Разработка веб сервисов для научных и практических задач. Цели.

Изучение современных средств разработки ПО (языка программирования) для дальнейшего использования при изучении других дисциплин и реализации своих академических исследований.

Изучение современных технологий для решения научных и прикладных задач.

Изучение технологий для предоставления сервисов пользователям, технологий высоконагруженных систем.

Изучение средств разработки. Python. Почему Python.

- Why not?
  Язык подходит для быстрой реализации какого-либо проекта, осуществления какой-либо идеи, оценки ее работоспособности. У языка высокая динамичность, возможность изменять любые объекты и возвращать любые объекты.
- Python лаконичный.
  Что дает возможность получать короткий код, который еще и легко читаемый. Нет множества фигурных скобочек и точек с запятой или других способов выделения блоков загромождающих код.

Популярность

PYPL PopularitY of Programming Language

https://pypl.github.io/PYPL.html

TIOBE (TIOBE programming community index) — индекс, оценивающий популярность языков программирования, на основе подсчёта результатов поисковых запросов, содержащих название языка (запрос вида +"<language> programming").

https://www.tiobe.com/tiobe-index/

Множество реализованных библиотек для решения научных задач, машинного обучения, нейронных сетей, обработки данных, больших данных, библиотеки и фреймворки реализации сетевых сервисов, распределенных вычислений и тензорных вычислений. Возможности обращения к библиоте-кам Си, различным СУБД. Есть свой JIT-компилятор PyPy. На Python есть множество библиотек для решения научных задач

```
Keras, Tensorflow, Theano, Skikit-Learn, pandas, matplotlib.
```

Это интерпретируемый язык, хотя для него существуют вариант jit-компилятора pypy, а также возможность транслиро-вания в исполнимый модуль, так же есть трансляторы в CIL и Java байт код.

Больше можно посмотреть на

https://proglib.io/p/50-python-projects/

Например

Python Google Images Download

Утилита командной строки, которая позволяет искать изображения в Google Images по ключевым словам или фразам и загружать их на компьютер. Скрипт также можно запускать из любого python-файла.

Библиотека Mask R-CNN предназначена для обнаружения объектов и сегментации изображений.

Detectron – так же для сегментации.

Развитие языка Python происходит согласно чётко регламентированно-му процессу создания, обсуждения, отбора и реализации документов PEP.

PEP - Python Enhancement Proposal - это предложения по развитию питона https://www.python.org/dev/peps/. Процесс PEP является основным механиз-мом для предложения новых возможностей и для документирования проект-ных решений, которые прошли в Python.
Самым известным PEP является PEP8 - это свод рекомендаций по оформлению кода (https://www.python.org/dev/peps/pep-0008/).

Java style

```
// A Java program for splitting a string
// using split()

import java.io.*;
public class Test
{
    public static void main(String args[])
    {
        String Str = new String("Geeks-for-Geeks");

        // Split above string in at-most two strings
        for (String val: Str.split("-",2))
                System.out.println(val);

        System.out.println("");

        // Splits Str into all possible tokens
        for (String val: Str.split("-"))
                System.out.println(val);
    }
}
```

Python 3.x style

```
line = "Geek \nGeek2 \nGeek3"
print(line.split())
print(line.split('\n'))
```

- Python легко читаемый

Пример кода

```
def magic(top):
    acc = []
    for entry in os.scandir(top):
        if entry.is_file() and entry.name.endwith(".py"):
            acc.append(entry.path)
    return acc
```

CPython (PVM) – основная оригинальная реализация Python
(*.py->*.pyc->выполнение PVM)
JPython – трансляция(компиляция) в байт код Java, исполняемый JVM.
Поддержка Java компонент, классов.
Пример того же – Kotlin (Другой язык).
IronPytnon – транслятор для .Net или Mono.
Соответственно байт код CIL исполняется CLR.
Psyco – транслирует часть байт кода в машинный,
что ускоряет исполнение порой в 4-100 раз.
Shedskin C++ - транслирует Python на язык C++,
потом можно скомпилировать.
Py2exe, freeze, PyInstaller – создает
исполнимый exe модуль или исполнимый модуль для Linux,
который тащит с собой PVM.
PyPy – Jit-компилятор python, сразу с языка в машинный на лету,
не поддерживает некоторые библиотеки или частично,
например numpy не все, поддерживает Flask, Django.
Намного быстрее Cpython при сравнении напрямую
для численных задач без использования numpy
(реализовать самостоятельно если работу с массивами и т.д.)
В 2017 отказался от бэкендов CIL, JVM, JavaScript.

PyPy-stm (Software transaction memory)

Позволяет решить проблему стандартного Python интерпретатора глобальной блокировки интерпретатора (GIL, global interpreter lock), не позволяющей обеспечить параллельное выполнение нескольких нитей кода на языке Python на разных ядрах.
Используется программная транзакционная память (оптимистичная).
Оптимистична в том плане, что поток завершает изменения памяти независимо от действий других потоков. При записи не проверяется влияет ли это на другие потоки (нет блокировки как обычно), проверка осуществляется считывающим устройством при завершении транзакции. Если транзакция не может быть выполнена из-за конфликтов записей, она прерывается и выполняется заново.

Python интерпретатор, транслирующий в байт-код и затем интерпретирующий этот байт код. Используется код для стековых виртуальных машин, он медленнее чем для регистровых на десятки процентов и подвержен уязвимостям, но благодаря переносимости и быстрой разработки все равно python получил широкое распространение, кроме того есть варианты с jit-компиляторами. В современных Jit-компиляторах используется регистровые виртуальные машины.

```
print("Hello world")
# получение байт кода
import dis
dis.dis('print("Hello world")')
```

Особенности

Все является объектом

Все является выражением (даже, например определение функции это выражение)

Поддерживается ООП

Поддерживается функциональная парадигма

Возможности

Интроспекция.

Высокая динамичность. Можно получить и изменить данные о программном объекте во время исполнения.

Необходимые для интроспекции данные хранятся в специальных атрибутах. Так, например, получить все пользовательские атрибуты большинства объектов можно из специального атрибута - словаря (или из другого объекта, предоставляющего интерфейс `dict`) `__dict__`

Примеры

```
>>> class x(object):pass
....
>>> f = x()
>>> f.attr = 12
>>> print(f.__dict__) ##### 1
{'attr': 12}
>>> print(x.__dict__) # т.к. классы тоже являются экземплярами объекта type,
                      # то они поддерживают этот тип интроспекции
{'__dict__': <attribute '__dict__' of 'x' objects>, '__module__' ........}
```

Получение кода и байткода функции

```
def f(x):pass
print(f.__code__)
print(f.__code__.co_code)
```

Получение интроспекции на функцию. Модуль inspect.

```
import inspect
def function(x, y=10)
    return x**2
val = inspect.getfullargspec(function)
val1 = inspect.signature(function)
print(f'{val}')
print(f'signature {val1}')
```

Результат

```
FullArgSpec(args=['x', 'y'],
varargs=None, varkw=None,
defaults=(10,), kwonlyargs=[], kwonlydefaults=None,
annotation={})
signature (x, y=10)
# дефольное значение начинается с первого слева,
# с момента начала первого по умолчанию последующее отстутствие значения по умолчанию даст ошибку.
```

```
import inspect
def fun(x, y=10, *unnamed, par1=4, par2=3, **named)->int:
    for val in unnamed:
        print(f'unnamed value {val}')
    for key, val in named.items():
        print(f'named value {key} = {val}')

    return x+y
print(inspect.getfullargspec(fun))
print(fun(2, 2, 'ul', 1, 2, par1=2, par2=2, name1=1, name2='s'))
```

Выводятся неименованные неуказанные *args (*unnamed) и именованные неуказанные аргументы **kwargs (*named) позволяющие передать дополнительные произвольные значения параметров.

В принципе * и ** выступают как разыменование списка или словаря соответственно.
Внутри функции можно обратиться к нескольким переданным значениям, а при вызове можно передать произвольное количество параметров.
Здесь передаются дополнительно неименованные значения ul, 1, 2 и именованные name1, name2, между ними находятся именованные указанные переменные функции par1, par2.

Результат

```
FullArgSpec(args=['x', 'y'],
varargs='unnamed', varkw='named',
defaults=(10,), kwonlyargs=['par1', 'par2'],
kwonlydefaults={'par1': 4, 'par2': 3},
annotations={'return':<class 'int'>})
unnamed value u1
unnamed value 1
unnamed value 2
named value name1 = 1
named value name2 = s
4
```

Аннотации типов

Необходима для анализа кода, контроля типов, несмотря на динамическую типизацию, например с помощью `mypy`.

```
python -m mypy test_type.py
```

Три способа аннотации переменых +1 для функций

1. `var = value # type: annotation`
2. `var: annotation; var = value`
3. `var: annotation = value`

Примеры

```
>>> name "John" # type: str  #### Автоматическая аннотация
>>> name: str; name = "John" #### Вручную в 2 команды
>>> name: str = "John"       #### То же самое в 1 команду
```

+1 для функции

4. `def func()->int`

Проверка типов

Различают типы `int`, `float`, `bool`, `str`, `complex`...

```
>>> type(int)
<class 'type'>

>>> type(2+2)
<class 'int'>

>>> d = (1+2j)
>>> type(d)
<class 'complex(2.0)'>
```

Обзор типов.

None.

Аналог NULL, но является полноценным объектом

```
>>> res = print(None) # Любая функция принимая None вернет None
None

>>> res == None       # Не делайте так
True

>>> res is None       # Лучше так
True

>>> id(None)           # В CPython - адрес
140503861072
```

Bool.

Логический тип, принимает значения `True` и `False`

На нем определены

- операции алгебры логики `not`, `and`, `or` ..
- операции сравнения `>`, `<`, `==`, `>=`, `<=` ..

```
>>> to_be = False
>>> to_be or not to_be    # Получается использовать слова вместо значков
True
```

```
>>> x = 1
>>> y = 2
>>> x**2 + y**2 < 5 == True    # True синглтон, не делайте так
False

>>> x**2 + y**2 < 5 is True    # И так тоже не делайте
False

>>> x**2 + y**2 < 5    # Гуд
False
```

```
>>> False and print('also')    # выражение false,
                              # вычислять вторую часть не надо
False
# здесь уже требуется вычисление второй части
>>> res = True and print('also')
also

>>> print(res)
None

>>> False or 92    # Чтобы узнать результат нужно вычислить второе выражение
92
```

Операции

```
>>> flag = True
>>> flag
True

>>> flag + True
2

>>> flag + False
1

>>> flag and False
False
```

Assert применяется для отлавливания багов, вызывая исключение, если какое-то условие было нарушено.

```
def apply_discount(product, discount):
    price = int(product['цена'] * (1.0 - discount))
    assert 0 <= price <= product['цена']
    return price
```

```
>>> apply_discount(shoes, 0.25)    # Все верно, скидка 25%
11175
>>> apply_discount(shoes, 2.0)     # Скидка в 200% вызовет ошибку исполнения
Traceback (most recent call last):
    File "<input>", line 1, in <module>
        apply_discount(prod, 2.0)
    File "<input>", line 4, in apply_discount
        assert 0 <= price <= product['price']
AssertionError
```

Truthy/Falsy

```
>>> bool(True)
True

>>> bool(0)
False

>>> bool(1)
True

>>> bool([])
False

>>> bool([0])
True
```

```
if len(xs) == 0:     # Плохо
    pass

if xs:
    pass

if not xs:
    pass
```

Numbers. Операции

- возведение в степень `2**10` (`== 1024`),
- деление нацело `15//2` (`== 7`),
- вещественное деление `15/2` (`== 7.5`),
- Взятие остатка `15%2` (`== 1`).

```
>>> -3 // 2     # Как в алгебре!
-2
>>> -1 % 3      # C/Java/Rust скажут -1
2
# Определение остатка от деления
x = a//b
a = b*x+r
r  >=0  and < b
-9//2 = -5
-9%2 = 1
-5*2+1 = -9
```

Дополнительный синтаксис.

```
>>> x = 10
>>> 0 <= x and x < 100
True

>>> 0 <= x < 100
True
```

List.

Упорядоченный список элементов `[]`, `[1, 2, 'd']`

- взятие значения по индексу `some_list[0]` (+slicec)
- присвоение по индексу `some_list[0] = 'Hello, world!'`
- умножение дублирует `[1, 2] * 3 = [1, 2, 1, 2, 1, 2]`
- взятие длины `len()`

Опрератор

- наличие данного элемента `item in list`

```
>>> 1 in [1, 2, 3]
True
```

Методы

- добавить в конец `.append()`
- удалить элемент (удаляет последний по умолчанию) `.pop()`
- создать новый список и объединить с другим `.insert(oldL, newL)`

```
>>> xs = [[0] * 3] * 3    # Не делайте так, продублируются ссылки на изначальный список
>>> xs
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

>>> xs[0][0] = 1    # Нежелательное изменение в 2 других местах
>>> xs
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
```

```
>>> xs += [1]     # Не надо так делать
>>> xs
[92, 2, 3, 1]
```

slices

С их помощью можно производить выборку элементов списков

- срезает по такому правилу `item[START:STOP:STEP]`

Параметры могут опущены (хоть все), могут быть отрицательными

```
>>> xs
[0, 1, 2, 3, 4]
>>> xs[-1]      # xs[len(xs) - 1]
4
>>> xs[2:4]     # Start Stop
[2, 3]
>>> xs[:-2]     # Stop
[0, 1, 2]
>>> xs[::2]     # Step
[0, 2, 4]
>>> xs[:]       # Defaults
[0, 1, 2, 3, 4]
```

```
>>> x = range(15,1,-1)
>>> list(x)
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2]
>>> x = range(15,-1,-1)
>>> list(x)
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> y[0:-1:3]
[15, 12, 9, 6, 3]
>>> y[0:-1:3] = [3]*5
>>> y
[3, 14, 13, 3, 11, 10, 3, 8, 7, 3, 5, 4, 3, 2, 1, 0]
5 – нужно правильно рассчитать, иначе вызывается исключение)
```

Важная особенность

```
>>> new_list = old_list    # Ошибка! скопируется ссылка на old_list
>>> id(new_list) == id(old_list)
True
>>> new_list = old_list[:] # Это поэлементно скопирует список
>>> id(new_list) == id(old_list)
False
```

Str.

Класс строки.

Операторы

- конкатенация строк `str1 + str2`
- умножение строк по аналогии со списком `str * 3`
- наличие подстроки в строке 'substr in str'

```
>>> "Hello " + "world!"    # Конкатенация
'Hello world!'

>>> "multi " * 3           # Умножение
'multi multi multi '

>>> "world" in "Hello, world!"
True
```

Методы

- Разбиение по стокам `.splitlines()`
- Разбиение по словам `.split()`
- Убрать пробелы `.strip()`
- Соединить список строк через разделитель `.join()`
- Явное приведение к типу `.str()`
- Длина `.len()`

```
>>> "Hello \nworld".splitlines()    # По строкам
['hello ', 'world']
>>> "a, b, c".split()               # По словам
['a', 'b', 'c']
>>> ", ".join(["a", "b", "c"])      # Соединить через разделитель
'a, b, c'
>>> str(123)                        # Приведение типа
'123'
>>> len("some_string")              # Не считая символа конца строки
11
```

```
#Ввод строк
>>> str=input("Input any value: ")
Input any value: 45
>>> val=int(input("Input any value: "))
Input any value: 45
>>> str
'45'
>>> val
45
```

Форматирование строк

`"%some_format" % (some_var)`

- `%` - оператор форматирования

```
>>> print("%d :: %s :: %d" % (flag, str, val))    # Выглядит почти по аналогии с Си
1 :: some_str :: 123
```

- можно модифицировать оператор формата

```
>>> print("%8s :: %s :: %05d" % (flag, str, val))
True     :: some_str :: 00123
```

Дополнительный вариант форматирования строк - метод формат

`"{Номер значения 1} {Номер значения 2} ...".format(значения по порядку)`

```
>>> s = "{} {} {}?".format("what", "is", "it")
# Автоматический порядок
>>> s
'what is it?'

>>> s = "{2} {1} {0}?".format("what", "is", "it")
# Явно указываю порядок
>>> s
'it is what?'

>>> s = "{:>7} {} {}?".format("what", "is", "it")
# Формат фнутри
'   what is it?'

>>> s = "{2:>7} {1} {0}?".format("what", "is", "it")
# Формат внутри + явное указание порядка
'     it is what?'
```

Пример рисования звездочек

```
>>> print("{0:^15}\n{1:^15}\n{2:^15}".format("*","*"*3,"*"*5))
       *
      ***
     *****
```

^,>,< выравнивание по центру, по правому и левому краю

f- строки Python 3.6+

join — склеивание списка в строку

split разбиение строки на элементы списка

```
>>> print("".join([f"{'*'*(i*2+1):^15}\n" for i in range(5)]))
       *
      ***
     *****
    *******
   *********
>>>x = 9
>>> print("".join([f"{'*'*(i*2+1):^{x*2-1}}\n" for i in range(x)]))
>>> treg = lambda x,y: "" if x==-1 else treg(x-1,y)+f"{'*'*(x*2+1):^{y}}\n"
print(treg(7,7*2+1))
```

```
exec('treg = lambda x,y,c: "" if x==-1 else
treg(x-1,y,c)+f"{(c)*(x*2+1):^{y}}\\n"\n
print(treg(6,6*2+1,"x"))')

#Использование Y — комбинатора лямбда исчисления
print((lambda a:lambda v:a(a,v))(lambda s,x:""
if x==0 else s(s,x-1)+f"{'*'*(2*x-1):^15}\n")(8))
```

Во многом работа со строками похожа на работу со списками, но в отличии от списков, строки не изменяемы.

```
>>> s = "Hallo!"
>>> s[1] = "e"
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Однако, это опять же можно обойти, используя слайсеры.

```
>>> s = "Hallo!"
>>> s = s[0] + 'e' + s[2:]
>>> s
'Hello!'
```

tuple

Класс кортежа. API, как у списка, но менять нельзя.

- пустой кортеж `()`,
- кортеж одного элемента `1` или `(1, )`,
- скобки опциональны `date = "September", 2018`

```
>>> date = ("September", 2018)
>>> len(date)
2
>>> date[1] = 2019
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does bot support item assignment
>>> xs = ([], [])
>>> xs[0].extend([1, 2, 3])
# Это работает, т.к. мы не меняем саму ссылку на объект
>>> xs
([1, 2, 3], [])
```

```
# Пример возвращения кортежа функцией.
# Реализация векторной функции векторного аргументта.
>>> def div_mod(x, y):
...     return x // y, x % y

# присваивание кортежем d = x // y, m
>>> d, m = div_mod(10, 3)       = x % y

# Ассерты для такой функции
>>> assert (d, m) == (3, 1)
```

set

Класс множества. По определению не может быть повторений и порядок элементов не важен.
`xs = {1, 2, 3, ...}`.

Пустое множество не имеет собственного особого литерала и обозначается `set()`.

Операторы

- наличие элемента во множестве `in`,
- перегруженные операторы для метододов `|`, `&`, `^`

```
>>> xs = {1, 2, 3}
>>> 1 in xs
True

>>> 92 in xs
False
```

Методы

- добавить элемент `add()`,
- объединение множеств `|` или `.union()`,
- пересечение множеств `&` или `.intersection()`,
- исключающее или `^` или `.symmetric_difference`,
- исключение элемента из множества `.discard()`.

```
>>> xs = {1, 2, 3}
>>> xs.add(1)     # Уже есть
>>> xs.add(92)
>>> xs
{1, 2, 3, 92}
```

```
>>> {1, 2, 3}.union({3, 4, 5})    # Объединение
{1, 2, 3, 4, 5}

>>> {1, 2, 3} | {3, 4, 5}
{1, 2, 3, 4, 5}

>>> {1, 2, 3}.intersection({3, 4, 5})    # Пересечение
{3}

>>> {1, 2, 3} & {3, 4, 5}
{3}

>>> {1, 2, 3}.symmetric_difference({3, 4, 5})    # XOR
{1, 2, 4, 5}

>>> {1, 2, 3} ^ {3, 4, 5}
{1, 2, 4, 5}

>>> xs.discard(2)    # В действительности операция редкая
>>> xs
{1, 3}
```

Нельзя добавить список во множество

```
>>> xs = set()
>>> xs.add([])
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

dict

Класс словаря.

`dict = {"key0": value_a, "key1": value_b}`

- "длина" словаря - число входящих элементов `len()`,
- взятие значения по ключу `dict[key]`,
- присвоение по ключу `dict[key] = 14`.

```
>>> date = {"year": 2018, "month": "September"}

>>> len(date)    # Длина словаря
2

>>> date["year"]    # Взятие по индексу
2018

>>> date["day"] = 14    # Присвоение по индексу
>>> date["year"]
14
```

Операторы

- наличие ключа в словаре `in`.

```
>>> 'day' in date.keys()    # Плохо!
True

>>> 'day' in date
True
```

Методы

- писваивание значения по умолчанию `.get(key, value)`,
- удалить запись из словаря и вернуть ее значение `.pop(key)`,
- вывести все ключи `.keys()`,
- вывести все значения `.values()`,
- вывести все и сразу `.items()`.

```
>>> date.get("day", 14)    # Значение по умолчанию
14

>>> date.pop("year")
2018

>>> date.keys()
dict_keys(['month', 'day'])

>>> date.values()
dict_values(['September', 14])

>>> date.items()
dict_items([('month', 'September'), ('day', 14)])
```

Упорядочение элементов

Порядок элементов остается неизменнным и зависит только от времени присвоения ключу его первого значения.

```
>>> dict_ = {}
>>> dict_["a"] = 1
>>> dict_["b"] = 2
>>> dict_["c"] = 3
>>> list(dict_.keys())
['a', 'b', 'c']

>>> dict_["a"] = 3
>>> dict_["b"] = 2
>>> dict_["c"] = 1
>>> list(dict_.keys())    # Порядок гарантируется
['a', 'b', 'c']

>>> list(dict_.values())    # Порядок гарантируется
[3, 2, 1]
```

Конструкции

if

```
if CONDIT1:
    expression1()
elif CONDIT2:
    expression2()
else:
    expression3()
```

ternary if

return **this()** `if` ***CONDIT1*** `else` **that()**

```
value = x if x < y else y    # int value = (x < y)? x : y; ## На Си
```

Не следует переносить условия на новую строку

```
# Плохо :(((
if x[0] < 100 and x[1] > 100 and (is_full_moon() or not it_thursday()) and user.is_admin
    pass

# Это не работает! >:(((((
if x[0] < 100 and x[1] > 100
    and (is_full_moon() or not is_thursday())
    and user.is_admin:
    pass

# Тоже плохо :(((
if x[0] < 100 and x[1] > 100 \
    and (is_full_moon() or not is_thursday()) \
    and user.is_admin:
    pass

# Со скобочками чуть лучше :((
if (x[0] < 100 and x[1] > 100
    and (is_full_moon() or not is_thursday())
    and user.is_admin):
    pass
```

while

Цикл по условию.

```
while CONDIT:
    expression()
```

for

`for` **var** `in` **list** `:` *expression*

```
for x in range(10):     # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    print(x)

for ch in "Hello world":     # H, e, l, l,o, ' ', w, o, r, l, d
    print(ch)
```

Забегая вперед

`range(START, STOP, STEP)` - генератор

break

Преждевременный принудительный выход из цикла

```
target = some_target
for item in items:
    if item == target:
        print("Found!", item)
        break
else:
    print("Not found")    # Особый вариант, сработает только если не было принудительного выхода из цикла
```

continue

Принудительное завершение итерации цикла

```
target = some_target
res = []
for item in items:
    if item != target:
        continue
    res.append(item )
```