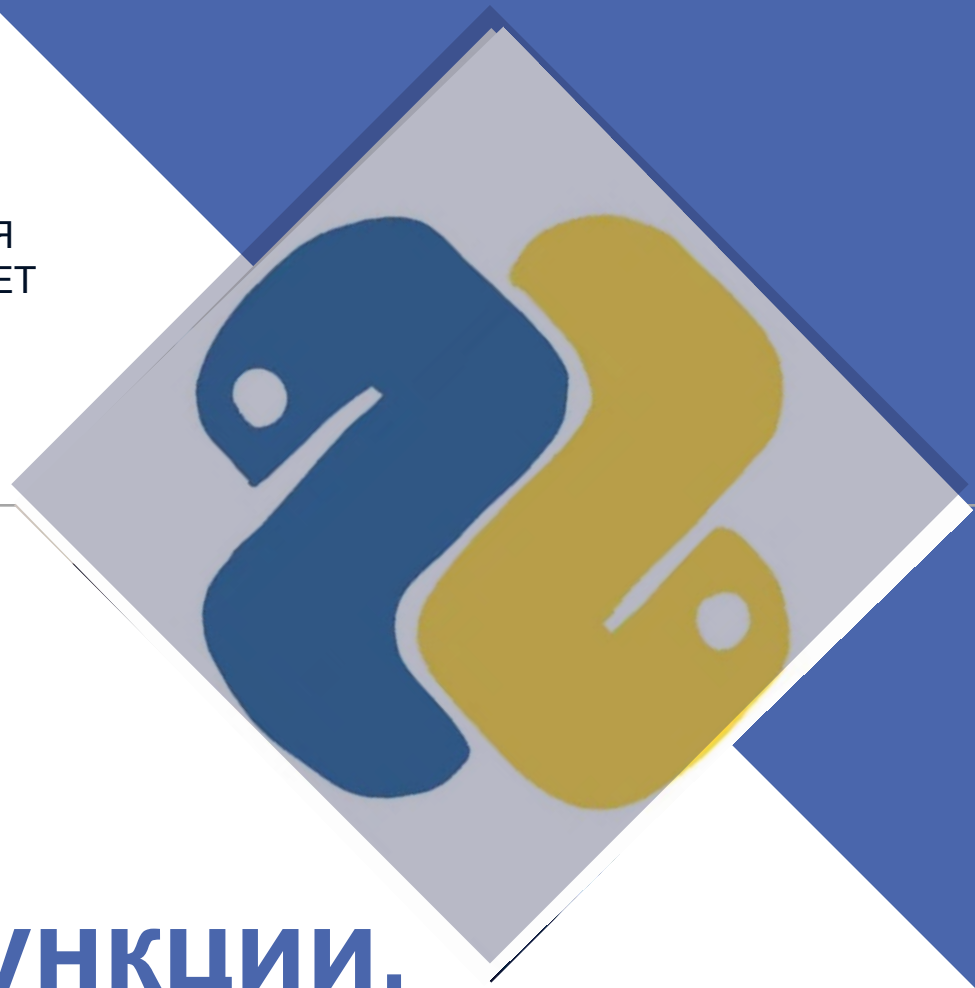
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«КАБАРДИНО-БАЛКАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Х.М. БЕРБЕКОВА»

ИНСТИТУТ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА И ЦИФРОВЫХ
ТЕХНОЛОГИЙ



Рекурсивные функции, анонимные функции, области видимости, вложенные функции

Рекурсивные функции

Рекурсивная функция — это та, которая вызывает сама себя.

Пример

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

Благодаря условной конструкции переменная n вернется только в том случае, если ее значение будет равно 1. Данное условие называют условием завершения.

Рекурсия останавливается в момент удовлетворения условиям.

В блоке else условной конструкции возвращается произведение n и значения этой же функции с параметром n-1.

Факториал числа — это число, умноженное на каждое предыдущее число вплоть до 1.

Пример: **7! = 7*6*5*4*3*2*1 = 5040**

Рекурсия – это возможность некоторого объекта или понятия быть частью самого себя.

В программировании рекурсия – это возможность из тела функции вызвать эту же функцию. Такая возможность иногда позволяет упростить код, но неправильное использование вызовет зацикливание программы.

Детали работы рекурсивной функции

Чтобы еще лучше понять, как это работает, разобьем на этапы процесс выполнения функции с параметром 3. Для этого ниже представим каждый экземпляр с реальными числами. Это поможет «отследить», что происходит при вызове одной функции со значением 3 в качестве аргумента:

```
# Первый вызов
factorial_recursive(3):
    if  3 == 1:
        return 3
    else:
        return 3*factorial_recursive(3-1)

# Второй вызов
factorial_recursive(2):
    if  2 == 1:
        return 2
    else:
        return 2*factorial_recursive(2-1)

# Третий вызов
factorial_recursive(1):
    if  1 == 1:
        return 1
    else:
        return 1*factorial_recursive(1-1)
```

Рекурсия в Python имеет ограничение в 3000 слоев!!!!!

Основные принципы работы рекурсивной функции

Рекурсивная функция должна иметь следующие два свойства:

- **Рекуррентное отношение**
- **Условие прекращения**

Рассмотрим пример, чтобы понять эти моменты. Функция представляет собой конкретное рекуррентное сос

$$f(n) = \begin{cases} f(n-1) & n \geq 1 \\ 1 & n < 1 \end{cases}$$

n ≤ 1 — условие завершения, или условие привязки, или базовое условие. Когда оно выполняется, рекурсия останавливается. Указание этого условия является обязательным. В противном случае функция попадет в бесконечный цикл и будет выполняться непрерывно.

Пример с зацикливанием

```
def func():
    print("text")
    func() # ссылка на саму функции в ее теле

func() # вызов рекурсивной функции
```

```
text
text
text
text
text
text
text
```

Расчет факториала числа N как N * (N – 1)! с учетом, что 1! = 1.

```
def factorial(n):
    if n == 1:
        return 1 # исключение при расчете 1
    else:
        return n * factorial(n - 1) # рекурсия

print(factorial(6))
```

```
720
```

Анонимные функции

Анонимная функция(Лямбда-функция) в Python — это просто функция Python, но это некий особенный тип с ограниченными возможностями, это те функции, которые не имеют уникального имени и объявляются в месте использования. Они используются для сокращения кода в случае, если функции не обязательно давать имя.

В python для анонимных функций используется ключевое слово lambda

Лямбда-функция имеет следующий синтаксис:

Lambda аргументы: выражение

Прежде чем переходить как разбору понятия лямбда в Python, попробуем понять, чем является обычная функция Python на более глубоком уровне.

Для этого потребуется немного поменять направление мышление. Как вы знаете, все в Python является объектом.

Например, когда мы запускаем эту простейшую строку кода x = 5. Создается объект Python типа int, который сохраняет значение 5. x же является символом, который ссылается на объект. Теперь проверим тип x и адрес, на которой он ссылается. Это можно сделать с помощью встроенных функций type и id.

```
>>> type(x)
<class 'int'>
>>> id(x)
4308964832
```

В итоге x ссылается на объект типа int, а расположен он по адресу, который вернула функция id.

А что происходит при определении вот такой функции:

```
>>> def f(x): return x * x
```

Повторим упражнение и узнаем type и id объекта f.

```
>>> def f(x): return x * x
 >>> type(f) <class 'function’>
>>> id(f) 4316798080
```

Оказывается, в Python есть класс function, а только что определенная функция f — это его экземпляр. Так же как x был экземпляром класса integer. Другими словами, о функциях можно думать как о переменных. Разница лишь в том, что переменные хранят данные, а

Рассмотрим простой пример, где функция f передается другой функции.

```
def f(x):
    return x * x

def modify_list(L, fn):
    for idx, v in enumerate(L):
        L[idx] = fn(v)

L = [1, 3, 2]
modify_list(L, f)
print(L)

#вывод: [1, 9, 4]
```

Пример лямбда-функции, удваивающей вводимое значение.

```
double = lambda x: x*2
print(double(5))
```

Вывод:

10

Различие между обычной функцией и лямбда-функцией

Рассмотрим пример и попробуем понять различие между определением (Def) для обычной функции и lambda-функции. Этот код возвращает заданное значение, возведенное в куб:

```
def defined_cube(y):
    return y*y*y


lambda_cube = lambda y: y*y*y
print(defined_cube(2))
print(lambda_cube(2))
```

Вывод:

```
8
8
```

Анонимные функции

Анонимную функцию можно хранить в переменной (делегате)

```
func = lambda: print("it is lambda") # запись анонимной функции в переменную

func() # использование переменной
```

```
it is lambda
```

Анонимные функции

Если анонимная функция имеет параметры, то они определяются после ключевого слова lambda. Если анонимная функция возвращает какой-то результат, то он указывается после двоеточия.

```
plus = lambda a, b: a + b # функция принимает числа a и b и возвращает (a + b)

print(plus(10, 20))
print(plus(30, 40))
```

```
30
70
```

Анонимные функции

Если ваша функция принимает функцию в качестве параметра или возвращает ее в качестве результата – то в таком случае тоже можно использовать анонимные функции.

```
def doit(a, b, operation): # doit ожидает функцию на вход
    print(operation(a, b))

doit(2, 3, lambda x, y: x * y) # в doit передается анонимная функция
```

```
30
70
```

Анонимные функции

Кроме того, анонимные функции могут использоваться как аргумент базовых функций python. Например, функция map позволяет обработать элементы списка и должна получать функцию обработчик и сам список.

```
data = [1, 2, 3, 4, 5] # список
result = map(lambda x: x**2, data) # в map передается функция возведения в квадрат
print(list(result))
```

```
[1, 4, 9, 16, 25]
```

Вложенные функции

Вложенные функции (внутренние функции) – функции, которые определены внутри других функций.

В Python такая функция может иметь доступ к переменным и именам, определенным во включающей функции

Вложенная функция - это просто функция внутри другой функции, и ее иногда называют "внутренней функцией".

```
def [имя функции] ( [аргументы] ):
    [тело функции]
    def [имя вложенной функции] ( [аргументы] ):
        [тело вложенной функции]
```

Вложенные функции

Внутренняя функция может быть вызвана во внешней, но не может использоваться за ее пределами. Это позволит скрыть те функции, которые не нужны за пределами внешней функции.

```
def outer(): # внешняя функция
    print("it is outer")
    def inner(): # внутренняя функция
        print("it is inner")

    inner() # вызов внутренней функции

outer() # запуск внешней функции
# inner() # внутреннюю функцию здесь запустить не выйдет
```

```
it is
outer
it is
inner
```

В Python понятие объекта является ключевым. Они везде! Фактически все, что программа Python создает или с чем работает, — это объект.

Выражение присваивания создает символическое имя, которое вы можете использовать для ссылки на объект. Так выражение x = 'foo' создает символическое имя x, которое ссылается на строковый объект 'foo'.

Пространство имен — это совокупность определенных в настоящий момент символических имен и информации об объектах, на которые они ссылаются.

Существует 4 типа пространств имен:

- **Встроенное.**
- **Глобальное.**
- **Объемлющее.**
- **Локальное.**

Они обладают разными жизненными циклами. По мере выполнения программы Python создает необходимые пространства имен и удаляет их, когда потребность в них пропадает.

Встроенное пространство имен

Встроенное пространство имен содержит имена всех встроенных объектов, которые всегда доступны при работе в Python. Вы можете перечислить объекты во встроенном пространстве с помощью следующей команды:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException','BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError',
'ResourceWarning',
'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__',
'__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',
'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate',
'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Область видимости

Область видимости переменной – эта та область программы (те функции), где можно использовать данную переменную.

Глобальное пространство имен

Глобальное пространство имен содержит имена, определенные на уровне основной программы, и создаётся сразу при запуске тела этой программы. Сохраняется же оно до момента завершения работы интерпретатора.

Локальное и объемлющее пространства имен

Интерпретатор создает новое пространство имен при каждом выполнении функции. Это пространство является локальным для функции и сохраняется до момента завершения ее действия.

Выделяют две основные области видимости:

- **глобальная – переменная доступна во всей программе (и в ее функциях)**
- **локальная – только в той функции, где переменная определена**

Python будет искать его следующих областях видимости в таком порядке:

1. **Локальная.** Если вы ссылаетесь на x внутри функции, то интерпретатор сначала ищет его в самой внутренней области, локальной для этой функции.
2. **Объемлющая**. Если x не находится в локальной области, но появляется в функции, располагающейся внутри другой функции, то интерпретатор ищет его в области видимости объемлющей функции.
3. **Глобальная.** Если ни один из вышеуказанных вариантов не принес результатов, то интерпретатор продолжит поиск в глобальной области видимости.
4. **Встроенная**. Если интерпретатор не может найти x где-либо еще, то он направляет поиски во встроенную область видимости.

Область видимости

Если переменная определена вне функций – она глобальная и доступна во всех функциях

```
name = "Ezio" # глобальная переменная, определена вне функций

def Hi():
    print("Hi", name) # функция использует глобальную переменную
def Hello():
    print("Hello", name)

Hi() # вызов функций
Hello()
```

```
Hi Ezio
Hello Ezio
```

Область видимости

Локальная переменная доступна только в той функции, в которой она определена

```
def Hi():
    name = "Ezio"  # локальная переменная функции Hi
    print("Hi", name) # функция использует локальную
переменную
def Hello():
    name = "Altair" # дрругая локальная переменная (имя
переменной может повторяться в разных контекстах)
    print("Hello", name)

Hi() # вызов функций
Hello()
```

```
Hi Ezio
Hello
Altair
```

Область видимости

Если в функции есть локальная переменная с таким же именем, что и у глобальной – то в рамках данной функции будет использоваться только локальная переменная (вместо глобальной).

```
name = "Altair" # глобальная переменная

def Hi():
    name = "Ezio"  # локальная переменная скрывает
глобальную
    print("Hi", name) # функция использует локальную
переменную
def Hello():
    print("Hello", name) # а здесь используется глобальная


Hi() # вызов функций
Hello()
```

```
Hi Ezio
Hello
Altair
```

Область видимости

Ключевое слово global в функции позволяет обращаться к глобальной переменной и менять ее значение

```
name = "Altair" # глобальная переменная
def Hi():
    global name # обращаемся к глобальной переменной
    name = "Ezio"  # можем изменить значение глобальной
переменной
    print("Hi", name)
def Hello():
    print("Hello", name) # а здесь используется глобальная

Hello() # имя еще не изменили
Hi() # функция HI меняет глобальную переменную с именем
Hello() # теперь у глобальной переменной новое значение
```

```
Hello
Altair
Hi Ezio
Hello Ezio
```

Область видимости

При этом у вложенных функций могут быть свои локальные переменные

```
temp = 0 # глобальная переменная
def outer():
    temp = 23 # локальная (для outer)

    def inner():
        temp = -10 # локальная (для inner)
        print(temp)
    inner()
    print(temp)

outer() # печать двух разных локальных temp
print(temp) # и печать глобальной temp
```

```
-10
23
0
```

Область видимости

Если во вложенной функции необходимо использовать локальную переменную внешней функции – используется

```
temp = 0 # глобальная переменная
def outer():
    temp = 23 # локальная (для outer)

    def inner():
        nonlocal temp # будет использоваться локальная
переменная внешней функции
        temp = -10 # изменяем внешнюю локальную переменную
        print(temp)
    inner()
    print(temp)
outer() # печать двух локальных переменных
print(temp) # и печать глобальной temp
```

```
-10
-10
0
```

Замыкания

Замыкания – это функции, которая запоминает свое окружение (состояние внешней функции) даже если она выполняется вне своей области видимости.

Для этого необходимо:

- **внешняя функция в которой определены переменные (окружение)**
- **вложенная функция, которая использует это окружение**
- **внешняя функция возвращает вложенную**

Замыкания

Например, функция inner запоминает состояние внешнего окружения (переменную n функции power) и использует в расчетах

```
def power(n): # внешняя функция с переменной n
    def inner(m): # внутренняя функция
        return n ** m # использование окружения (n)
    return inner # power возвращает внутреннюю функцию

fn = power(10) # fn = inner()
print(fn(3)) # возведение 10 в степени разных чисел
print(fn(4))
print(fn(5))
```

```
1000
10000
100000
```

Декораторы

Декораторы - функция, которая в качестве параметра получает функцию и в качестве результата также возвращает функцию. Декоратор позволяет модифицировать исходную функцию без явного изменения исходного кода функции.

```
def [имя декоратора] ( [имя функции] ):
    [тело декоратора]
    return [модифицированная функция]

@ [имя декоратора]
def [имя исходной функции] ( [аргументы] ):
    [тело исходной функции]
```

Декораторы

Например, декоратор может добавить к исходной функции дополнительный функционал

```
def select(input_func): # определение функции декоратора
    def output_func():  # определяем функцию, которая будет выполняться вместо оригинальной
        print("Hello")  # модификация оригинальной функции
        input_func()  # вызов оригинальной функции

    return output_func  # возвращаем новую функцию

@select  # применение декоратора select
def hello(): # определение оригинальной функции
    print("World") # содержимое оригинальной функции

hello() # вызов оригинальной функции
```

```
Hello
World
```

Пример1. Необходимо создать функцию setFuncs, принимающую список чисел и две другие функции в качестве аргумента. Функция должна применять к четным первую функцию-аргумент, а к нечетным - вторую и выводить результат. Проверьте работу созданной функции передав ей список чисел и функции возведение в квадрат и возведение в куб. Функции надо передавать в виде лямбда функций.

```
data = [1, 2, 3, 4, 5]
setFuncs(data, lambda a: a**2, lambda a: a**3)
```

```
def setFuncs(data, firstFunc, secondFunc):
    for i in data:
        if (i%2==0):
            print(firstFunc(i))
        else:
            print(secondFunc(i))

data = [1, 2, 3, 4, 5]
setFuncs(data, lambda a: a**2, lambda a: a**3)
```

Пример 2. Создайте функцию story для вывода всех имен из списка (который передается как аргумент). При создании функции вместо цикла нужно использовать рекурсию (то есть вызывать функцию из нее же).

```
story(["jack", "john", "james"])
```

```
hello, jack
hello, john
hello, james
```

```
def story(names):
    print("hello,", names[0])
    if (len(names)>1):
        story(names[1:])

story(["jack", "john", "james"])
```

Пример 3. Создайте функцию showStars, которая получает список чисел и выводит такое же количество звездочек в строку (через пробел), то есть при вводе 2 3 5 на выход должны вывестись ** * *****. Вывод указанного количества звездочек реализуйте в качестве внутренней функции.**

```
showStars([2, 4, 6, 8, 10])
```

```
**  ****  ******  ********  **********
```

```
def showStars(data):
    def stars(n):
        return (n * "*")
    text=""
    for i in data:
        text = text + stars(i) + "  "
    print(text)

showStars([2, 4, 6, 8, 10])
```

Пример 4. Создайте программу с глобальной переменной temp (температура в помещении) и двумя функциями для heat и cold (нагрев и охлаждение). Обе функции принимают значение времени (time) и имеют свои (различные) коэффициенты эффективности (coef), при этом первая увеличивает температуру на time*coef, а вторая уменьшает. Проверьте работу вашей программы.

```
print(temp)
heat(2)
print(temp)
cold(2)
print(temp)
```

```
temp=20
def heat(time):
    coef = 2
    global temp
    temp += coef * time
def cold(time):
    coef = 5
    global temp
    temp -= coef * time

print(temp)
heat(2)
print(temp)
cold(2)
print(temp)
```

Пример 4

```
temp=20
def heat(time):
    coef = 2
    global temp
    temp += coef * time
def cold(time):
    coef = 5
    global temp
    temp -= coef * time

print(temp)
heat(2)
print(temp)
cold(2)
print(temp)
```

Задача 1. Создайте функцию calc, которой можно передать символ + - * /, а функция должна вернуть соответствующую функцию, то есть при, вызове func("+") мы должны получить функцию, которая складывает два числа. Для проверки присвойте двум переменным результат созданной вами функции (например, для * и для /) и проверьте работу этих переменных на разных парах чисел.

```
mul = calc("*")
div = calc("/")
print(mul(10, 2))
print(div(10, 2))
```

```
def calc(operation):
    ?????????????

mul = calc("*")
div = calc("/")

print(mul(10, 2))
print(div(10, 2))
```

Задача 2. Необходимо создать функцию changeRange, которая принимает два числа (start и end) и функцию operation. В ней над всеми элементами ряда от start до end необходимо совершить operation. Функция должна вернуть сумму всех чисел от start до end, применив переданную ей операцию. Проверьте функцию, передав ей два числа и лямбда функцию (например, умножения на 10)

```
print(changeRange(1, 2, lambda a: a*10))
```

```
def changeRange(start, end, operation):
    ??????????????
    return (sum)

print(changeRange(1, 2, lambda a: a*10))
```

Задания для самостоятельного решения

1. Напишите рекурсивную функцию factorial(n), которая будет принимать положительное целое число n и возвращать факториал от этого числа (1 x 2 x 3 x … x n).
2. Напишите рекурсивную функцию summation(n), которая будет принимать положительное целое число n и возвращать сумму чисел от 1 до n.
3. Напишите рекурсивную функцию sum_odd(lis), которая будет принимать список из целых чисел и возвращать сумму только нечётных чисел из списка.
4. Напишите рекурсивную функцию sum_sub(list), которая будет принимать список целых чисел. Эта функция будет суммировать все нечётные числа и вычитать все чётные числа. В конце она будет возвращать получившееся значение.
5. Слова-палиндромы — это строки, которые одинаково читаются с обеих сторон. Например: «aba», «abba», «abcba», «aaa», «ababa» и так далее.
6. Напишите функцию is_palindrome(string), которая будет принимать строку и возвращать «True», если строка является палиндромом, и «False» во всех других случаях.
7. Напишите рекурсивную функцию fib(n), которая будет принимать положительное целое число n и возвращать n-ое число Фибоначчи.
8. Числа Фибоначчи — это серия чисел, которая начинается с 0 и 1. Каждое последующее число будет являться суммой двух предыдущих чисел.

СПАСИБО ЗА ВНИМАНИЕ!