```
# 7 raise to the power 2 below


# divide 7 by 2 nd round it to the nearest integer
```

Expected output:

9

5

14

3.5

1

49

3

---

3. Order of Operations:

Now Arithmetic operators in python follow the **BEDMAS** order of operations.

Look at this mathematical equation (2+3*5) what do you think will be the answer 25 or 17. The answer is 17 because in maths we follow a convention called as the order of operations BEDMAS ie.

```
MYPYTHON="PythonGuides"

myPython="PythonGuides"

myPython7="PythonGuides"
```

[ ]

```
#Variable name not Allowed


7mypython="PythonGuides"

-mypython="PythonGuides"

myPy@thon="PythonGuides"

my Python="PythonGuides"

for="PythonGuides"


#It shows invalid syntax

#It will execute one by one and will show the error.
```

Also there are some naming convention that needs to be followed like:

- try to keep the name of the variables descriptive short but descriptive. **for example:** when taking inputs for the height of a tree of a box the appropriate variable name will be just *height* not *x* not *h* not *height_of_the_tree*.
- Also the pythonic way to name variables is to use all lowercase letters and underscores to separate words.

[ ]

Methods are like some of the functions you have already seen:

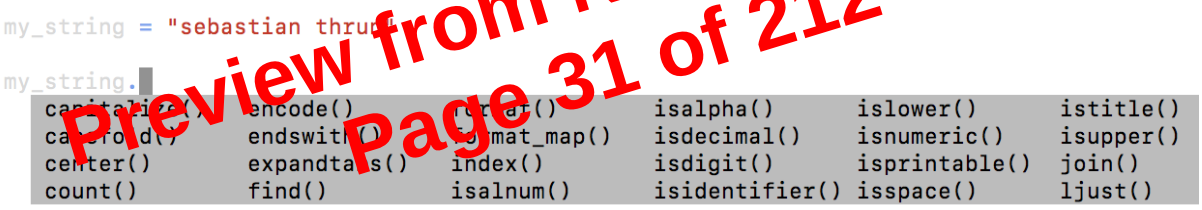
- len("this")
- type(12)
- print("Hello world")

These three above are functions - notice they use parentheses, and accept one or more arguments. Functions will be studied in much more detail in a later lesson!

A method in Python behaves similarly to a function. Methods actually are functions that are called using dot notation. For example, *lower()* is a string method that can be used like this, on a string called **"sample string"**: *sample_string.lower()*.

Methods are specific to the data type for a particular variable. So there are some built-in methods that are available for all strings, different methods that are available for all integers, etc.

Below is an image that shows some methods that are possible with any string.



Each of these methods accepts the string itself as the first argument of the method. However, they also could receive additional arguments, that are passed inside the parentheses. Let's look at the output for a few examples.

[ ]

```
my_string = "ShapeAI"

print(my_string.islower())
```

```
print(students[-3])
```

If you try to access an index in a list that doesn't exist then you will get an Error as seen below.

[ ]

```
print(students[20])
```

Question:

Try to use *len()* to pull the last element from the above list

---

[ ]

```
# TODO: write your code here
students[len(students)-1]
```

13.a. Membership Operators:[Lists]

In addition to accessing individual elements froma a list, we can use pythons sliceing notation to access a subsequence of a list.

Slicing means using indicies to slice off parts of an object like list/string. Look at an example

---

[ ]

```
students = ['sam', 'pam', 'rocky', 'austin', 'steve', 'banner', 'tony', 'bruce',
```

```
print(flash)
```

[ ]

```
# length of the list and the string

print(len(students))

print(len(student))
```

Of the types we have seen lists are most familier to strings, both supports the *len()* function, indexing and slicing.

Here above you have seen that the length of a string is the no of characters in the string, while the length of a list is the no of elements in the list.

Another thing that they both supports aare membership operators.

- **in:** evaluates if an object on the left side is included in the object on the right side.
- **not in:** evaluates if object on left side is not included in object on right side.

[ ]

```
greeting = "Hello there"

print('her' in greeting, 'her' not in greeting)
```

[ ]

```
print('ShapeAI' in students, 'ShapeAI' not in students)
```

13.b. Mutability and Order:

```
vector = (4, 5, 9)

print("x-coordinate:", vector[0])

print("y-coordinate:", vector[1])

print("z-coordinate:", vector[2])
```

Tuples are similar to lists in that they store an ordered collection of objects which can be accessed by their indices. Unlike lists, however, tuples are immutable - you can't add and remove items from tuples, or sort them in place.

Tuples can also be used to assign multiple variables in a compact way.

The parentheses are optional when defining tuples, and programmers frequently omit them if parentheses don't clarify the code.

[ ]

```
location = 108.7774, 92.5556

latitude, longtitude = location

print("The coordinates are {} x {}".format(latitude, longtitude))
```

In the second line, two variables are assigned from the content of the tuple location. This is called tuple unpacking. You can use tuple unpacking to assign the information from a tuple into multiple variables without having to access them one by one and make multiple assignment statements.

If we won't need to use location directly, we could shorten those two lines of code into a single line that assigns three variables in one go!

---

4. Following the *for* loop heading is an indented block of code, the body of the loop, to be executed in each iteration of this loop. There is only one line in the body of this loop - *print(city)*.
5. After the body of the loop has executed, we don't move on to the next line yet; we go back to the for heading line, where the iteration variable takes the value of the next element of the iterable. In the second iteration of the loop above, *city* takes the value of the next element in *cities*, which is "mountain view".
6. This process repeats until the loop has iterated through all the elements of the iterable. Then, we move on to the line that follows the body of the loop - in this case, *print("Done!")*. We can tell what the next line after the body of the loop is because it is unindented. Here is another reason why paying attention to your indentation is very important in Python!

You can name iteration variables however you like. A common pattern is to give the iteration variable and iterable the same names, except the singular and plural versions respectively (e.g., 'city' and 'cities').

Using the *range()* Function with for Loops:

range() is a built-in function used to create an iterable sequence of numbers. You will frequently use *range()* with a for loop to repeat an action a certain number of times, as in this example:

[ ]

```
for i in range(3):
    print("Hello!")
```

```
Hello!
```

Hello!

Hello!

---

range(start=0, stop, step=1)

The range() function takes three integer arguments, the first and third of which are optional:

- The 'start' argument is the first number of the sequence. If unspecified, 'start' defaults to 0.
- The 'stop' argument is 1 more than the last number of the sequence. This argument must be specified.
- The 'step' argument is the difference between each number in the sequence. If unspecified, 'step' defaults to 1.

Notes on using range():

- If you specify one integer inside the parentheses withrange(), it's used as the value for 'stop,' and the defaults are used for the other two. Example-

---

[ ]

```
for i in range(4):
  print(i)
```

0

1

2

3

- If you specify two integers inside the parentheses withrange(), they're used for 'start' and 'stop,' and the default is used for 'step.' Example-

---

[ ]

```
for i in range(2, 6):
  print(i)
```

```
2
3
4
5
```

- Or you can specify all three integers for 'start', 'stop', and 'step.' Example-

---

[ ]

```
for i in range(1, 10, 2):
  print(i)
```

```
1
3
5
7
9
```

[ ]

```
names = ["Joey Tribbiani", "Monica Geller", "Chandler Bing", "Phoebe Buffay"]

usernames = []


# write your for loop here



print(usernames)
```

Question:

Write a for loop that iterates over a list of strings, tokens, and counts how many of them are XML tags.

XML is a data language similar to HTML. You can tell if a string is an XML tag if it begins with a left angle bracket "<" and ends with a right angle bracket ">".

Keep track of the number of tags using the variable count.

You can assume that the list of strings will not contain empty strings.

[ ]

```
tokens = ['<greeting>', 'Hello World!', '</greeting>']

count = 0


# write your for loop here
```

[ ]

```
# number to find the factorial of

number = 6


# start with our product equal to one

product = 1


# write your for loop here




# print the factorial of number

print(product)
```

Question:

Suppose you want to count from some number start_num by another number count_by until you hit a final number end_num. Use break_num as the variable that you'll change each time through the loop. For simplicity, assume that end_num is always larger than start_num and count_by is always positive.

Before the loop, what do you want to set break_num equal to? How do you want to change break_num each time through the loop? What condition will you use to see when it's time to stop looping?

After the loop is done, print out break_num, showing the value that indicated it was time to stop looping. It is the case that break_num should be a number that is the first number larger than end_num.

[ ]

```
start_num = 5

end_num = 30

count_by = 3


# write a while loop that uses break_num as the ongoing number to
#   check against end_num


print(break_num)
```

Break and Continue:

For Loop iterate over every element in a sequence, while loops iterate over every element untill stopping condition is met.

This is sufficient for most purposes, but sometimes we need more precise control over when we need to end a loop.

In these cases we use the *break* keyword.

```
        "Brave Knight Runs Away",
        "Papperbok Review: Totally Triffic"]

news_ticker = ""
# write your loop here


print(news_ticker)
```

Zip and Enumerate:

zip and enumerate are useful built-in functions that can come in handy when dealing with loops.

Zip:

zip returns an iterator that combines multiple iterables into one sequence of tuples. Each tuple contains the elements in that position from all the iterables. For example, printing

```
manifest = [("bananas", 15), ("mattresses", 24), ("dog kennels", 42),
        ("machine", 120), ("cheeses", 5)]
```

[ ]

```
items = ['bananas', 'mattresses', 'dog kennels', 'machine', 'cheeses']
```

```
points = []

# write your for loop here



for point in points:

    print(point)
```

Question:

Use zip to create a dictionary cast that uses names as keys and heights as values.

```
[ ]

cast_names = ["Joey Tribbiani", "Monica Geller", "Chandler Bing",
              "Ross Geller", "Phoebe Buffay"]


cast_heights = [172, 168, 172, 166, 170]


cast = # replace with your code

print(cast)
```

Question:

Use zip to transpose data from a 4-by-3 matrix to a 3-by-4 matrix.

[ ]

```
data = ((0, 1, 2), (3, 4, 5), (6, 7, 8), (9, 10, 11))


data_transpose = # replace with your code

print(data_transpose)
```

Question:

Use enumerate to modify the cast list so that each element contains the name followed by the character's corresponding income in $ 1000. For example, the first element of cast should change from "Barney Stinson" to "Barney Stinson 150".

[ ]

```
cast = ["Barney Stinson", "Robin Scherbatsky", "Ted Mosby",
        "Lily Aldrin", "Marshall Eriksen"]

heights = [150, 100, 40, 0, 80]


# write your for loop here



print(cast)
```

List Comprehensions:

List comprehensions allow us to create a list using a for loop in one step.

You create a list comprehension with brackets [], including an expression to evaluate for each element in an iterable. This list comprehension above calls city.title() for each element city in cities, to create each element in the new list, capitalized_cities.

Conditionals in List Comprehensions:

You can also add conditionals to list comprehensions (listcomps).

Lets look at an example:

```
[ ]

 squares = []


 for x in range(9):
  squares.append(x**2)


 print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64]
```


---

writing the same as a list comprehension will be like:

```
[ ]

 squares = [x**2 for x in range(9)]
```

To define a function we first write the keyword "def" followed by "name of the function" and round paranthesis "()".

```
def Fun_Name():
  ## Here we just defined a function
  ## The name of the function is Fun_Name
  ## COOL XD
```

---

2. Writing the Function body

After defining the function now you need to write the body of the function which is the working of youe function *i.e. what you want your function to do for you*

```
def Hello():
  print('Hello')
```


The code above will print Hello whenever you call the function.

Don't worry we will see calling function in details later...

[ ]

```
#LET'S TRY THE ABOVE CODE
def Hello():
  print('Hello')
```

```
print(Sub(5,2))    #EXPEXTED OUTPUT: 3
```

3

Components of a Lambda Function:

1. The lambda keyword is used to indicate that this is a lambda expression.
2. Following lambda are one or more arguments for the anonymous function separated by commas, followed by a colon :. Similar to functions, the way the arguments are named in a lambda expression is arbitrary.
3. Last is an expression that is evaluated and returned in this function. This is a lot like an expression you might see as a return statement in a function.

With this structure, lambda expressions aren't ideal for complex functions, but can be very useful for short, simple functions.

Question:

map() is a higher-order built-in function that takes a function and iterable as inputs, and returns an iterator that applies the function to each element of the iterable. The code below uses map() to find the mean of each list in numbers to create the list averages. Give it a test run to see what happens.

Rewrite this code to be more concise by replacing the mean function with a lambda expression defined within the call to map().

[ ]

```
numbers = [
```

```
# We create a 3 x 2 ndarray full of ones.
X = np.ones((3,2))

# We print X
print()
print('X = \n', X)
print()

# We print information about X
print('X has dimensions:', X.shape)
print('X is an object of type:', type(X))
print('The elements in X are of type:', X.dtype)
```

X =

[[1. 1.]

[1. 1.]

[1. 1.]]

X has dimensions: (3, 2)

X is an object of type: <class 'numpy.ndarray'>

The elements in X are of type: float64

np.full():

We can also create an ndarray with a specified shape that is full of any number we want. We can do this by using the np.full() function. The np.full(shape, constant value) function takes two arguments. The first argument is the shape of the ndarray you want to make and the second is the constant value you want to populate the array with. Let's see an example:

In [13]:

```
print('Reshaped x = \n', x)
print()

# We print information about the reshaped x
print('x has dimensions:', x.shape)
print('x is an object of type:', type(x))
print('The elements in x are of type:', x.dtype)
```

Original x = [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]

Reshaped x =

[[0 1 2 3 4]

[5 6 7 8 9]

[10 11 12 13 14]

[15 16 17 18 19]]

x has dimensions: (4, 5)

x is an object of type: <class 'numpy.ndarray'>

The elements in x are of type: int64

One great feature about NumPy, is that some functions can also be applied as methods. This allows us to apply different functions in sequence in just one line of code. ndarray methods are similar to ndarray attributes in that they are both applied using dot notation (.). Let's see how we can accomplish the same result as in the above example, but in just one line of code:

In [23]:

```
# We create a a rank 1 ndarray with sequential integers from 0 to 19
and
# reshape it to a 4 x 5 array
```

```
[ 7  8  9]]
```

Adding and Deleting elements:

Now, let's take a look at how we can add and delete elements from ndarrays. We can delete elements using the np.delete(ndarray, elements, axis) function. This function deletes the given list of elements from the given ndarray along the specified axis. For rank 1 ndarrays the axis keyword is not required. For rank 2 ndarrays, axis = 0 is used to select rows, and axis = 1 is used to select columns. Let's see some examples:

In [30]:

```
# We create a rank 1 ndarray
x = np.array([1, 2, 3, 4, 5])

# We create a rank 2 ndarray
Y = np.array([[1,2,3],[4,5,6],[7,8,9]])

# We print x
print()
print('Original x = ', x)

# We delete the first and last element of x
x = np.delete(x, [0,4])

# We print x with the first and last element deleted
print()
print('Modified x = ', x)

# We print Y
print()
```

```
print('Original Y = \n', Y)


# We delete the first row of y
w = np.delete(Y, 0, axis=0)


# We delete the first and last column of y
v = np.delete(Y, [0,2], axis=1)


# We print w
print()
print('w = \n', w)


# We print v
print()
print('v = \n', v)
```

Original x = [1 2 3 4 5]

Modified x = [2 3 4]

Original Y =

[[1 2 3]

[4 5 6]

[7 8 9]]

w =

[[4 5 6]

[7 8 9]]

```
# We print x

print()

print('x = ', x)


# We print Y

print()

print('Original Y = \n', Y)


# We append a new row containing 7,8,9 to y

v = np.append(Y, [[7,8,9]], axis=0)


# We append a new column containing 9 and 10 to y

q = np.append(Y,[[9],[10]], axis=1)


# We print v

print()

print('v = \n', v)


# We print q

print()

print('q = \n', q)
```

Original x = [1 2 3 4 5]

x = [1 2 3 4 5 6]

x = [1 2 3 4 5 6 7 8]

```
# We create a rank 1 ndarray
x = np.array([1,2])

# We create a rank 2 ndarray
Y = np.array([[3,4],[5,6]])

# We print x
print()
print('x = ', x)

# We print Y
print()
print('Y = \n', Y)

# We stack x on top of Y
z = np.vstack((x,Y))

# We stack x on the right of Y. We need to reshape x in order to stack
it on the right of Y.
w = np.hstack((Y,x.reshape(2,1)))

# We print z
print()
print('z = \n', z)

# We print w
print()
print('w = \n', w)
```

```
{
 "nbformat": 4,
 "nbformat_minor": 0,
 "metadata": {
  "colab": {
   "name": "Day_6_Numpy_Part_2.ipynb",
   "provenance": []
  },
  "kernelspec": {
   "name": "python3",
   "display_name": "Python 3"
  }
 },
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {
    "id": "KhRg_FoegBcB",
    "colab_type": "text"
   },
   "source": [
    "## **Accessing Elements in ndarays:**\n",
    "Elements can be accessed using indices inside square brackets, [ ]. NumPy allows you to use both positive and negative indices to access elements in the ndarray. Positive indices are used to access elements from the beginning of the array, while negative indices are used to access elements from the end of the array. "
   ]
  },
```

```
    ],
    "execution_count": null,
    "outputs": [
     {
      "output_type": "stream",
      "text": [
       "\n",
       "X = \n",
       " [[1 2 3]\n",
       " [4 5 6]\n",
       " [7 8 9]]\n",
       "\n",
       "This is (0,0) Element in X: 1\n",
       "This is (0,1) Element in X: 2\n",
       "This is (2,2) Element in X: 9\n"
      ],
      "name": "stdout"
     }
    ]
   },
   {
    "cell_type": "markdown",
    "metadata": {
     "id": "TN6nKaJFgbyq",
     "colab_type": "text"
    },
    "source": [
     "Elements in rank 2 ndarrays can be modified in the same way as with rank 1 ndarrays. Let's see an example:"
```

```
"# We print x\n",
"print()\n",
"print('Original x = ', x)\n",
"\n",
"# We append the integer 6 to x\n",
"x = np.append(x, 6)\n",
"\n",
"# We print x\n",
"print()\n",
"print('x = ', x)\n",
"\n",
"# We append the integer 7 and 8 to x\n",
"x = np.append(x, [7,8])\n",
"\n",
"# We print x\n",
"print()\n",
"print('x = ', x)\n",
"\n",
"# We print Y\n",
"print()\n",
"print('Original Y = \\n', Y)\n",
"\n",
"# We append a new row containing 7,8,9 to y\n",
"v = np.append(Y, [[7,8,9]], axis=0)\n",
"\n",
"# We append a new column containing 9 and 10 to y\n",
"q = np.append(Y,[[9],[10]], axis=1)\n",
"\n",
```

```
  "\n",
  "# We insert a column full of 5s between the first and second column of y\n",
  "v = np.insert(Y,1,5, axis=1)\n",
  "\n",
  "# We print w\n",
  "print()\n",
  "print('w = \\n', w)\n",
  "\n",
  "# We print v\n",
  "print()\n",
  "print('v = \\n', v)"
 ],
 "execution_count": null,
 "outputs": [
  {
   "output_type": "stream",
   "text": [
    "\n",
    "Original x =  [1 2 5 6 7]\n",
    "\n",
    "x =  [1 2 3 4 5 6 7]\n",
    "\n",
    "Original Y = \n",
    " [[1 2 3]\n",
    " [7 8 9]]\n",
    "\n",
    "w = \n",
    " [[1 2 3]\n",
```

```
  "print()\n",
  "print('z = \\n', z)\n",
  "\n",
  "# We print w\n",
  "print()\n",
  "print('w = \\n', w)"
 ],
 "execution_count": null,
 "outputs": [
  {
   "output_type": "stream",
   "text": [
    "\n",
    "x =  [1 2]\n",
    "\n",
    "Y = \n",
    " [[3 4]\n",
    " [5 6]]\n",
    "\n",
    "z = \n",
    " [[1 2]\n",
    " [3 4]\n",
    " [5 6]]\n",
    "\n",
    "w = \n",
    " [[3 4 1]\n",
    " [5 6 2]]\n"
   ],
```

"We will now see some examples of how to use the above methods to select different subsets of a rank 2 ndarray.\n"

]

},

{

"cell_type": "code",

"metadata": {

"id": "HP1atdZBg0Tg",

"colab_type": "code",

"colab": {

"base_uri": "https://localhost:8080/"

},

"outputId": "96beec84-9af1-40ba-9f06-bd04c4e12e16"

},

"source": [

"# We create a 4 x 5 ndarray that contains integers from 0 to 19\n",

"X = np.arange(20).reshape(4, 5)\n",

"\n",

"# We print X\n",

"print()\n",

"print('X = \\n', X)\n",

"print()\n",

"\n",

"# We select all the elements that are in the 2nd through 4th rows and in the 3rd to 5th columns\n",

"Z = X[1:4,2:5]\n",

"\n",

"# We print Z\n",

"print('Z = \\n', Z)\n",

```
      "colab_type": "code",
      "colab": {
        "base_uri": "https://localhost:8080/",
        "height": 137
      },
      "outputId": "40e08849-c383-4111-d028-9df536f3f2a1"
    },
    "source": [
      "# We create an unsorted rank 1 ndarray\n",
      "x = np.random.randint(1,11,size=(10,))\n",
      "\n",
      "# We print x\n",
      "print()\n",
      "print('Original x = ', x)\n",
      "\n",
      "# We sort x and print the sorted array using sort as a function.\n",
      "print()\n",
      "print('Sorted x (out of place):', np.sort(x))\n",
      "\n",
      "# When we sort out of place the original array remains intact. To see this we print x again\n",
      "print()\n",
      "print('x after sorting:', x)"
    ],
    "execution_count": 4,
    "outputs": [
      {
        "output_type": "stream",
        "text": [
```

```
      "colab_type": "text"
    },
    "source": [
      "When sorting rank 2 ndarrays, we need to specify to the np.sort() function whether we are sorting by rows or columns. This is done by using the axis keyword. Let's see some examples:\n"
    ]
  },
  {
    "cell_type": "code",
    "metadata": {
      "id": "XdZi81Q4lFv7",
      "colab_type": "code",
      "colab": {
        "base_uri": "https://localhost:8080/"
      },
      "outputId": "a4201ae6-6e75-4632-9e4c-864768dcc73f"
    },
    "source": [
      "# We create an unsorted rank 2 ndarray\n",
      "X = np.random.randint(1,11,size=(5,5))\n",
      "\n",
      "# We print X\n",
      "print()\n",
      "print('Original X = \\n', X)\n",
      "print()\n",
      "\n",
      "# We sort the columns of X and print the sorted array\n",
      "print()\n",
```

```
{
  "cell_type": "markdown",
  "metadata": {
    "id": "sB3FyKXnNokr",
    "colab_type": "text"
  },
  "source": [
    "We can also apply mathematical functions, such as sqrt(x), to all elements of an ndarray at once."
  ]
},
{
  "cell_type": "code",
  "metadata": {
    "id": "gUpizA7QNmKU",
    "colab_type": "code",
    "colab": {
      "base_uri": "https://localhost:8080/",
      "height": 170
    },
    "outputId": "ac45f76b-69e7-4682-d310-e86fbc2f1c43"
  },
  "source": [
    "# We create a rank 1 ndarray\n",
    "x = np.array([1,2,3,4])\n",
    "\n",
    "# We print x\n",
    "print()\n",
    "print('x = ', x)\n",
```

```
},
"source": [
  "# We create a 2 x 2 ndarray\n",
  "X = np.array([[1,2], [3,4]])\n",
  "\n",
  "# We print x\n",
  "print()\n",
  "print('X = \\n', X)\n",
  "print()\n",
  "\n",
  "print('3 * X = \\n', 3 * X)\n",
  "print()\n",
  "print('3 + X = \\n', 3 + X)\n",
  "print()\n",
  "print('X - 3 = \\n', X - 3)\n",
  "print()\n",
  "print('X / 3 = \\n', X / 3)
],
"execution_count": 14,
"outputs": [
  {
    "output_type": "stream",
    "text": [
      "\n",
      "X = \n",
      " [[1 2]\n",
      " [3 4]]\n",
      "\n",
```

"In order to do element-wise operations, NumPy sometimes uses something called Broadcasting. Broadcasting is the term used to describe how NumPy handles element-wise arithmetic operations with ndarrays of different shapes. For example, broadcasting is used implicitly when doing arithmetic operations between scalars and ndarrays.\n",

"\n",

"In the examples above, NumPy is working behind the scenes to broadcast 3 along the ndarray so that they have the same shape. This allows us to add 3 to each element of X with just one line of code.\n",

"\n",

"Subject to certain constraints, Numpy can do the same for two ndarrays of different shapes, as we can see below."

]

},

{

"cell_type": "code",

"metadata": {

"id": "sZdEElKmOIJE",

"colab_type": "code",

"colab": {

"base_uri": "https://localhost:8080/",

"height": 434

},

"outputId": "0ea5fd84-3656-4e88-d073-71d226fb872f"

},

"source": [

"# We create a rank 1 ndarray\n",

"x = np.array([1,2,3])\n",

"\n",

"# We create a 3 x 3 ndarray\n",

"Y = np.array([[1,2,3],[4,5,6],[7,8,9]])\n",