

# Лекция 2

Язык программирования Python.

Хайрулин Сергей Сергеевич

email: [s.khairulin@g.nsu.ru](mailto:s.khairulin@g.nsu.ru), [s.khayrulin@gmail.com](mailto:s.khayrulin@gmail.com)

Ссылка на [материалы](#)

# План

- Лекции/практические занятия
  - Тест
- Дифференцированный зачет в конце семестра
  - Защита задания

Все приведенные ниже примеры были выполнены в среде IPython (ну почти все)

# Литература

## **Начальный уровень**

- Mark Pilgrim. Dive into Python - <http://www.diveintopython.net/>
- Марк Лутц. Изучаем Python, 4-е издание // Символ-Плюс 2011.
- ...

## **Стандарт/Документация**

- PEP-8 - <https://www.python.org/dev/peps/pep-0008/>
- <https://www.python.org/>
- <https://github.com/python/cpython>

## **Экспертный уровень**

- Лучано Рамальо: Python. К вершинам мастерства
- Mitchell L. Model. Bioinformatics Programming Using Python // O'Reilly 2010.

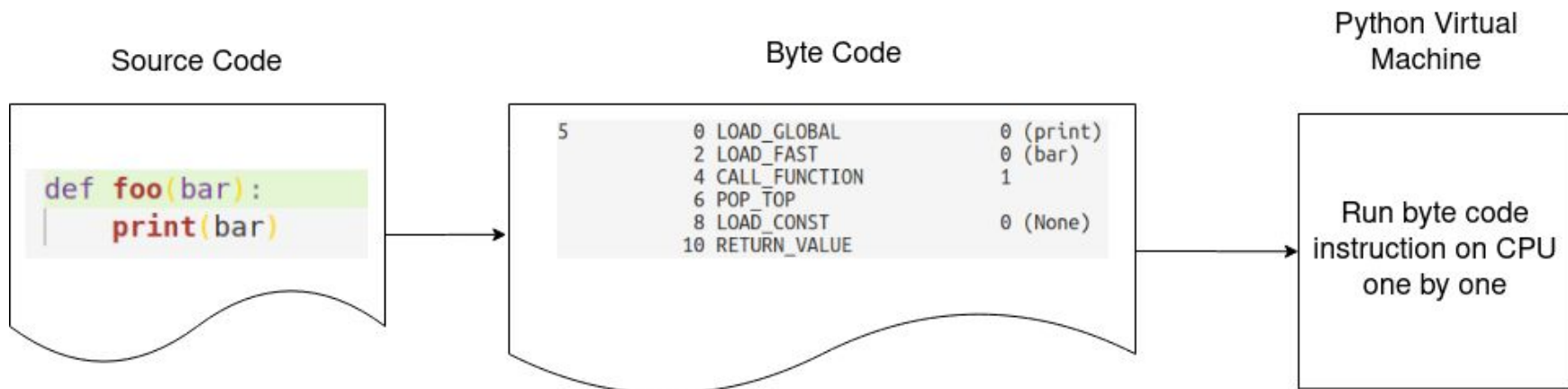
# Версии Python

- Python 2 вышел 2010 году последняя версия 2.7.16 - исправлялись только баги(ошибки) с января 2020 года поддержка прекращена.
- Python 3 появился в 2008, является актуальной версией языка. Текущая стабильная версия 3.8.5 -> в пред релиз 3.9, в разработке 3.10
  - Python 3 не гарантирует совместимости кода с Python 2

# План занятия

- Еще немного об интерпретации кода
- Базовые типы данных
  - неизменяемые
  - изменяемые
- Арифметические операции над числами
- Динамическая типизация
- Переменные
- Работа с вводом/выводом
- Практика

# Еще немного об интерпретации кода





# Python VM

1. Зачем вообще переводить код в byte-code?
2. Какая машина под капотом стандартного Python?
3. А какие бывают?

# Python VM

Зачем вообще переводить код в byte-code?

```
In [2]: def foo():  
        return 1 + 2
```



```
In [3]: dis.dis(foo)  
2  
0 LOAD_CONST          1 (3)  
2 RETURN_VALUE
```

Интерпретатор “умный” и способен производить оптимизирующие операции над исходным кодом, что приводит к увеличению производительности.

# Python VM

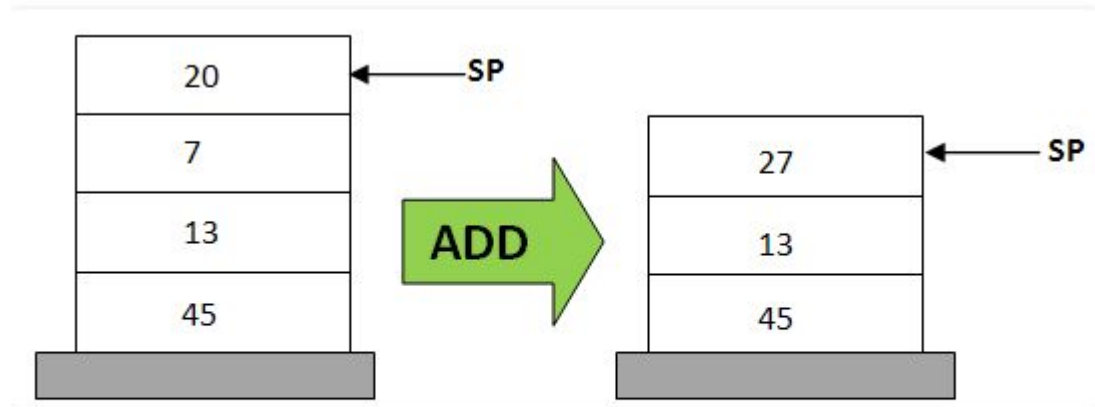
Какая машина под капотом стандартного Python?

# Python VM

А какие бывают?

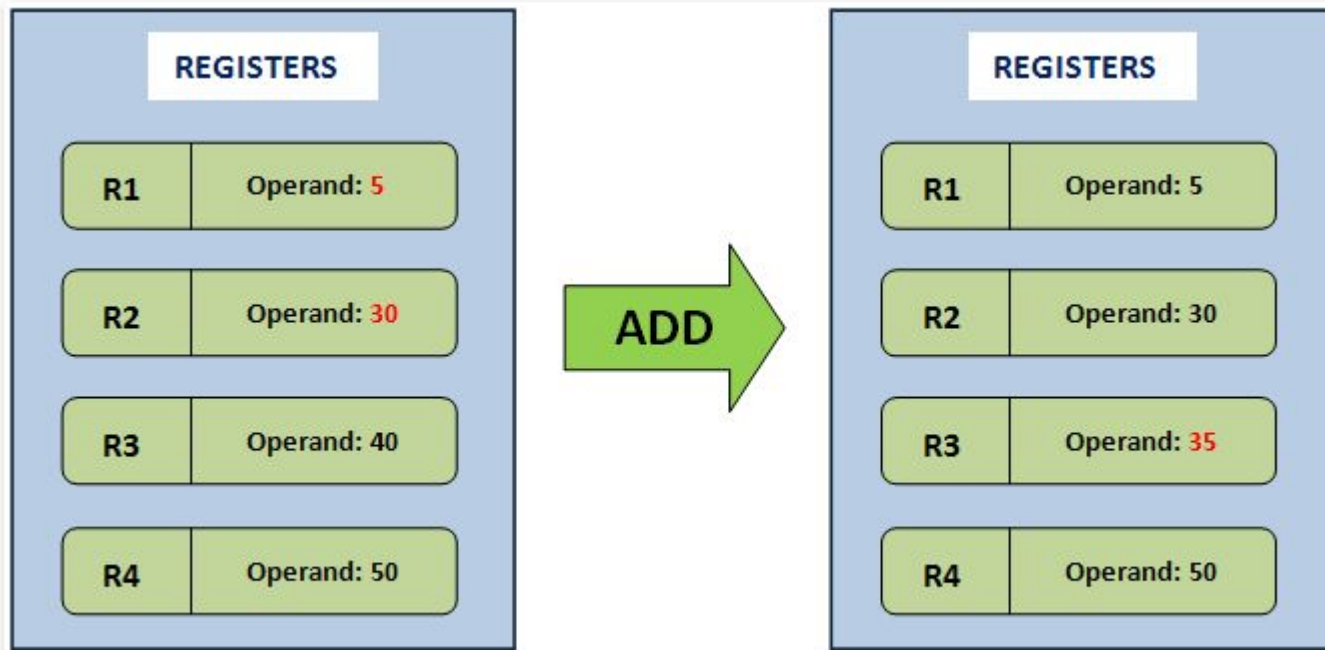
- Стековые. Инструкция и операнды находятся в специализированной структуре данных стеке.
- Регистровые. Такой тип VM основан на регистровой модели процессора, операнды помещаются в специализированные регистры (зафиксированные участки памяти), машина строка за строкой считывает инструкции в которых явно задаются регистры

# Python VM



```
1 | POP    20
2 | POP    7
3 | ADD    20, 7, result
4 | PUSH   result
```

# Python VM



1 | `ADD R1, R2, R3; # складывает содержимое R1 и R2, результат заносит в R3`

# Еще немного об интерпретации кода

## Что-то вроде summary...

Полученный код не зависит от окружения в котором он выполняется, так как для его исполнения нужно только наличие интерпретатора, который сам переведет инструкции в виртуальной машины в бинарные инструкции понятные центральному процессору (ЦПУ). Очевидно, что это влечет к накладным расходам, но в тоже время дает возможность к оптимизации кода...

# Комментарии в коде (как оформлять?)

- Что такое комментарии?
- Поддерживает ли этот функционал Python?
- Как обрабатывает комментарии интерпретатор?
- Как их оформлять?
- Я хочу много комментариев в своем супер приложении, как мне это сделать?



# Примеры комментариев (однострочный)

Однострочные комментарии начинаются с символа '#'. Если интерпретатор встречает подобный символ, то игнорирует все что написано после него

A screenshot of a code editor with a dark background. The text '# this is one line comment it starts with symbol '#' is displayed in a light blue font. The text is on a single line, and the cursor is visible at the end of the line.

```
# this is one line comment it starts with symbol '#'
```

# Примеры комментариев (многострочный)

Многострочные комментарии можно оформлять по разному

1. Как группу однострочных комментариев, написанных один под другим
2. Через три подряд идущих открывающих/закрывающих символа кавычки “  
или ’

```
# Multi line comment  
# line 1  
# line 2
```

```
""" Other multiline comment  
line 1  
line 2  
"""
```

# Комментарии и документация

Документация - подробная или не очень описание функционала вашего кода...

Комментарии тесно связаны с документацией. Документация для модуля помещается в начало файла и обычно оформляется как многострочный комментарий начинающийся `"""`. Для функции документация также помещается перед определением тела функции

```
In [1]: def foo(arg1, arg2):  
...:     """ This function literally do nothing  
...:     """  
...:     :param arg1: description of arg1  
...:     :param arg2: description of arg2  
...:     """  
...:     pass  
...:
```

# Функция help

После оформления комментария документации у пользователя появляется возможность посмотреть документацию с помощью встроенной функции help

```
help(foo)
```



```
Help on function foo in module __main__:  
  
foo(arg1, arg2)  
    This function literally do nothing  
  
    :param arg1: description of arg1  
    :param arg2: description of arg2  
(END)
```

# Базовые типы данных

## Числовые типы данных - int, float complex

- **int** (integer) - целое число, например 10, 1, 0, ....
- **float** - числа с плавающей точкой например 1.2, 3.14, ....
- **complex** - комплексные числа определяются двумя числами вещественной частью и мнимой:

$$\text{num} = a + i*b \rightarrow \text{где } i = \sqrt{-1}$$

## Логические типы

- True/False

## Итераторы

- Специальные объекты позволяющие пройти по последовательности

# Базовые типы данных

## Последовательности

- list -> [1, 3, 4, 5]
- tuple -> (1, 2, 3, 4)
- range -> range(start, end, step) e.g. range(0,10,1)

## Текстовые последовательности

- str -> "Hello Python!"

## Пустое значение

- None

## Словари/множества

- set -> set(1,2,3,4)
- frozenset -> frozenset(1,2,3,4)
- dict -> {"John Doe": "+7903222334", "Albert Einstein": "+142345553", ...}

# Целые числа

int - целые числа как отрицательные так и положительные

Примеры: 1, 2, 0, -1 -2321...

Максимально допустимые значение для целых чисел. В Python 2.7 (`sys.maxint`)

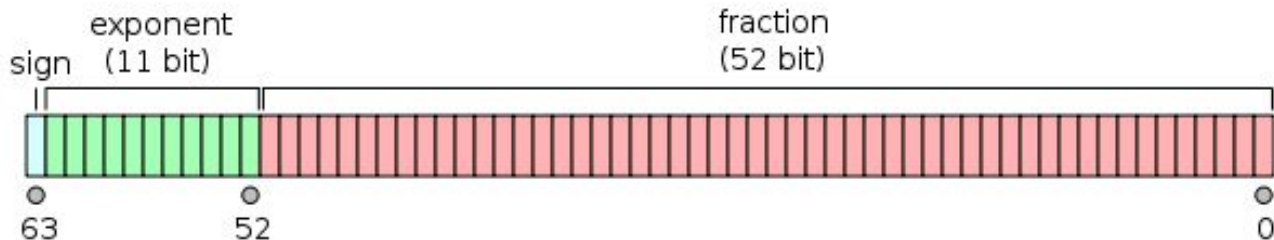
- $2^{31}-1$  для 32 разрядной архитектуры
- $2^{63} - 1$  для 64 разрядной архитектуры

В Python 3.x - таких ограничений нет. Число может быть сколько угодно большим вы ограничены лишь размером оперативной памяти.

# Числа с плавающей точкой

float -> float32 для 32х разрядной системы, float64 - для 64х разрядной (а какая у вас система?)

Для представления мантииссы числа у вас есть 52 битов (это порядка 16 чисел после точки) и 11 битов для представления целой компоненты. Один бит используется для хранения знака числа 0 -> +, 1->- (все вышесказанное действительно только для 64 разрядной архитектуры).





# Примеры

```
In [5]: 3.14
Out[5]: 3.14

In [6]: 5.E-05
Out[6]: 5e-05

In [7]: 5.e-05
Out[7]: 5e-05

In [8]: 5.e+05
Out[8]: 500000.0
```

# Комплексные числа

Для определения комплексного числа нужно воспользоваться встроенным типом `complex` при этом в скобках через запятую передаются значения реальной и мнимой части числа

```
In [10]: complex(1,2)
Out[10]: (1+2j)
```

# Логические типы(bool)

В python логические типы могут иметь ТОЛЬКО два значение, в прочем, в других ЯП это также. Булевы переменные определяют свое значение как одно из двух возможных:

**True** - истина

**False** - ложь

**True, False** - являются встроенными(built-in) константами для Python

# None

В Python None - это константа, которая служит для идентификации того, что переменная которая ссылается на None в данный момент не указывает ни на какой объект в оперативной памяти.

Аналоги для других ЯП: NULL, null, nil, ...

Часто используется, для инициации переменных значение которых еще не вычислено и будет вычислено позже. **Обращение к таким переменным как к объекту содержащему полезную информацию, без проверки его на пустоту приведет к ошибке времени выполнения!**

# Последовательности (Список)

Последовательность элементов, сохраняющую порядок. То есть элементы попадающие в список будут находиться в нем в том порядке, в котором были добавлены. Списки поддерживают операцию индексации [index]

```
In [1]: l = [1,2,3]
```

```
In [2]: l
```

```
Out[2]: [1, 2, 3]
```

```
In [4]: l = list()
```

```
In [5]: l.append(1)
```

```
In [6]: l.append(2)
```

```
In [7]: l.append(3)
```

```
In [8]: l
```

```
Out[8]: [1, 2, 3]
```

# Кортеж (Tuple)

Также последовательность элементов, сохраняющая порядок следования. **НО** эта последовательность. То есть после определения кортежа вы не сможете поменять его состав или увеличить/уменьшить... Хотя операцию индексации этот тип данных также поддерживает.

```
In [10]: t = (1,2,3)
```

```
In [11]: t  
Out[11]: (1, 2, 3)
```

```
In [16]: t[1]  
Out[16]: 2
```

```
In [14]: t = tuple([1,2,3])
```

```
In [15]: t  
Out[15]: (1, 2, 3)
```

```
In [17]: t[1] = 4
```

```
TypeError
```

```
<ipython-input-17-87b0f225887f> in <module>
```

```
----> 1 t[1] = 4
```

```
TypeError: 'tuple' object does not support item assignment
```

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

# Строки

Строка - буквенно знаковая последовательность символов

Кодировка (UTF-8) - наиболее распространенная кодировка, использует для кодирования символов от 1 до 4 байтов (1 байт - 8 бит)

```
In [3]: s_example1 = "String example one"
```

```
In [4]: s_example2 = "String example two"
```

```
In [1]: привет="Hello"
```

```
In [2]: print(привет)
Hello are +@ and -@)
```

# Итераторы

Специализированные объекты позволяющие итерироваться (“пробежаться”) по коллекции элементов, при этом каждый следующий элемент может генерироваться на каждой итерации.

```
In [23]: g = (i for i in range(10))

In [24]: for e in g:
...:     print(e)
...:

0
1
2
3
4
5
6
7
8
9
```



# Словари

Структура хранящая элементы как ключ → значение. НЕ СОХРАНЯЕТ ПОРЯДОК ЭЛЕМЕНТОВ. Обеспечивает быстрой доступ к элементам, быструю вставку/удаление

```
In [27]: d = dict(name='Sergey', surname='khayrulin')  
  
In [28]: d  
Out[28]: {'name': 'Sergey', 'surname': 'khayrulin'}  
  
In [29]: d = {'name': 'Sergey', 'surname': 'Khayrulin'}  
  
In [30]: d  
Out[30]: {'name': 'Sergey', 'surname': 'Khayrulin'}
```

# Множества

Dict-like объект, отличия от словаря заключается в том, что множества хранят только ключи без значений, кроме того сущность множество предоставляет набор основных теоретико-множественных операций - объединения, пересечения, дополнение и так далее...

```
In [32]: s = {1,2,3}
```

```
In [33]: s
```

```
Out[33]: {1, 2, 3}
```

```
In [34]: s = set([1,2,3])
```

```
In [35]: s
```

```
Out[35]: {1, 2, 3}
```

# Базовые типы данных

Class	Description	Immutable?
<b>bool</b>	Boolean value	✓
<b>int</b>	integer (arbitrary magnitude)	✓
<b>float</b>	floating-point number	✓
<b>list</b>	mutable sequence of objects	
<b>tuple</b>	immutable sequence of objects	✓
<b>str</b>	character string	✓
<b>set</b>	unordered set of distinct objects	
<b>frozenset</b>	immutable form of set class	✓
<b>dict</b>	associative mapping (aka dictionary)	

# Арифметические операции над числами

Operation	Result	Notes	Full documentation
<code>x + y</code>	sum of <code>x</code> and <code>y</code>		
<code>x - y</code>	difference of <code>x</code> and <code>y</code>		
<code>x * y</code>	product of <code>x</code> and <code>y</code>		
<code>x / y</code>	quotient of <code>x</code> and <code>y</code>		
<code>x // y</code>	floored quotient of <code>x</code> and <code>y</code>	(1)	
<code>x % y</code>	remainder of <code>x / y</code>	(2)	
<code>-x</code>	<code>x</code> negated		
<code>+x</code>	<code>x</code> unchanged		
<code>abs(x)</code>	absolute value or magnitude of <code>x</code>		<a href="#">abs()</a>
<code>int(x)</code>	<code>x</code> converted to integer	(3)(6)	<a href="#">int()</a>
<code>float(x)</code>	<code>x</code> converted to floating point	(4)(6)	<a href="#">float()</a>
<code>complex(re, im)</code>	a complex number with real part <code>re</code> , imaginary part <code>im</code> . <code>im</code> defaults to zero.	(6)	<a href="#">complex()</a>
<code>c.conjugate()</code>	conjugate of the complex number <code>c</code>		
<code>divmod(x, y)</code>	the pair <code>(x // y, x % y)</code>	(2)	<a href="#">divmod()</a>
<code>pow(x, y)</code>	<code>x</code> to the power <code>y</code>	(5)	<a href="#">pow()</a>
<code>x ** y</code>	<code>x</code> to the power <code>y</code>	(5)	

# Побитовые операции с целыми типами

Битовые операции могут быть выполнены только над целыми числами!

Operation	Result	Notes
<code>x   y</code>	bitwise <i>or</i> of <i>x</i> and <i>y</i>	(4)
<code>x ^ y</code>	bitwise <i>exclusive or</i> of <i>x</i> and <i>y</i>	(4)
<code>x &amp; y</code>	bitwise <i>and</i> of <i>x</i> and <i>y</i>	(4)
<code>x &lt;&lt; n</code>	<i>x</i> shifted left by <i>n</i> bits	(1)(2)
<code>x &gt;&gt; n</code>	<i>x</i> shifted right by <i>n</i> bits	(1)(3)
<code>~x</code>	the bits of <i>x</i> inverted	

<https://docs.python.org/3/library/stdtypes.html#bitwise-operations-on-integer-types>

# Приоритеты операций

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

[https://www.tutorialspoint.com/python/operators\\_precedence\\_example.htm](https://www.tutorialspoint.com/python/operators_precedence_example.htm)

# Приоритеты операций

```
#!/usr/bin/python

a = 20
b = 10
c = 15
d = 5
e = 0

e = (a + b) * c / d      #( 30 * 15 ) / 5
print "Value of (a + b) * c / d is ", e

e = ((a + b) * c) / d    # (30 * 15) / 5
print "Value of ((a + b) * c) / d is ", e

e = (a + b) * (c / d);   # (30) * (15/5)
print "Value of (a + b) * (c / d) is ", e

e = a + (b * c) / d;     # 20 + (150/5)
print "Value of a + (b * c) / d is ", e
```

# Приведение типов

Явное - из название должно быть понятно, что подобное приведение делается явно вами как программистами

Неявное - скорее всего за вас это делает интерпретатор. Вы можете не подозревать о нем, что может приводить к обидным ошибкам, которые еще и искать трудно :(



# Явные операции приведения типа (Python)

- `int` - приведение к целому числу
- `float` - приведение к числу с плавающей точкой
- `str` - приведение к строке
- `bool` - приведение к логическому типу

# int()

```
In [1]: int(3.14)
Out[1]: 3
```

**Обратите внимание**, что при приведении типа к целому числу у числа с плавающей точкой обрезаются дробная часть числа (все что находится за точкой). Это может пригодиться при округлении **НО** эта операция не работает по правилам математического округления числа для этого лучше использовать функцию **round(...)**

**int(num)  $\neq$  round(num)**

```
In [8]: round(2.2)
Out[8]: 2

In [9]: round(2.5)
Out[9]: 2

In [10]: round(2.7)
Out[10]: 3
```

# int() -> число из строки

```
In [2]: int("33")  
Out[2]: 33
```

Также операцию int(...) удобно использовать для приведение строки содержащей число к числу

```
In [7]: int("one")  
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-7-d992879e91e3> in <module>  
----> 1 int("one")  
  
ValueError: invalid literal for int() with base 10: 'one'
```

**НО** так уже не работает Python не настолько умный :(

# float()

Работает примерно также как и `int()` только результатом будет число с плавающей точкой. Причем для целых чисел операция просто вернет число с плавающей точкой со значением дробной части равной нулю. Также можно приводить строки содержащие числа.

```
In [14]: float(3)
Out[14]: 3.0

In [15]: float("3.14")
Out[15]: 3.14
```

# str()

Приведение объекта к строке, в случае если конечно объект может быть приведен к строке

```
In [1]: str(1)
Out[1]: '1'

In [2]: str(True)
Out[2]: 'True'

In [3]: str(0.5)
Out[3]: '0.5'

In [4]: str('test')
Out[4]: 'test'

In [5]: def foo():
...:     pass
...:

In [6]: str(foo)
Out[6]: '<function foo at 0x7f129aebc820>'
```

# bool()

Стоит **ЗНАТЬ**, что любое значение не равное 0 при приведении типа к bool результатом будет **True**, соответственно для пустых значение таких как 0, ""(пустая строка), [] (пустой список) ... результатом приведения к bool будет **False**

```
In [17]: bool(1)
Out[17]: True

In [18]: bool(0)
Out[18]: False

In [19]: bool("")
Out[19]: False

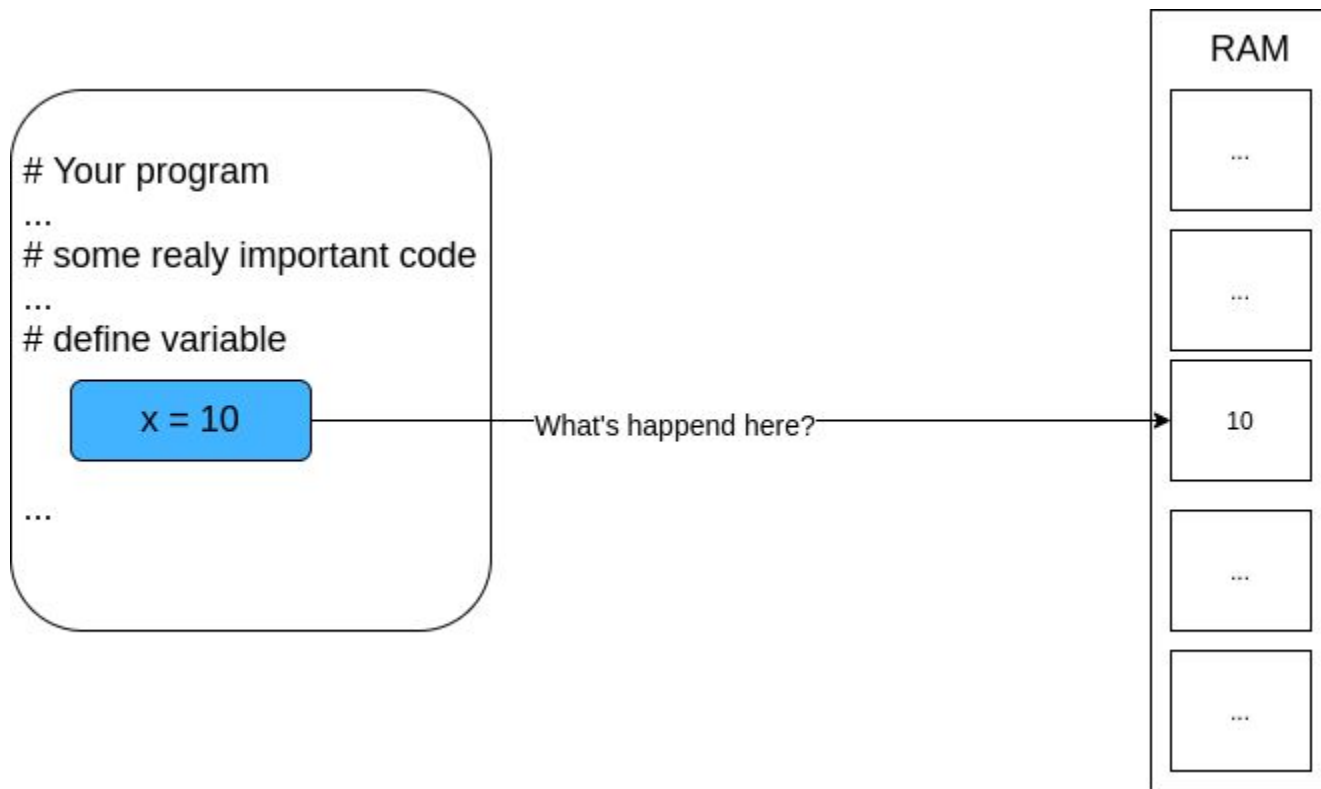
In [20]: bool([])
Out[20]: False
```

# Переменные

Для хранения данных как промежуточных так и постоянных предполагается работать с оперативной памятью компьютера (RAM), ну и с жестким диском конечно, если это необходимо.

Для хранения ссылок на участки памяти во всех языках программирования вводится понятие переменной - **именованная сущность программы предоставляющая доступ к отдельному участку памяти (это вольное определение данное мной не претендую на точность но суть передана).**

# Переменные и RAM





# Переменные - именования

Полный список требований лучше почитать [здесь](#). Здесь кратко:

1. Имя переменной содержит только цифробуквенные символы и символ `_`.  
**НАРУШЕНИЕ ЭТОГО ПРАВИЛА ВЛЕЧЕТ ОШИБКУ ИНТЕРПРЕТАЦИИ!**
2. Имя переменной **НЕ МОЖЕТ** начинаться с цифры. **ТОЖЕ ЧРЕВАТО ОШИБКАМИ!**
3. Длина имени  $> 1$  (это не строгое требование в некоторых случаях вполне оправдано в основном историческими причинами)
4. Максимальное имя тоже ограничено
5. Имя переменной должно быть лаконичным и отражать смысл данных, которые вы предполагаете получать при работе с этой переменной

# Переменные - именования (стиль)

Вообще в Python принято пользоваться двумя стилями для именования сущностей программы

CamelCase - для именования имен классов и комплексных сущностей (для разделения смысловых частей названия используется заглавная буква)

пример -> MyClass, HTTPServer,...

underscore - для именования всего остального (для разделения смысловых частей названия используется символ '\_')

пример -> my\_function(...), my\_variables, ...

# Переменные - именования

```
# Плохое имя переменной, но интерпретатор молчаливо пережует его  
f, aa, d1, r55, m32, this_is_very_long_name_for_variables...
```

```
# Примеры имен, которые вызовут ошибку интерпретации  
1, 1f, aaa@aaa ...
```

```
# Хорошее имя переменной  
some_text, db_connection, received_data, point, user_list,...
```

# Переменные инициализация (синтаксис)

Переменные инициализируются через оператор присваивания =

Пример: `variable_name = variable_value`

# Объявление переменных в Python

```
# объявление переменной
```

```
x = 10
```

```
# Переопределение значения переменной,
```

```
# то есть после этой операции переменная
```

```
# будет указывать на другой объект в памяти.
```

```
some_text = 'Hello World'
```

```
pi = 3.14
```

```
# Изменение переменной. После этой операции переменная
```

```
# x указывает на новое значение 4.14
```

```
x = x + 1
```

# Переменные

- После объявления переменной у вас появляется возможность обращаться к текущему значению переменной в вашей программе
- Переменные можно использовать для хранения данных или инициализации других переменных

```
radius = 1
```

```
pi = 3.14
```

```
area = pi * radius * radius # вычисление площади круга
```

# Операция удаления del

После объявления переменной, как вы уже поняли, у вас появляется возможность обращаться к значению переменной через ее имя, что делать если мне нужно удалить переменную из контекста программы?

```
In [1]: x = 1
```

```
In [2]: print(x)
```

```
1
```

```
In [3]: del x
```

```
In [4]: print(x)
```

```
-----  
NameError
```

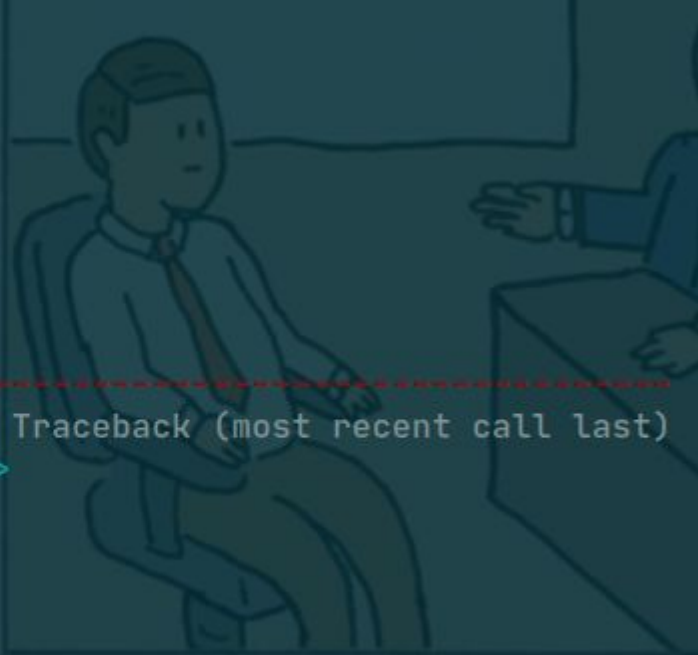
```
<ipython-input-4-fc17d851ef81> in <module>
```

```
----> 1 print(x)
```

```
NameError: name 'x' is not defined
```

```
In [5]:
```

Traceback (most recent call last)





# Динамическая типизация

***Strong** typing means that the type of a value doesn't change in unexpected ways. A string containing only digits doesn't magically become a number, as may happen in Perl. Every change of type requires an explicit conversion.*

***Dynamic** typing means that runtime objects (values) have a type, as opposed to static typing where variables have a type.*

Python - **динамически** типизированный язык.

При динамической типизации переменная не знает тип значение на которое в данный момент эта переменная указывает (**ЭТО ВАЖНО**) - при этом само значение хранит эту информацию.

# Динамическая типизация

Не стоит считать, что все языки поддерживают динамическую типизацию, как и по типу выполнения кода языки, также можно объединять по типу типизации.

## Dynamic typing Language

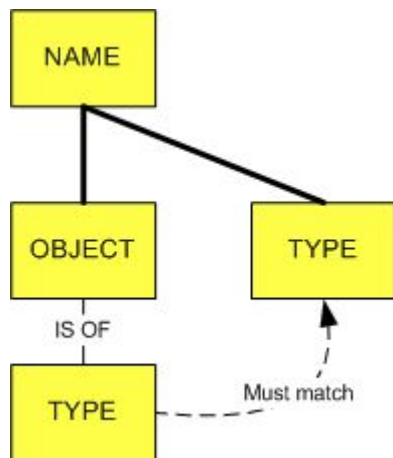
- ...
- Python
- JS
- ...

## Static typing Language

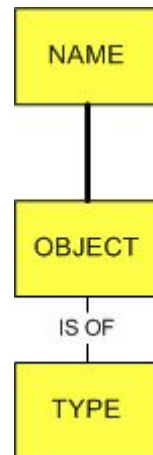
- ...
- C++
- C
- ...

# Динамическая типизация

Статическая типизация



Динамическая типизация



# Динамическая типизация (+/-)

+	-
Удобство объявления	Неочевидное поведение программы
Красиво и компактно	Memory Overhead

# Динамическая типизация in action (Python)

Python (динамическая типизация все что написано ниже верно и не вызовет ошибок)

```
x = 10
```

```
x = 'Hello World'
```

```
x = 3.14
```

C++ (статическая типизация)

```
int intVar = 10;
```

```
string stringVar = "Hello world"
```

```
float floatVar = 3.14
```

```
// а вот так в C++ сделать не получится компилятор станет ругаться
```

```
floatVar = "Hello world"
```

# Переменные(базовые операции)

*# Для сокращения записи, если выражение*

*# подразумевает изменение той же переменной,*

*# то разумно использовать следующие варианты записи*

`x += 1` *# тоже самое что и  $x = x + 1$*

`x -= 1` *# тоже самое что и  $x = x - 1$*

`x *= 1` *# тоже самой что и  $x = x * 1$*

`x /= 2` *# тоже самое что и  $x = x / 2$*

`x **= 2` *# тоже самое что и  $x = x ** 2$*

*# Присвоение*

`x = 1`

`y = 2`

`x = y` *# теперь переменная x равна 2*

`z = x + y` *# теперь переменная z равна 4*

# Переменные (базовые операции)

Узнать тип переменной можно с помощью функции `type(...)`

```
x = 10
```

```
# результат выполнения этой функции будет
```

```
# тип объекта переданного в аргументе
```

```
# в нашем случае int
```

```
type(x)
```

Узнать месторасположение объекта в памяти можно с помощью `id(...)`

```
# результат зависит от платформы
```

```
# НО точно будет являться целым числом
```

```
id(x)
```

# Работа с вводом/выводом

Взаимодействие с пользователем может настраиваться через стандартные потоки ввода и вывода `stdin/stdout`.

Для получения входной информации от пользователя можно воспользоваться вызовом функции [`input\(...\)`](#)

Для вывода информации из программы можно воспользоваться функцией [`print\(...\)`](#)

Важно: функция `input` блокирует выполнение скрипта и ждет ввода пользователя, для того чтобы продолжить работу. Кроме того сама функция возвращает введенную строку, **НЕ ЗАБЫВАЙТЕ КОНВЕРТИРОВАТЬ ТИПЫ**



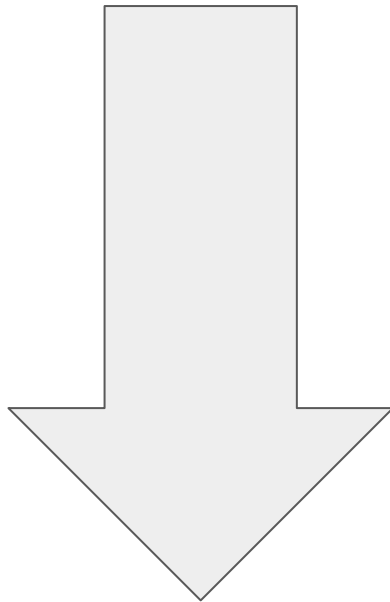
# Работа с вводом/выводом

```
In [21]: my_name = input("Put your name here: ")  
Put your name here: Sergey
```

```
In [22]: print(my_name)  
Sergey
```

```
In [23]: █
```

# Практическая Часть



1. Вычислить степень двойки разными способами (используйте встроенные функции пакета `math` `pow` и `**` )
2. Вычислите степень 0.5 из отрицательного числа

1. Приведите целое число к числу с плавающей точкой (команда `float(...)`)
2. Приведите число с плавающей точкой к целому (команда `int(...)`)
3. Вычислите дробную часть числа
4. Вычислите остаток от деления (%)
5. Вычислите целое частное (//)
6. Как проверить, что число четное?

1. Создайте целочисленную переменную и присвойте ей некоторое значение (например 123)
2. Создайте две переменные и присвойте значение первой второй ( операция равенства/присвоение = )
3. Создайте переменную как результат некоторой операции над числами (например  $10 + 1$ )
4. Создайте переменную как результат некоторой операции над другими переменными (например  $\text{var1} + \text{var2}$ )
5. Создайте переменную и удалите ее из контекста (операция `del val`)
6. Какой тип будет у переменной, которая является результатом сложения целого числа и числа с плавающей точкой

1. Выведете строку приветствия с вашим именем (функция `print(...)`)
2. Разработайте приложение принимающее на вход два числа и выводящее сумму этих чисел

ДЗ: Реализовать программу, которая спрашивает у пользователя: имя, фамилию, год рождения. После ввода всех данных программа должна выводить строку следующего вида:

```
"Hello {Name} {Surname} your age is {year} year"
```