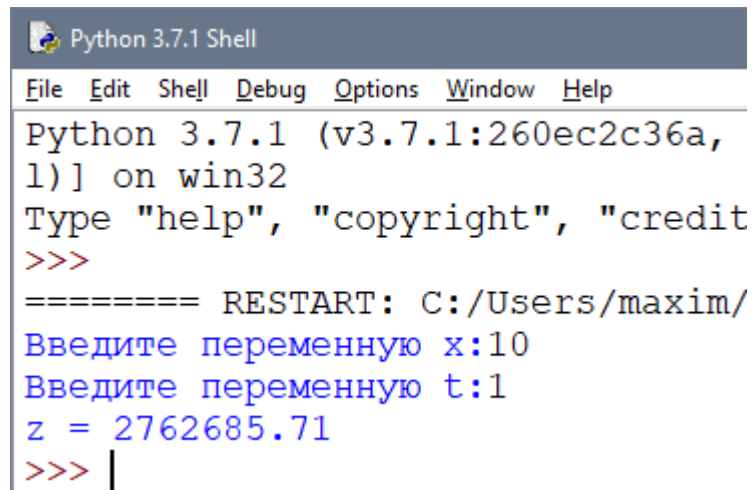
Сам формат записывается в апострофах в фигурных скобках {}.

В задаче требуется вывести число с двумя знаками после запятой, значит вид формата будет выглядеть следующим образом: `{0:.2f}`, где 2 - это количество знаков после запятой, а f указывает на то, что форматируется вещественное число. При этом перед 2 нужно поставить точку, указав тем самым на то, что форматируем именно дробную часть числа.



```
example_math_var0.py - C:/Users/maxim/Desktop/Python/example_math_var0.py (3.7.1)
File Edit Format Run Options Window Help
import math
x=int(input("Введите переменную x:"))
t=int(input("Введите переменную t:"))
z=((9*math.pi*t+10*math.cos(x))/(math.sqrt(t)-math.fabs(math.sin(t))))*math.pow(math.e,x)
print("z = {}".format(z))
```

Результат.



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a,
1)] on win32
Type "help", "copyright", "credit
>>>
===== RESTART: C:/Users/maxim/
Введите переменную x:10
Введите переменную t:1
z = 2762685.71
>>> |
```

Задания для самостоятельной работы.

Воспроизвести задание из примера. Сделать скриншоты кода и результата.

Лабораторная работа 3. Структура ветвление в Python.

Цель работы: познакомиться со структурой ветвление (if, if-else, if-elif-else).

Научиться работать с числами и строками используя данную структуру.

Условный оператор ветвления if, if-else, if-elif-else.

Оператор ветвления if позволяет выполнить определенный набор инструкций в зависимости от некоторого условия. Возможны следующие варианты использования.

1. Конструкция if

Синтаксис оператора if выглядит так:

if логическое выражение:

команда_1 команда_2

...

команда_n

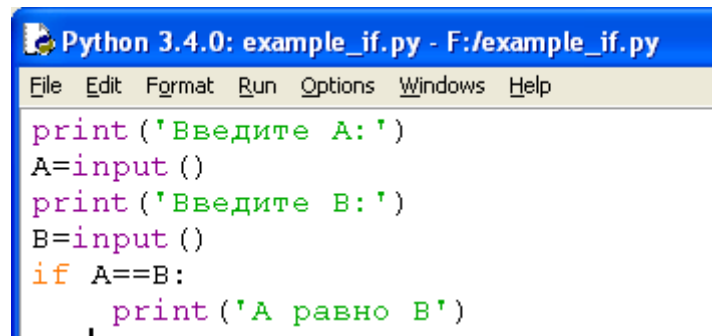
После оператора if записывается логическое выражение.

Логическое выражение — конструкция языка программирования, результатом вычисления которой является «истина» или «ложь».

Если это выражение истинно, то выполняются инструкции, определяемые данным оператором. Выражение является истинным, если его результатом является число не равное нулю, непустой объект, либо логическое True. После выражения нужно поставить двоеточие “:”.

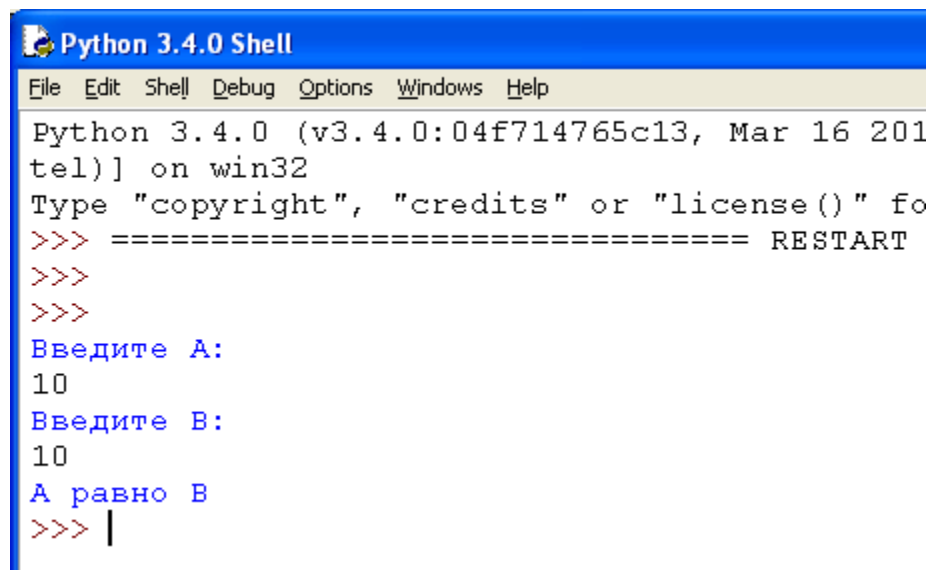
ВАЖНО: блок кода, который необходимо выполнить, в случае истинности выражения, отделяется **четырьмя** пробелами слева!

Программа запрашивает у пользователя два числа, затем сравнивает их и если числа равны, то есть логическое выражение $A==B$ истинно, то выводится соответствующее сообщение.



```
Python 3.4.0: example_if.py - F:/example_if.py
File Edit Format Run Options Windows Help
print ('Введите A:')
A=input ()
print ('Введите B:')
B=input ()
if A==B:
    print ('A равно B')
```

Пример программы на Python.



```
Python 3.4.0 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 201
tel)] on win32
Type "copyright", "credits" or "license()" fo
>>> ===== RESTART
>>>
>>>
Введите A:
10
Введите B:
10
A равно B
>>> |
```

Результат выполнения программы с использованием условного оператора if.

2. Конструкция if – else

Бывают случаи, когда необходимо предусмотреть альтернативный вариант выполнения программы. Т.е. при истинном условии нужно выполнить один набор

инструкций, при ложном – другой. Для этого используется конструкция if – else.

Синтаксис оператора if – else выглядит так:

2 if логическое выражение:

команда_1

команда_2

...

команда_n

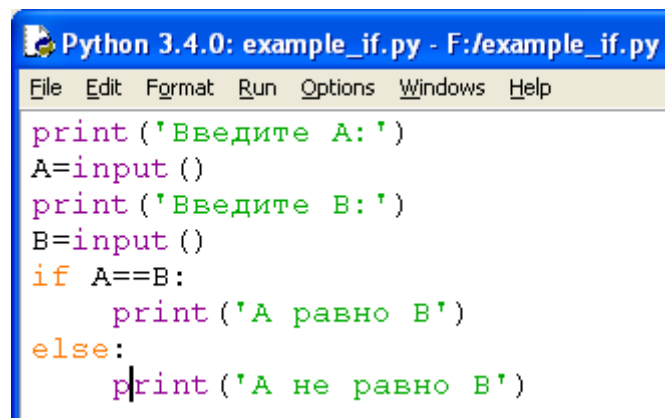
3 else:

команда_1 команда_2

...

команда_n

Программа запрашивает у пользователя два числа, затем сравнивает их и если числа равны, то есть логическое выражение $A==B$ истинно, то выводится соответствующее сообщение. В противном случае выводится сообщение, что числа не равны.



```
Python 3.4.0: example_if.py - F:/example_if.py
File Edit Format Run Options Windows Help
print ('Введите A:')
A=input ()
print ('Введите B:')
B=input ()
if A==B:
    print ('A равно B')
else:
    print ('A не равно B')
```

Пример программы на Python.

```
>>> ===== RESTART
>>>
Введите A:
10
Введите B:
5
A не равно B
>>> |
```

Результат выполнения программы с использованием условного оператора if-else

3. Конструкция if – elif – else

Для реализации выбора из нескольких альтернатив можно использовать конструкцию if – elif – else.

Синтаксис оператора if – elif – else выглядит так:

4 if логическое выражение_1:

команда_1 команда_2

...

команда_n

5 elif логическое выражение_2:

команда_1 команда_2

...

команда_n

6 elif логическое выражение_3:

команда_1 команда_2

...

команда_n

7 else:

команда_1 команда_2

...

команда_n

Программа запрашивает число у пользователя и сравнивает его с нулём $a < 0$.

Если оно меньше нуля, то выводится сообщение об этом. Если первое логическое выражение не истинно, то программа переходит ко второму - $a == 0$. Если оно истинно, то программа выведет сообщение, что число равно нулю, в противном случае, если оба вышеуказанных логических выражения оказались ложными, то программа выведет сообщение, что введённое число больше нуля.

```
Python 3.4.0: example_if.py - F:/example_if.py
File Edit Format Run Options Windows Help
a = int(input("Введите число:"))
if a < 0:
    print(a, " меньше нуля")
elif a == 0:
    print(a, " равно нулю")
else:
    print(a, " больше нуля")
|
```

Пример программы на Python.

```
Введите число:41
41 больше нуля
>>> ===== RESTART =
>>>
Введите число:-5
-5 меньше нуля
>>> ===== RESTART =
>>>
Введите число:0
0 равно нулю
>>>
```

Результат выполнения программы с использованием условного оператора if-elif-else

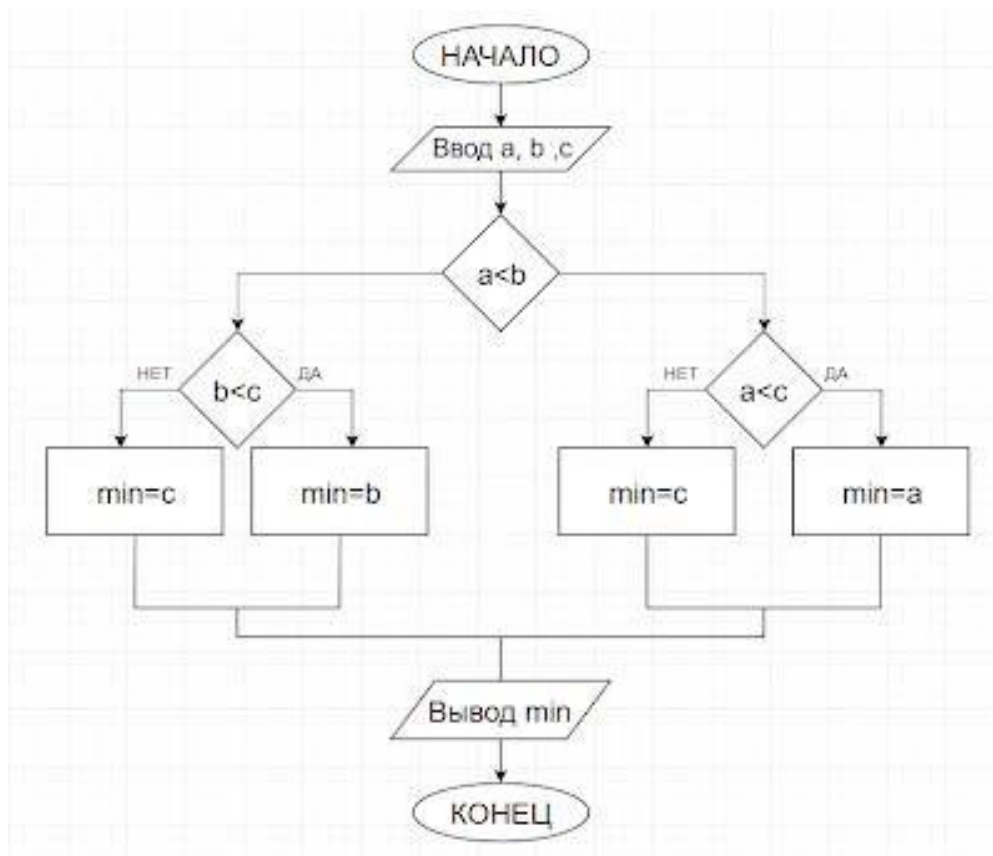
Пример.

Задание.

Дано 3 числа. Найти минимальное среди них и вывести на экран.

Решение.

Для простоты построим блок-схему задачи.



Командами

`a=input()`

`b=input()`

`c=input()`

введём три числа, присвоив значения переменным a, b, c.

Условной конструкцией if-else проверим на истинность логическое выражение $a < b$. Если оно истинно, то переходим на проверку логического выражения $a < c$. Если оно истинно, то переменной "y" присвоим значение переменной "a", т.е. "a" будет минимальным, а иначе "y" присвоится значение переменной "c".

Если в начале логическое выражение $a < b$ оказалось ложным, то переходим на проверку другого логического выражения $b < c$.

Если оно истинно, то "y" присвоится значение переменной "b", иначе "c". Командой `print()` выводим минимальное значение.

```
#нахождение минимального из 3-х чисел
a=input('Введите целое число \n')
b=input('Введите целое число \n')
c=input('Введите целое число \n')
if a<b:
    if a<c:
        y=a
    else:
        y=c
else:
    if b<c:
        y=b
    else:
        y=c
print('Минимальное:', y)
```

Пример программы.

```
Введите целое число
2
Введите целое число
5
Введите целое число
1
Минимальное: 1
```

Результат выполнения программы.

Задания для самостоятельной работы.

Задание.

Даны три целых числа. Выбрать из них те, которые принадлежат интервалу[1,3].

Лабораторная работа 4. Работа со строками в Python.

Цель работы: Изучить способы создания строк и функции для работы с ними: слияние строк, изменение регистра, получение подстрок, выравнивание.

Строковый тип данных и ввод-вывод строк.

В программировании один из наиболее часто используемых типов данных - строки. В строковых данных можно хранить фамилии и имена, адреса, названия товаров и т.д. Чтобы задать в программе на языке Python строку необходимо заключить последовательность символов в двойные или одинарные кавычки:

```
a="У лукоморья дуб зеленый"b="" # Это пустая строка
```

Считать строковую переменную с клавиатуры можно при помощи стандартной функции `input`, при этом пользователь также должен заключить вводимую строку в кавычки. Пример:

```
a=input()
```

Операции со строками.

Главная операция со строками - их объединение, когда одна строка записывается после другой строки. Эта операция называется *конкатенацией* и для нее используется оператор `+`:

```
a="abc"b="xyz" c=a+b  
print c # Будет напечатано abcxyz
```

Также при помощи оператора `*` можно многократно повторять одну и ту же строку. "Умножать" строку можно только на натуральное число:

```
print "="*20 # Будет напечатано 20 знаков "=" подряд
```

В отличие от списков, изменять отдельные элементы строки (то есть символы) нельзя. Вместо этого можно пользоваться конкатенацией строк. Например, если хочется в строке `s` заменить первый символ на букву "a", то это можно сделать при помощи конкатенации строки "a" и всей строки за исключением первого символа: `s="a"+s[1:]`.

Рассмотрим пример программы, которая считывает строку с клавиатуры и находит в ней первое слово (то есть все символы до первого пробела).

```
s=input()          # 1
for i in range(len(s)): # 2
if s[i]==" ":      # 3
print s[:i]       # 4
break             # 5
```

В первой строке программы в переменную *s* считывается строка с клавиатуры. Затем организовывается цикл, в котором переменная *s* меняется от 0 до $\text{len}(s)-1$, то есть переменная *i* принимает подряд все номера символов в строке. В третьей строке программы проверяется, является ли *i*-й символ строки пробелом (то есть совпадает ли он со строкой из одного пробела). Если проверяемое условие истинно, то был найден первый слева строки пробел, на экран печатается первые *i* символов строки, то есть все символы с начала строки до найденного пробела, после чего выполнение цикла завершается инструкцией `break`.

Оборудование и материалы.

Персональный компьютер, среда разработки Python.

Указания по технике безопасности:

Соответствуют технике безопасности по работе с компьютерной техникой.

Задания.

Благодаря поддержке стандарта Unicode Python 3 может содержать символы любого языка мира, а также многие другие символы. Необходимость работы с этим стандартом была одной из причин изменения Python 2.

Строки являются первым примером последовательностей в Python. В частности, они представляют собой последовательности символов. В отличие от других языков, в Python строки являются неизменяемыми. Вы не можете изменить саму строку, но можете скопировать части строк в другую строку, чтобы получить тот же эффект.

Создаем строки с помощью кавычек.

Строка в Python создается заключением символов в одинарные или двойные

```
>>> 'Печенюшки'  
'Печенюшки'  
>>> "Семки"  
'Семки'
```

кавычки, как показано в следующем примере:

Интерактивный интерпретатор выводит на экран строки в одинарных кавычках, но все они обрабатываются одинаково. Зачем иметь два вида кавычек? Основная идея заключается в том, что вы можете создавать строки, содержащие кавычки. Внутри одинарных кавычек можно расположить двойные и наоборот.

Для задания строк можно использовать тройные кавычки, это удобно для

```
>>> роем = '''Товарищ, верь, пройдет она,  
и демократия, и гласность.  
И вот тогда госбезопасность  
Припомнит наши имена!'''
```

создания многострочного блока текста:

(Это стихотворение было введено в интерактивный интерпретатор, который поприветствовал нас символами >>> в первой строке и выводил символы ... до тех пор, пока мы не ввели последние тройные кавычки и не перешли к следующей строке.)

Если бы вы попробовали создать стихотворение с помощью одинарных кавычек,

Python выдал бы ошибку, когда бы вы перешли к следующей строке.

Если внутри тройных кавычек располагается несколько строк, символы конца строки будут сохранены внутри нее. Если перед строкой или после нее находятся пробелы, они также будут сохранены.

Вам может понадобиться работать с пустой строкой. В ней нет символов, но она совершенно корректна. Вы можете создать пустую строку с помощью любых

```
>>> ''  
''  
>>> '''  
'''  
>>> ''''''  
''''''  
>>> ''''''''  
''''''''
```

упомянутых ранее кавычек:

Зачем может понадобиться пустая строка? Иногда приходится компоновать строку из других строк и для этого нужно начать с чистого листа, то есть с пустой строки.

```
>>> men = 15
>>> base = ''
>>> base+=str(men)
>>> base+=' человек на сундук мертвеца'
>>> base
'15 человек на сундук мертвеца'
```

Преобразование типов данных с помощью функции str()

Вы можете преобразовывать другие типы данных Python в строки с

```
>>> str(98.6)
'98.6'
>>> str(1.0e4)
'10000.0'
>>> str(True)
'True'
```

помощью функции str():

Создаем управляющие символы с помощью символа \

Python позволяет вам создавать управляющие последовательности внутри строк, чтобы добиться эффекта, который по-другому было бы трудно выразить. Размещая перед символом обратный слеш (\), вы наделяете этот символ особым

```
>>> palindrome = 'Аргентина \nманит \nнегра'
>>> print(palindrome)
Аргентина
манит
негра
```

значением. Наиболее распространена последовательность \n, которая означает переход на новую строку. С ее помощью вы можете создать многострочные строки из однострочных. Наберите в командной строке EDLE:

Вы также увидите последовательность \t (табуляция), которая используется для выравнивания текста:

```
>>> print('\tКрасная шапочка')
      Красная шапочка
>>> print('Красная \tшапочка')
Красная      шапочка
>>> print('Красная шапочка\t')
Красная шапочка
```

В последней строке табуляция стоит в конце, ее вы, конечно, увидеть не можете. Если вам нужен обратный слеш, просто напечатайте два:

```
>>> speech = 'Сегодня нам понадобился обратный слеш: \\'
>>> print(speech)
Сегодня нам понадобился обратный слеш: \
```

Объединяем строки с помощью символа +

Вы можете объединить строки или строковые переменные в Python с

```
>>> 'Я обернулся посмотреть '+'не обернулась ли она'
      Я обернулся посмотреть не обернулась ли она'
```

помощью оператора +, как показано далее:

Можно также объединять строки (не переменные), просто расположив одну перед другой:

```
>>> "чтоб посмотреть, " "не обернулся ли я"
      'чтоб посмотреть, не обернулся ли я'
```

Не забывайте добавлять пробелы при объединении строк.

Размножаем строки с помощью символа *

Оператор * можно использовать для того, чтобы размножить строку.

```
>>> start = 'Раз-два '*4 + '\n'
>>> end = 'Проверка связи.'
>>> print(start + end)
Раз-два Раз-два Раз-два Раз-два
Проверка связи.
```

Попробуйте ввести в интерактивный интерпретатор следующие строки и посмотреть, что получится:

Извлекаем символ с помощью символов [].

Для того чтобы получить один символ строки, задайте смещение внутри квадратных скобок после имени строки. Смещение первого (крайнего слева) символа равно 0, следующего — 1 и т. д. Смещение последнего (крайнего справа) символа может быть выражено как -1, поэтому вам не придется считать, в таком

случае смещение последующих символов будет равно -2 , -3 и т. д.:

```
>>> letters = 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя'
>>> letters[0]
'a'
>>> letters[1]
'б'
>>> letters[-1]
'я'
>>> letters[-2]
'ю'
```

Если вы укажете смещение, равное длине строки или больше (помните,

```
>>> letters[33]
Traceback (most recent call last):
  File "<pyshell#137>", line 1, in <module>
    letters[33]
IndexError: string index out of range
```

смещения лежат в диапазоне от 0 до длины строки -1), сгенерируется исключение:

Поскольку строки неизменяемы, вы не можете вставить символ непосредственно в строку или изменить символ по заданному индексу. Попробуем изменить слово Маша на слово Даша и посмотрим, что произойдет:

```
>>> name = 'Маша'
>>> name[0] = 'Д'
Traceback (most recent call last):
  File "<pyshell#139>", line 1, in <module>
    name[0] = 'Д'
TypeError: 'str' object does not support item assignment
```

Вместо этого вам придется использовать комбинацию строковых функций

```
>>> name = 'Маша'
>>> name.replace('М', 'Д')
'Даша'
>>> 'Д' + name[1:]
'Даша'
```

вроде `replace()` или `slice` (которая будет рассмотрена далее):

Извлекаем подстроки с помощью оператора [start : end : step].

Из строки можно извлечь подстроку (часть строки) с помощью функции `slice`. Вы определяете `slice` с помощью квадратных скобок, смещения начала подстроки `start` и конца подстроки `end`, а также опционального размера шага `step`. Некоторые из этих параметров могут быть исключены. В подстроку будут

включены символы, расположенные начиная с точки, на которую указывает смещение start, и заканчивая точкой, на которую указывает смещение end.

- Оператор [:] извлекает всю последовательность от начала до конца.
- Оператор [start :] извлекает последовательность с точки, на которую указывает смещение start, до конца.
- Оператор [: end] извлекает последовательность от начала до точки, на которую указывает смещение end минус 1.
- Оператор [start : end] извлекает последовательность с точки, на которую указывает смещение start, до точки, на которую указывает смещение end минус 1.
- Оператор [start : end : step] извлекает последовательность с точки, на которую указывает смещение start, до точки, на которую указывает смещение end минус 1, опуская символы, чье смещение внутри подстроки кратно step.

Как и ранее, смещение слева направо определяется как 0, 1 и т. д., а справа налево

- как -1, -2 и т. д. Если вы не укажете смещение start, функция будет использовать в качестве его значения 0 (начало строки). Если вы не укажете смещение end, функция будет использовать конец строки.

```
>>> letters = 'абвгдеёжзийклмнопрстуфхцчщцъьэя'
```

Создадим строку, содержащую русские буквы в нижнем регистре:

Использование простого двоеточия аналогично использованию последовательности 0: (целая строка):

```
>>> letters[:]  
'абвгдеёжзийклмнопрстуфхцчщцъьэя'
```

```
>>> letters[20:]  
'уфхцчщцъьэя'
```

Вот так можно получить все символы, начиная с 20-го и заканчивая последним:

А теперь получим символы с 12-го по 14-й (Python не включает символ, расположенный под номером, который указан последним):

```
>>> letters[12:15]  
'лмн'
```

Последние три символа:

```
>>> letters[-3:]  
'эя'
```

В следующем примере мы начинаем со смещения 18 и идем до четвертого с конца символа. Обратите внимание на разницу с предыдущим примером, где старт с позиции -3 получал символ э. В этом примере конец диапазона -3 означает, что последним будет символ по адресу -4 — ь:

```
>>> letters[18:-3]  
'стуфхцчшщъь'
```

В следующем примере мы получаем символы, начиная с шестого с конца и

```
>>> letters[-6:-2]  
'ъьэ'
```

заканчивая третьим с конца:

Если вы хотите увеличить шаг, укажите его после второго двоеточия, как показано в нескольких следующих примерах.

Каждый седьмой символ с начала до конца:

```
>>> letters[::7]  
'ажфы'
```

Каждый третий символ, начиная со смещения 4 и заканчивая 19-м символом: Каждый четвертый символ, начиная с 19-го:

```
>>> letters[19::4]  
'тцью'  
>>> letters[:21:5]  
'аейоу'
```

Каждый пятый символ от начала до 20-го:

Опять же значение end должно быть на единицу больше, чем реальное смещение.

И это еще не все! Если задать отрицательный шаг, Python будет двигаться в обратную сторону. В следующем примере движение начинается с конца и заканчивается в начале, ни один символ не пропущен:

```
>>> letters[-1::-1]  
'яэыьгъшщцхфутсрпномлкйизжёедгвба'
```

Можно добиться того же результата, используя такой пример:

```
>>> letters[::-1]  
'яэыьгъшщцхфутсрпномлкйизжёедгвба'
```

Операция slice более мягко относится к неправильным смещениям, чем поиск по индексу. Если указать смещение меньше, чем начало строки, оно будет

обрабатываться как 0, а если указать смещение большее, чем конец строки, оно будет обработано как -1. Это показано в следующих примерах.

```
>>> letters[-50:]  
'абвгдеёжзийклмнопрстуфхцчшщъыьэя'
```

Начиная с -50-го символа и до конца:

Начиная с -51-го символа и заканчивая -50-м:

```
>>> letters[-51:-50]  
''
```

Получаем длину строки с помощью функции len().

До этого момента мы использовали специальные знаки препинания вроде +, чтобы манипулировать строками. Но существует не так уж много подобных функций. Теперь мы начнем использовать некоторые встроенные функции Python: именованные фрагменты кода, которые выполняют определенные операции.

```
>>> len(letters)  
33  
>>> empty = ""  
>>> len(empty)  
0
```

Функция len() подсчитывает символы в строке:

Функция len() используется также для других структур данных, которые будут рассмотрены далее.

Разделяем строку с помощью функции split()

В отличие от функции len() некоторые функции характерны только для строк. Для того чтобы использовать строковую функцию, введите имя строки, точку, имя функции и аргументы, которые нужны функции: строка.функция(аргументы). Более подробно о функциях мы будем говорить позже.

Вы можете использовать встроенную функцию split(), чтобы разбить строку на список небольших строк, основываясь на разделителе. Со списками вы познакомитесь в следующей лабораторной работе. Список — это последовательность значений, разделенных запятыми и окруженных квадратными скобками:

```
>>> todos = 'купить молоко, помыть полы, почитать ребёнку'  
>>> todos.split(',')  
['купить молоко', ' помыть полы', ' почитать ребёнку']
```

В предыдущем примере строка имела имя todos, а строковая функция

называлась `split()` и получала один аргумент `' '`. Если вы не укажете разделитель, функция `split()` будет использовать любую последовательность пробелов, а также символы новой строки и табуляцию:

```
>>> todos.split()
['купить', 'молоко,', 'помыть', 'поль,', 'почитать', 'ребёнку']
```

Если вы вызываете функцию `split` без аргументов, вам все равно нужно добавлять круглые скобки — именно так Python узнает, что вы вызываете функцию.

Объединяем строки с помощью функции `join()`.

Функция `join()` является противоположностью функции `split()`: она объединяет список строк в одну строку. Вызов функции выглядит немного запутанно, поскольку сначала вы указываете строку, которая объединяет остальные, а затем — список строк для объединения: `string.join(list)`. Для того чтобы объединить список строк `lines`, разделив их символами новой строки, вам нужно написать `'\n'.join(lines)`. В следующем примере мы объединим несколько имен в список, разделенный запятыми и пробелами:

```
>>> crypto_list = ['Йети', 'Полтергейст', 'Лохнесское чудовище']
>>> crypto_string = ','.join(crypto_list)
>>> print('Они обитают в лесах Нечерноземья: ', crypto_string)
Они обитают в лесах Нечерноземья: Йети,Полтергейст,Лохнесское чудовище
```

Прочие функции для работы со строками.

Python содержит большой набор функций для работы со строками. Рассмотрим принцип работы самых распространенных из них. Объектом для тестов станет следующее стихотворение:

```
>>> poem = '''Был этот мир глубокой тьмой окутан.
Да будет свет! И вот явился Ньютон.
Но Сатана недолго ждал реванша -
Пришел Эйнштейн, и стало всё как раньше.'''
```

Для начала получим первые 13 символов (их смещения лежат в диапазоне от 0 до

12):

```
>>> poem[:13]
'Был этот мир '
```

Сколько символов содержит это стихотворение? (Пробелы и символы новой строки учитываются.)

```
>>> len(поем)
145
```

```
Начинается ли стихотворение с буквосочетания Был этот?
>>> поем.startswith('Был этот')
True
```

Заканчивается ли оно буквосочетанием *всё как*?

```
>>> поем.endswith('всё как')
False
```

Найдем первое вхождение слова *мир*:

```
>>> word = 'мир'
>>> поем.find(word)
9
```

Найдем последнее вхождение буквы *и*:

```
>>> поем.rfind('и')
122
```

Сравните с первым:

```
>>> поем.find('и')
10
```

Сколько раз встречается буква *и*?

```
>>> поем.count('и')
4
```

Являются ли все символы стихотворения буквами или цифрами?

```
>>> поем.isalnum()
False
```

Нет, в стихотворении имеются еще и знаки препинания.

Регистр и выравнивание.

В этом разделе мы рассмотрим еще несколько примеров использования

```
>>> setup = 'американец, француз и русский попали на необитаемый остров...'
```

встроенных функций. В качестве подопытной выберем следующую строку:

Удалим символ «.» с обоих концов строки:

```
>>> setup.strip('.')
'американец, француз и русский попали на необитаемый остров'
```

Замечание:

Поскольку строки неизменяемы, ни один из этих примеров не изменяет строку `setup`. Каждый пример просто берет значение переменной `setup`, выполняет над ним некоторое действие, а затем возвращает результат как новую строку.

Замечание:

Поскольку строки неизменяемы, ни один из этих примеров не изменяет строку `setup`. Каждый пример просто берет значение переменной `setup`, выполняет над ним некоторое действие, а затем возвращает результат как новую строку.

Напишем первое слово с большой буквы:

```
>>> setup.capitalize()
'Американец, француз и русский попали на необитаемый остров...'
```

Напишем все слова с большой буквы:

```
>>> setup.title()
'Американец, француз И Русский Попали На Необитаемый Остров...'
```

Запишем все слова большими буквами:

```
>>> setup.upper()
'АМЕРИКАНЕЦ, ФРАНЦУЗ И РУССКИЙ ПОПАЛИ НА НЕОБИТАЕМЫЙ ОСТРОВ...'
```

Запишем все слова маленькими буквами:

```
>>> setup.lower()
'американец, француз и русский попали на необитаемый остров...'
```

Сменим регистры букв:

```
>>> setup.swapcase()
'АМЕРИКАНЕЦ, ФРАНЦУЗ И РУССКИЙ ПОПАЛИ НА НЕОБИТАЕМЫЙ ОСТРОВ...'
```

Если бы в строке были буквы в разных регистрах, они бы поменяли свой регистр на противоположный. В данном случае все буквы выведены в верхнем регистре.

Теперь мы поработаем с функциями выравнивания. Строка выравнивается внутри заданного количества пробелов (в данном примере 30). Отцентрируем строку в промежутке из 30 пробелов:

```
>>> setup = 'Заходит лошадь в бар...'  
>>> setup.center(30)
'   Заходит лошадь в бар...   '
```

Выровняем ее по левому краю:

```
>>> setup.ljust(30)
'Заходит лошадь в бар...     '
```

А теперь по правому:

```
>>> setup.rjust(30)
'           Заходит лошадь в бар...'
```

О форматировании и преобразовании строк мы более подробно поговорим далее.

Также далее будет затронуто использование символа `%` и функции `format()`.

8. Что будет выведено, если запросить срез переменной с русским

```
>>> letters[0:100]
```

алфавитом (33 буквы) следующим образом:

9. Для чего предназначена функция len()?

10. Для чего предназначена функция split()?

```
>>> todos = 'купить молоко, помыть полы, почитать ребёнку'  
>>> todos.split(',')
```

11. Что будет результатом выполнения команд ?

12. Что будет, если в предыдущем примере вызвать функцию split() без аргумента?

```
>>> todos.split()
```

13. Для чего предназначена функция join()? Приведите пример использования.

14. Для чего предназначена функция strip()?

15. Для чего предназначена функция capitalize? Что будет, если применить её к строке

‘переходи на тёмную сторону, юзернейм, у нас печеньки!’

16. Как изменится строка после применения функции capitalize?

17. Какая функция позволяет вывести все слова в строке с прописных букв?

Примените её к строке ‘переходи на тёмную сторону, юзернейм, у нас печеньки!’.

18. Для чего предназначена функция upper?

19. Создайте строку «Заходит лошадь в бар». Примените функции, которые изменяют её выравнивание. Продемонстрируйте результат преподавателю.

20. Для чего используется функция replace()? Приведите пример её использования.

Лабораторная работа 5. Создание типа данных «класс».

Цели занятия: работа посвящена исследованию вопроса объектно-ориентированному программированию в *Python*. Разобраны такие темы как создание объектов и классов, работа с конструктором, наследование и полиморфизм в *Python*.

1 Основные понятия объектно-ориентированного программирования.

Объектно-ориентированное программирование (ООП) является методологией разработки программного обеспечения, в основе которой лежит понятие класса и объекта, при этом сама программа создается как некоторая совокупность объектов, которые взаимодействуют друг с другом и с внешним миром. Каждый объект является экземпляром некоторого класса. Классы образуют иерархии. Более подробно о понятии ООП можно прочитать на [википедии](#).

Выделяют три основных “столпа” ООП- это инкапсуляция, наследование и полиморфизм.

Инкапсуляция.

Под инкапсуляцией понимается сокрытие деталей реализации, данных и т.п. от внешней стороны. Например, можно определить класс “холодильник”, который будет содержать следующие данные: производитель, объем, количество камер хранения, потребляемая мощность и т.п., и методы: открыть/закрыть холодильник, включить/выключить, но при этом реализация того, как происходит непосредственно включение и выключение пользователю вашего класса не доступна, что позволяет ее менять без опасения, что это может отразиться на использующей класс «холодильник» программе. При этом класс становится новым типом данных в рамках разрабатываемой программы. Можно создавать переменные этого нового типа, такие переменные называются объектами.

Наследование.

Под наследованием понимается возможность создания нового класса на базе существующего. Наследование предполагает наличие отношения “является” между классом наследником и классом родителем. При этом класс потомок будет

содержать те же атрибуты и методы, что и базовый класс, но при этом его можно (и нужно) расширять через добавление новых методов и атрибутов.

Примером базового класса, демонстрирующего наследование, можно определить класс “автомобиль”, имеющий атрибуты: масса, мощность двигателя, объем топливного бака и методы: завести и заглушить. У такого класса может быть потомок – “грузовой автомобиль”, он будет содержать те же атрибуты и методы, что и класс “автомобиль”, и дополнительные свойства: количество осей, мощность компрессора и т.п..

Полиморфизм.

Полиморфизм позволяет одинаково обращаться с объектами, имеющими однотипный интерфейс, независимо от внутренней реализации объекта. Например, с объектом класса “грузовой автомобиль” можно производить те же операции, что и с объектом класса “автомобиль”, т.к. первый является наследником второго, при этом обратное утверждение неверно (во всяком случае не всегда). Другими словами полиморфизм предполагает разную реализацию методов с одинаковыми именами. Это очень полезно при наследовании, когда в классе наследнике можно переопределить методы класса родителя.

2 Классы в *Python*.

Создание классов и объектов

Создание класса в *Python* начинается с инструкции *class*. Вот так будет выглядеть минимальный класс.

```
class C:
```

```
pass
```

Класс состоит из объявления (инструкция *class*), имени класса (в нашем случае это имя *C*) и тела класса, которое содержит атрибуты и методы (в нашем минимальном классе есть только одна инструкция *pass*).

Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:

```
имя_объекта = имя_класса()
```

Статические и динамические атрибуты класса.

Как уже было сказано выше, класс может содержать атрибуты и методы. Атрибут может быть статическим и динамическим (уровня объекта класса). Суть в том, что для работы со статическим атрибутом, вам не нужно создавать экземпляр класса, а для работы с динамическим – нужно. Пример:

```
class Rectangle:
    default_color = "green"
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

В представленном выше классе, атрибут *default_color* – это статический атрибут, и доступ к нему, как было сказано выше, можно получить не создавая объект класса *Rectangle*.

```
>>> Rectangle.default_color
'green'
```

width и *height* – это динамические атрибуты, при их создании было использовано ключевое слово *self*. Пока просто примите это как должное, более подробно про *self* будет рассказано ниже. Для доступа к *width* и *height* предварительно нужно создать объект класса *Rectangle*:

```
>>> rect = Rectangle(10, 20)
>>> rect.width
10
>>> rect.height
20
```

Если обратиться через класс, то получим ошибку:

```
>>> Rectangle.width
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
AttributeError: type object 'Rectangle' has no attribute 'width'
```

При этом, если вы обратитесь к статическому атрибуту через экземпляр класса, то все будет ОК, до тех пор, пока вы не попытаетесь его поменять.

Проверим ещё раз значение атрибута *default_color*:

```
>>> Rectangle.default_color
'green'
```

Присвоим ему новое значение:

```
>>> Rectangle.default_color = "red"
>>> Rectangle.default_color
'red'
```

Создадим два объекта класса *Rectangle* и проверим, что *default_color* у них совпадает:

```
>>> r1 = Rectangle(1,2)
>>> r2 = Rectangle(10, 20)
>>> r1.default_color
'red'
>>> r2.default_color
'red'
```

Если поменять значение *default_color* через имя класса *Rectangle*, то все будет ожидаемо: у объектов *r1* и *r2* это значение изменится, но если поменять его через экземпляр класса, то у экземпляра будет создан атрибут с таким же именем как статический, а доступ к последнему будет потерян:

Меняем *default_color* через *r1*:

```
>>> r1.default_color = "blue"
>>> r1.default_color
'blue'
```

При этом у *r2* остается значение статического атрибута:

```
>>> r2.default_color
'red'
>>> Rectangle.default_color
'red'
```

Вообще напрямую работать с атрибутами – не очень хорошая идея, лучше для этого использовать свойства.

Методы класса.

Добавим к нашему классу метод. Метод – это функция, находящаяся внутри класса и выполняющая определенную работу.

Методы бывают статическими, классовыми (среднее между статическими и обычными) и уровня класса (будем их называть просто словом метод). Статический метод создается с декоратором `@staticmethod`, классовый – с декоратором `@classmethod`, первым аргументом в него передается `cls`, обычный метод создается без специального декоратора, ему первым аргументом передается `self`:

```
class MyClass:  
    @staticmethod  
    def ex_static_method():  
        print("static method")  
    @classmethod  
    def ex_class_method(cls):  
        print("class method")  
    def ex_method(self):  
        print("method")
```

Статический и классовый метод можно вызвать, не создавая экземпляр класса, для вызова `ex_method()` нужен объект:

```
>>> MyClass.ex_static_method()  
static method  
>>> MyClass.ex_class_method()  
class method  
>>> MyClass.ex_method()  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>
```

```
TypeError: ex_method() missing 1 required positional argument: 'self'
```

```
>>> m = MyClass()
```

```
>>> m.ex_method()
```

```
method
```

Конструктор класса и инициализация экземпляра класса.

В *Python* разделяют конструктор класса и метод для инициализации экземпляра класса. Конструктор класса это метод `__new__(cls, *args, **kwargs)` для инициализации экземпляра класса используется метод `__init__(self)`. При этом, как вы могли заметить `__new__` – это классовый метод, а `__init__` таким не является. Метод `__new__` редко переопределяется, чаще используется реализация от базового класса *object* (см. раздел Наследование), `__init__` же наоборот является очень удобным способом задать параметры объекта при его создании.

Создадим реализацию класса *Rectangle* с измененным конструктором и инициализатором, через который задается ширина и высота прямоугольника:

```
class Rectangle:
    def __new__(cls, *args, **kwargs):
        print("Hello from __new__")
        return super().__new__(cls)
    def __init__(self, width, height):
        print("Hello from __init__")
        self.width = width
        self.height = height
>>> rect = Rectangle(10, 20)
Hello from __new__
Hello from __init__
>>> rect.width
10
>>> rect.height
20
```

Что такое *self* ?

До этого момента вы уже успели познакомиться с ключевым словом *self*. *self* – это ссылка на текущий экземпляр класса, в таких языках как *Java*, *C#* аналогом является ключевое слово *this*. Через *self* вы получаете доступ к атрибутам и методам класса внутри него:

```
class Rectangle:  
def __init__(self, width, height):  
    self.width = width  
    self.height = height  
def area(self):  
return self.width * self.height
```

В приведенной реализации метод *area* получает доступ к атрибутам *width* и *height* для расчета площади. Если бы в качестве первого параметра не было указано *self*, то при попытке вызвать *area* программа была бы остановлена с ошибкой.

Уровни доступа атрибута и метода.

Если вы знакомы с языками программирования *Java*, *C#*, *C++* то, наверное, уже задались вопросом: “а как управлять уровнем доступа?”. В перечисленных языках вы можете явно указать для переменной, что доступ к ней снаружи класса запрещен, это делается с помощью ключевых слов (*private*, *protected* и т.д.). В *Python* таких возможностей нет, и любой может обратиться к атрибутам и методам вашего класса, если возникнет такая необходимость. Это существенный недостаток этого языка, т.к. нарушается один из ключевых принципов ООП – инкапсуляция. Хорошим тоном считается, что для чтения/изменения какого-то атрибута должны использоваться специальные методы, которые называются *getter/setter*, их можно реализовать, но ничего не мешает изменить атрибут напрямую. При этом есть соглашение, что метод или атрибут, который

начинается с нижнего подчеркивания, является скрытым, и снаружи класса трогать его не нужно (хотя сделать это можно).

Внесем соответствующие изменения в класс *Rectangle*:

```
class Rectangle:
def __init__(self, width, height):
    self._width = width
    self._height = height
def get_width(self):
return self._width
def set_width(self, w):
    self._width = w
def get_height(self):
return self._height
def set_height(self, h):
    self._height = h
def area(self):
return self._width * self._height
```

В приведенном примере для доступа к *_width* и *_height* используются специальные методы, но ничего не мешает вам обратиться к ним (атрибутам) напрямую.

```
>>> rect = Rectangle(10, 20)
>>> rect.get_width()
10
>>> rect._width
10
```

Если же атрибут или метод начинается с двух подчеркиваний, то тут напрямую вы к нему уже не обратитесь (простым образом). Модифицируем наш класс *Rectangle*:

```
class Rectangle:
def __init__(self, width, height):
```



```
self.__width = width
self.__height = height
def get_width(self):
return self.__width
def set_width(self, w):
self.__width = w
def get_height(self):
return self.__height
def set_height(self, h):
self.__height = h
def area(self):
return self.__width * self.__height
```

Попытка обратиться к `__width` напрямую вызовет ошибку, нужно работать только через `get_width()`:

```
>>> rect = Rectangle(10, 20)
>>> rect.__width
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Rectangle' object has no attribute '__width'
>>> rect.get_width()
10
```

Но на самом деле это сделать можно, просто этот атрибут теперь для внешнего использования носит название: `__Rectangle__width`:

```
>>> rect.__Rectangle__width
10
>>> rect.__Rectangle__width = 20
>>> rect.get_width()
20
```

Свойства.

Свойством называется такой метод класса, работа с которым подобна работе с атрибутом. Для объявления метода свойством необходимо использовать декоратор `@property`.

Важным преимуществом работы через свойства является то, что вы можете осуществлять проверку входных значений, перед тем как присвоить их атрибутам.

Сделаем реализацию класса *Rectangle* с использованием свойств:

```
class Rectangle:
def __init__(self, width, height):
    self.__width = width
    self.__height = height
    @property
def width(self):
return self.__width
    @width.setter
def width(self, w):
if w > 0:
    self.__width = w
else:
raise ValueError
    @property
def height(self):
return self.__height
    @height.setter
def height(self, h):
if h > 0:
    self.__height = h
else:
raise ValueError
```

```
def area(self):  
return self.__width * self.__height
```

Теперь работать с *width* и *height* можно так, как будто они являются атрибутами:

```
>>> rect = Rectangle(10, 20)  
>>> rect.width  
10  
>>> rect.height  
20
```

Можно не только читать, но и задавать новые значения свойствам:

```
>>> rect.width = 50  
>>> rect.width  
50  
>>> rect.height = 70  
>>> rect.height  
70
```

Если вы обратили внимание: в *setter*'ах этих свойств осуществляется проверка входных значений, если значение меньше нуля, то будет выброшено исключение *ValueError*:

```
>>> rect.width = -10  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
File "test.py", line 28, in width  
raise ValueError  
ValueError
```

Наследование.

В организации наследования участвуют как минимум два класса: класс родитель и класс потомок. При этом возможно множественное наследование, в этом случае у класса потомка может быть несколько родителей. Не все языки программирования поддерживают множественное наследование, но

в *Python* можно его использовать. По умолчанию все классы в *Python* являются наследниками от *object*, явно этот факт указывать не нужно.

Синтаксически создание класса с указанием его родителя выглядит так:

Class имя_класса(имя_родителя1, [имя_родителя2,..., имя_родителя_n])

Переработаем наш пример так, чтобы в нем присутствовало наследование:

```
class Figure:
def __init__(self, color):
    self.__color = color
    @property
def color(self):
return self.__color
    @color.setter
def color(self, c):
    self.__color = c
class Rectangle(Figure):
def __init__(self, width, height, color):
    super().__init__(color)
    self.__width = width
    self.__height = height
    @property
def width(self):
return self.__width
    @width.setter
def width(self, w):
if w > 0:
    self.__width = w
else:
raise ValueError
    @property
def height(self):
```

```

return self.__height
@height.setter
def height(self, h):
if h > 0:
self.__height = h
else:
raise ValueError
def area(self):
return self.__width * self.__height

```

Родительским классом является *Figure*, который при инициализации принимает цвет фигуры и предоставляет его через свойства. *Rectangle* – класс наследник от *Figure*. Обратите внимание на его метод `__init__`: в нем первым делом вызывается конструктор (хотя это не совсем верно, но будем говорить так) его родительского класса:

```

super().__init__(color)

```

super – это ключевое слово, которое используется для обращения к родительскому классу.

Теперь у объекта класса *Rectangle* помимо уже знакомых свойств *width* и *height* появилось свойство *color*:

```

>>> rect = Rectangle(10, 20, "green")
>>> rect.width
10
>>> rect.height
20
>>> rect.color
'green'
>>> rect.color = "red"
>>> rect.color
'red'

```

Полиморфизм.

Как уже было сказано во введении в рамках ООП полиморфизм, как правило, используется с позиции переопределения методов базового класса в классе наследнике. Проще всего это рассмотреть на примере. Добавим в наш базовый класс метод *info()*, который печатает сводную информацию по объекту класса *Figure* и переопределим этот метод в классе *Rectangle*, добавим в него дополнительные данные:

```
class Figure:
    def __init__(self, color):
        self.__color = color
    @property
    def color(self):
        return self.__color
    @color.setter
    def color(self, c):
        self.__color = c
    def info(self):
        print("Figure")
        print("Color: " + self.__color)
class Rectangle(Figure):
    def __init__(self, width, height, color):
        super().__init__(color)
        self.__width = width
        self.__height = height
    @property
    def width(self):
        return self.__width
    @width.setter
    def width(self, w):
        if w > 0:
```

```
self.__width = w
else:
raise ValueError
@property
def height(self):
return self.__height
@height.setter
def height(self, h):
if h > 0:
self.__height = h
else:
raise ValueError
def info(self):
print("Rectangle")
print("Color: " + self.color)
print("Width: " + str(self.width))
print("Height: " + str(self.height))
print("Area: " + str(self.area()))
def area(self):
return self.__width * self.__height
```

Посмотрим, как это работает

```
>>> fig = Figure("orange")
>>> fig.info()
Figure
Color: orange
>>> rect = Rectangle(10, 20, "green")
>>> rect.info()
Rectangle
Color: green
Width: 10
```

Height: 20

Area: 200

Таким образом, класс наследник может расширять функционал класса родителя.

Индивидуальные задания.

Разработать программу на языке Python, в которой: создается класс, описывающий поведение объектов, представляющих таких персонажей:

1 – пользователей компьютера;

2 – литературных персонажей;

3 – студентов;

4 – героев мультипликации;

5 – исторических персонажей;

6 – персонажей художественных фильмов;

класс должен иметь следующие специальные методы: `__init__()`, `__str__()` и `__del__()`;

класс должен иметь такие атрибуты и/или методы:

1 – статический метод;

2 – атрибут класса;

3 – метод экземпляра класса;

4 – закрытый атрибут

5 – закрытый метод

осуществляется управление двумя атрибутами класса, для первого устанавливается режим "только чтение", для второго – согласно колонке "Управление" табл. 3):

1 выполняется чтение атрибута и запись в него;

2 выполняется чтение и удаление атрибута;

3 выполняется чтение, запись и удаление атрибута;

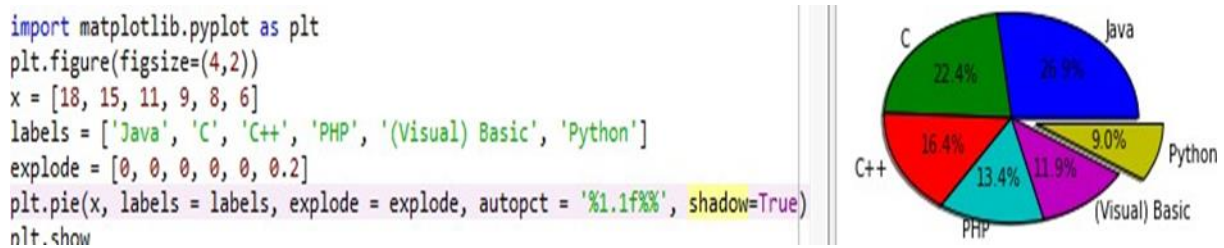
создаются объекты класса и проверяется их работа.

Лабораторная работа 6. Работа с графикой.

Цель работы: отработка навыков работы с графикой в приложениях.

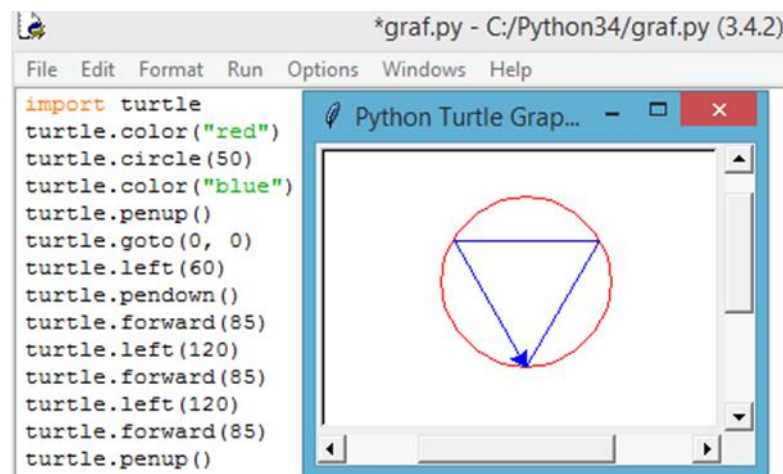
Методические указания к выполнению лабораторной работы

1. Листинг и результат реализации данных в виде круговой диаграммы PythonXY представлены на рисунке 10.



Реализация круговой диаграммы в PythonXY

2. Построить красный круг, вписать в него правильный треугольник. Листинг и результат реализации в Python представлены на рисунке 11.



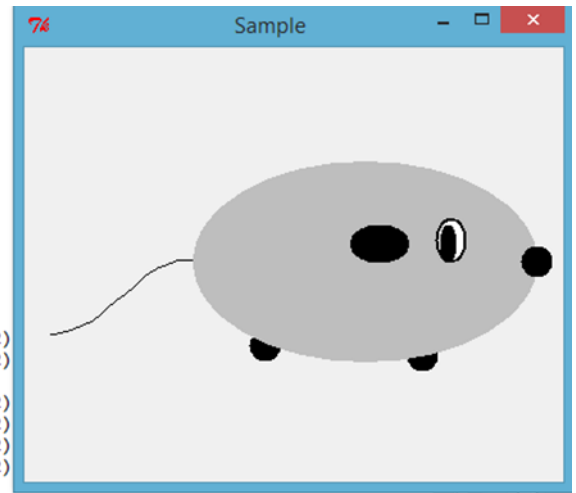
Реализация задачи в Python, рисование с помощью turtle

3. Нарисовать мышь. Листинг и результат реализации в PythonXY представлены на рисунке 12.

```

import Tkinter
import math
tk = Tkinter.Tk()
tk.title("Sample")
canvas = Tkinter.Canvas(tk)
canvas["height"] = 360
canvas["width"] = 480;
canvas["borderwidth"] = 2
canvas.pack()
points=[]
for n in range (-1, 7):
    y=math.sin(1-n)
    pp = (n*30+69, y*30+175)
    points.append(pp)
canvas.create_line(points, fill = "black", smooth = 1)
canvas.create_oval(300,206,320,226, fill="black", width=2)
canvas.create_oval(190,199,210,219, fill="black", width=2)
canvas.create_oval(150,80,390,220, fill="grey", width=0)
canvas.create_oval(380,140,400,160, fill="black", width=2)
canvas.create_oval(320,120,340,150, fill="white", width=2)
canvas.create_oval(323,125,333,150, fill="black", width=2)
canvas.create_oval(260,125,300,150, fill="black", width=2)
tk.mainloop()

```



Реализация мыши в PythonXY

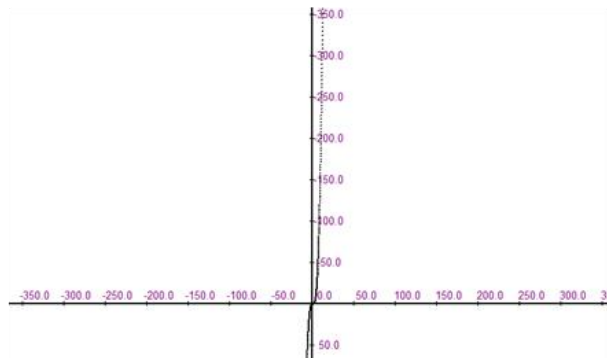
4. Построить график функции $y=(x^3+1)/5$.

Листинг и результатреализации в PythonXY представлены на рисунке 13.

```

from math import *
from tkinter import *
f = input('f(x):')
root = Tk()
canv = Canvas(root, width = 1000, height = 1000, bg = "white")
canv.create_line(500,1000,500,0,width=2,arrow=LAST)
canv.create_line(0,500,1000,500,width=2,arrow=LAST)
First_x = -500;
for i in range(16000):
    if (i % 800 == 0):
        k = First_x + (1 / 16) * i
        canv.create_line(k + 500, -3 + 500, k + 500, 3 + 500, width = 0.5, fill = 'black')
        canv.create_text(k + 515, -10 + 500, text = str(k), fill="purple", font=("Helvetica", "10"))
        if (k != 0):
            canv.create_line(-3 + 500, k + 500, 3 + 500, k + 500, width = 0.5, fill = 'black')
            canv.create_text(20 + 500, k + 500, text = str(k), fill="purple", font=("Helvetica", "10"))
    try:
        x = First_x + (1 / 16) * i
        new_f = f.replace('x', str(x))
        y = -eval(new_f) + 500
        x += 500
        canv.create_oval(x, y, x + 1, y + 1, fill = 'black')
    except:
        pass
canv.pack()
root.mainloop()

```



Реализация графика функции $y=(x^3+1)/5$

Индивидуальное задание.

В Python и PythonXY решить три задачи по вариантам.

Вариант 1.

- 1) Построить график функции $y=2+(9-x^2)/(6-x)$.
- 2) Нарисовать круговую диаграмму по данным продаж iPhone за год.
- 3) Нарисовать смайлик.

Вариант 2.

- 1) Построить график функции $y=(\sin x)/(x+2)$.
- 2) Нарисовать $y=x^3-7$.
- 3) Нарисовать флаг России.

Вариант 3.

- 1) Построить график функции $y=5/x$.
- 2) Построить две концентрические окружности с центром в точке (4,8).
- 3) Нарисовать дом.

Вариант 4.

- 1) Построить график функции $y=x/(2x^2+3x-1)$.
- 2) Нарисовать шахматную доску 4·4.
- 3) Нарисовать бабочку.

Вариант 5.

- 1) Построить график функции $y=2/(x-5)$.
- 2) Построить правильный 5-угольник и закрасить его.
- 3) Нарисовать ананас.

Вариант 6.

- 1) Построить график функции $y=(4x-2)/(8x^3-3x)$.
- 2) Построить квадрат и закрасить его красным цветом. На каждой стороне квадрата найти середину, соединить все середины сторон квадрата. Полученное закрасить зеленым цветом.
- 3) Нарисовать яблоко.

Вариант 7.

- 1) Построить график функции $y=(5-x) \cdot (6+2x)/(x-1)^2$.

2) Построить круг, закрасить. Вписать в него правильный треугольник.

Закрасить.

3) Нарисовать забор.

Вариант 8.

1) Построить график функции $y=x/(4x^2+2x-1)$.

2) Построить круг, разделить его на 6 секторов, закрасить разными цветами.

3) Нарисовать рыбу.

Вариант 9.

1) Построить график функции $y=(2-x)/(3+x)$.

2) Построить круг. Закрасить его синим цветом. Вписать в него квадрат желтого цвета.

3) Нарисовать паровоз.

Вариант 10.

1) Построить график функции $y=(2x+1)/x$.

2) Построить на экране множество точек, координаты которых удовлетворяют следующему неравенству $x^2+y^2 \leq 81$.

3) Нарисовать корабль.

Вариант 11.

1) Построить график функции $y=(x+4)/(x-1)$.

2) Нарисовать любой предмет ванны и его зеркальное отражение.

3) Нарисовать пианино.

Вариант 12.

1) Построить график функции $y=2x^2+3x$.

2) Построить квадрат со стороной a , у которого левая нижняя координата $(-3,-2)$.

3) Нарисовать собаку.

Вариант 13.

1) Построить график функции $y=\cos(1-x)/2$.

2) Построить треугольник со сторонами a , $a+1$, $a+2$.

3) Нарисовать птицу.

Вариант 14.

1) Построить график функции $y=3x^2-12$.

2) Нарисовать два разных дерева.

3) Нарисовать кошку.

Вариант 15.

1) Построить график функции $y=5/x + 4$.

2) Построить на экране множество точек, координаты которых удовлетворяют следующему неравенству $x^2+y^2 \leq 2(x+y)$.

3) Нарисовать летучую мышь.

Вариант 16.

1) Построить график функции $y=-x^2+25$

2) Нарисовать предмет кухни и его зеркальное отражение.

3) Нарисовать машину.

Вариант 17.

1) Построить график функции $y=8x^2-x+2$.

2) Построить круг, закрасить. Вписать в него 2 правильных треугольника. Закрасить разными цветами.

3) Нарисовать автобус.

Вариант 18.

1) Построить график функции $y=(8-2\cos x)/(3-x)$.

2) Построить круг, закрасить. Вписать в круг правильный семиугольник, закрасить. Соединить середины сторон семиугольника, полученную фигуру закрасить.

3) Нарисовать ежа.

Вариант 19.

1) Построить график функции $y=x^2-x^3$.

2) Построить два разных треугольника, соединить их точки симметрии.

3) Нарисовать здание АУЭС.

Вариант 20.

- 1) Построить график функции $y=x^3/3 - 1$.
- 2) Построить предмет столовой и его отражение.
- 3) Нарисовать велосипед.

Вариант 21.

- 1) Построить график функции $y=2x^2/7 + 2x - 4$.
- 2) Построить круговую диаграмму по данным статей расхода бюджета вашей семьи.
- 3) Нарисовать компьютер.

Список контрольных вопросов.

- 1 Tkinter – что это такое?
- 2 Какие библиотеки нужны для работы с использованием графики?
- 3 В каких средах Python можно работать с графическими файлами?