Практическое занятие 1

Математический анализ

https://docs.sympy.org/latest/tutorial/intro.html (https://docs.sympy.org/latest/tutorial/intro.html)

In [1]:
```
#Вначале для простоты будем подключать модуль sympy целиком
from sympy import *
```

Действия с числами, числовые выражения

Об основных типах данных Python 3.8 читайте здесь:

https://docs.python.org/3/reference/datamodel.html#index-19 (https://docs.python.org/3/reference/datamodel.html#index-19)

Сложение и вычитание как обычно "+" и "-", деление "/", умножение "*".

Возведение в степень в Python изображается $**$, например, 2^3 будет $2 * *3$.

Списки list, кортежи tuple, диапазон range

В Python есть упорядоченные последовательности элементов list и tuple, похожие на массивы, но в отличие от массивов их элементы не обязательно являются данными одного типа.

list изменяемый тип, tuple неизменяемый.

Создать list можно несколькими способами:

In [2]:
```
numbers = [-2, 4, 7, 1001]
odd_numbers = [2*i+1 for i in range(9)]
even_numbers = []
for i in range(1, 10):
    even_numbers += [2*i]
display(numbers)
display(odd_numbers)
display(even_numbers)
```

```
[-2, 4, 7, 1001]
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17]
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Попробуем созать tuple, заменив в коде для list квадратные скодки на круглые:

In [3]:
```
numbers = (-2, 4, 7, 1001)
display(numbers)
odd_numbers = (2*i+1 for i in range(9))
display(odd_numbers)
```

```
(-2, 4, 7, 1001)
```

```
<generator object <genexpr> at 0x00000216CA7FA5E8>
```

Вместо ожидаемого tuple теперь odd_numbers явояется генератором, т.е. правилом или алгоритмом получения последовательности, а не самой последовательностью. Починим odd_numbers, прменив к генератору функцию tuple:

In [4]:
```
odd_numbers = tuple(2*i+1 for i in range(9))
display(odd_numbers)
```

```
(1, 3, 5, 7, 9, 11, 13, 15, 17)
```

Поскольку у круглые скобки используются для выделения части математического выражения, то для добавления в инициирующему пустому tuple элемента $2 * i$ необходимо поставить запятую после $2 * i$, чтобы получилось $(2 * i,)$. Запятая всегда нужна в tuple, состоящем из одного элемента.

In [5]:
```
even_numbers = ()
for i in range(1, 10):
    even_numbers += (2*i,)
display(even_numbers)
```

```
(2, 4, 6, 8, 10, 12, 14, 16, 18)
```

Для красивого отображения формул использовали display().

Символы, символьные выражения

Для аналитических преобразований в sympy используется класс Symbol

https://docs.sympy.org/latest/modules/core.html?highlight=symbol#module-sympy.core.symbol (https://docs.sympy.org/latest/modules/core.html?highlight=symbol#module-sympy.core.symbol)

В этом классе есть метод Symbol для создания одного символа, метод symbols для создания нескольких символов одновременно.

Пример 1.

Создадим символ x и символы y, z, затем создадим символы $t_1, \ldots, t_6$.

Для распаковки кортежа символов используем $*$. Сравните результат с и без $*$.

In [6]:
```
x = Symbol('x')
y, z = symbols('y, z')
t = symbols('t1:7') # Обратите внимание, что последний номер 7, а не 6! Дело в том, что последний номер НЕ включается!
display(x, y, z, t, *t)
```

x

y

z

(t1, t2, t3, t4, t5, t6)

t_1

t_2

t_3

t_4

t_5

t_6

Функции пользователя

Для более наглядного и удобного решения задачи былает удобно разбить ее на подзадачи и каждую подзадачу оформить в виде функции. Функции бывают встроенные, такие как $\sin$ и $\log$, их можно использовать, подключив соответствующий модуль, например, Sympy. Можно написать собственные функции следующим образом:

def function_name(arg1,\ ...,\ arg2=Value):

```
    .....

    return something
```

Ключевое слово return можно опустить, тогда функция вернет в качестве результата None.

У функции могут быть только обязательные аргументы, но могут быть и аргументы со значениями по умолчанию (необязательные аргументы). Вначале опишем функцию f с обязательными аргументами x и a.

При вызове функции аргументы передаются по порядку, в нашем случае сначала значение x, потом a.

Пример 2

Опишем функцию $func\_power(x, a) = x^a$:

In [7]:
```
def func_power(x, a):
    return x**a
```

При вызове функции сначала передаем значение x, потом a.

In [8]:
```
func_power(2,3)
```

Out[8]: 8

Можно передавать в качестве аргументов не только числа, но символы:

In [9]:
```
a = symbols('a')
func_power(2, a)
```

Out[9]: 2^a

Аргументом функции может быть и имя другой функции.

Пример 3

Опишем функцию, вычисляющую разность значений функции f в точках x_1 и x_2, т.е. $f(x_2) - f(x_1)$.

In [10]:
```
def delta_f(f, x1, x2):
    return f(x2) - f(x1)
```

При вызове функции сначала передаем имя функции, потом точки x_1 и x_2. Возьмем в качестве функции $\sin$, а точки $\pi/6$ и $\pi/3$.

In [11]:
```
delta_f(sin, pi/6, pi/3)
```

Out[11]:
$$-\frac{1}{2} + \frac{\sqrt{3}}{2}$$

В качестве аргументов можно передать и значение функции, числовое или символьное. Пусть теперь функцией будет $\log$, а точки - значения функции $func\_power$ с аргументами (E, 3) и (a+1, a) соответственно.

In [12]:
```
delta_f(log, func_power(E, 3), func_power(a+1, a))
```

Out[12]:
$$\log\left((a+1)^a\right) - 3$$

Необязательные аргументы

Необязательные аргументы или аргументы со значением по умолчанию передаются всегда ПОСЛЕ обязательных аргументов!!!

Пример 4

Опишем функцию $g(x) = \log_a(x)/x$ с параметром a, по умолчанию равным числу e:

In [13]:
```
def g(x, a=E):
    return log(x, a)/x
```

При вызове функции передадим только обязательный аргумент, если нас устраивает значение по умолчанию

In [14]:
```
g(5)
```

Out[14]:
$$\frac{\log(5)}{5}$$

Обратим внимание, что $\log(x, a) = \ln(x)/\ln(a)$, со значением по умолчанию $a = e$, так что натуральный логарифм обозначается $\log$, а не $\ln$, как мы привыкли.

Если вместо значения по умолчанию нужно передать другое значение, просто передадим его в позиции нашего необязательного аргумента.

In [15]:
```
g(5, 2)
```

Out[15]:
$$\frac{\log(5)}{5\log(2)}$$

Как уже заметили, все вычисления по умолчанию выполняются аналитически, никаких приближенных значений и округлений.

Округленное значение можно получить несколькими способами.

Например, можно воспользоваться функциями round, ceiling, floor.

In [16]:
```
log_2_5=g(5, 2)
round(log_2_5, 4),  ceiling(log_2_5), floor(log_2_5)
```

Out[16]:
```
(0.4644, 1, 0)
```

Другой способ: можно воспользоваться методом evalf():

In [17]:
```
log_2_5.evalf()
```

Out[17]:
$$0.464385618977472$$

У этого метода есть параметр со значением по умолчанию, равный числу знаков после запятой, этот параметр можно передать при вызове метода и получить значение числового выражения, округленное до k знаков после запятой, например, так:

In [18]:
```
log_2_5.evalf(4)
```

Out[18]:
$$0.4644$$

Округление производится по знакомым из школы правилам.

Убедимся, что при этом само значение log_2_5 не изменилось:

In [19]:
```
log_2_5
```

Out[19]: $\frac{\log(5)}{5\log(2)}$

Функцию round тоже можно использовать как метод:

In [20]:
```
log_2_5.round(4)
```

Out[20]: 0.4644

А с ceiling и floor так не получится:

In [21]:
```
log_2_5.ceiling(), log_2_5.floor()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-21-2de39ed23014> in <module>
----> 1 log_2_5.ceiling(), log_2_5.floor()

AttributeError: 'Mul' object has no attribute 'ceiling'
```

Еще один способ передачи необязательных аргументов

Аргументы со значениями по умолчанию можно передавать, обращаясь к ним по имени. Например, при вызове функции g можно сделать так:

In [22]:
```
g(3, a=2)
```

Out[22]: $\frac{\log(3)}{3\log(2)}$

Такой способ удобен, когда у функции много параметров по умолчанию, а изменить нужно только некоторые из них:

Пример 5

In [23]:
```
def h(x, a=1, b=2, c=3, d=5):
    return x**a + x**b + x**c + x**d
h(a, c=-1)
```

Out[23]: $a^5 + a^2 + a + \frac{1}{a}$

Еще раз о циклах и условном операторе

В теле функции можно использовать циклы, условные операторы и многое другое, поэтому напомним о них.

Цикл for

Пример 6

Опишем функцию sum_even, которая создает $n > 0$ символов $x_2, \ldots x_{2n}$ с четными номерами и выдает в качестве результата их сумму. Для счетчика цикла используем range() (это неизменяемая последовательность целых чисел, используемая очень часто в качестве счетчика в цикле for). У range есть три необязательных аргумента, начало, конец и шаг, по умолчанию начало 0, конец и шаг 1. Важно!!! Конец не включается!!!

В нашем примере надо начинать с 2, конец $2n + 1$, поскольку последний нужный номер $2n$, а шаг 2.

Для превращения числа в текст используем str(), это позволяет создавать символы с нужными номерами.

In [24]:
```
def sum_even(n):
    res=0
    for k in range(2, 2*n + 1, 2):
        res += Symbol('x' + str(k))
    return res
```

In [25]:
```
sum_even(4)
```

Out[25]: $x_2 + x_4 + x_6 + x_8$

Условный оператор (if, elif, else)

Полное описание условного оператора:

if условие 1:

    тело 1

elif условие 2:

    тело 2

elif условие 3:

    тело 3

...

else:

    тело n

Если выполняется условие 1, то выполняется тело 1, а условия 2, 3, ..., n не проверяются (elif означает else if) и соответствующие тела не выполняются.

Если не выполняется условие 1, но выполняется условие 2, то условия 3, ..., n не проверяются и соответствующие тела не выполняются.

Для следующего примера понядобится функция input() с необязательным аргументом - приглашением ввода, а также функция int(), преобразующая текст, состоящий из цифр, в число в десятичной системе.

Пример 7.

Пусть программа просит пользователя ввести трехзначное натуральное число и пишет 'Спасибо!', если введено именно трехзначное число, 'Это двузначное число!', если введено двузначное число, 'Это однозначное число!', если однозначное и 'Это не натуральное число!', если введенное число меньше или равно 0. Предполагается, что пользователь может ввести только целые числа.

In [26]:
```
x = int(input('Введите трехзначное натуральное число: '))
if x <= 0:
    print('Это не натуральное число!')
elif x < 10:
    print('Это однозначное число!')
elif x < 100:
    print('Это двузначное число!')
elif x < 1000:
    print('Спасибо!')
else:
    print('Что-то пошло не так!')
```

```
Введите трехзначное натуральное число: 125
Спасибо!
```

Цикл while

while условие:

    тело

Вычислим, при каком n выполняется $n! < 10^{10}$. Для этого инициируем единицей переменную res, а затем в цикле будем домножать ее на n, пока $res < 10^{10}$. Выведем значение $n-1$, а также $n!$. (Подумайте, зачем нужно res/n)

In [27]:
```
res = 1
n = 1
while res < 10**10:
    n += 1
    res *= n
n-1, int(res/n)
```

Out[27]: (13, 6227020800)

Проверим, что это действительно нужное число и его факториал, для этого воспользуемся циклом for и диапазоном range(1,14).

Еще раз вспомним, что последний номер 14 не учитывается!

In [28]:
```
res = 1
for i in range(1,14):
    res *= i
i, res
```

Out[28]: (13, 6227020800)

Полное описание цикла while:

while условие:

    тело цикла 1

else:

```
тело 2
```

Усли даже условие цикла while не выполняется ни разу, тогда при наличии блока else все равно выполняется тело цикла 2. Блок else выполняется один раз в любом случае, независимо от того, сколько раз выполнилось тело цикла 1.

Чаще while используется без else, результат такой же, как если написать

while условие:

```
тело цикла 1
```

тело 2

кроме случая, если для выхода из цикла в теле цикла 1 используется команда break, тогда в случае с else тело 2 не будет исполнено, в отличие от случая без else.

Пример 8.

Будем искать наименьшее число $0 < t < M$, такое что $a < f(t) < b$ при $t = ks$, где k - натуральное число, $s < M$ - положительное число (шаг сетки, цена деления линейки и т.п., не обязательно целое число). Опишем функцию $g(f, a, b, s, M)$ с параметрами a, b, s, M, равными по умолчанию соответственно 0, 1, 1, 100. Функция возвращает tuple $(t, f(t))$ при $a < f(t) < b$ и tuple $(None, None)$, если подходящего t нет. Обратите внимание на часть else цикла while, она выполнится, например, если в самом начале, т.е. при $t = 0$, не будет удовлетворяться условие $f_t <= a$ *or* $f_n >= b$, что означает выполнение условия $a < f(t) < b$. Тело else выполнится также, если при выполнении тела while ни разу не случится ситуация $t >= M$, при которой работает break (в этом случае выполняется последняя команда return (None, None)). Таким образом, при условии корректных значений всех аргументов ($a < b$, $0 < s < M$) функция $g(f, a, b, s, M)$ вернет правильный ответ или tuple $(None, None)$, если подходящего числа на указанном промежутке нет.

Заметим, что в данном случае можно обойтись и без break, используя вместо него сразу return (None, None).

Протестируем полученную функцию на g(sin, M=2), g(sin, s=0.1, M=2), g(sin, a=1, b=2).

In [29]:
```
def g(f, a=0, b=1, s=1, M=100):
    t = 0
    f_t = f(t)
    while t < M and (f_t <= a or f_t >= b):
        t += s
        if t >= M:
            break
        f_t = f(t)
    else:
        return (t, f_t)
    return (None, None) # Если программа сюда дошла, значит прошли весь интервал ($a,b$), но нужного числа не нашли
g(sin, M=2), g(sin, s=0.1, M=2), g(sin, a=1, b=2)
```

Out[29]:
```
((1, sin(1)), (0.1, 0.0998334166468282), (None, None))
```