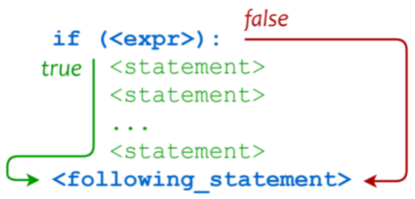
Gear Up

Written and complied by Bowen He

1 Basic Python Usage

1.1 Conditional Control

Use `if-else` branches to execute different code blocks under different Boolean conditions!



The basic usage for this functionality looks like the following:

```
if condition1:
    code block 1
elif condition2:
    code block 2
else:
    code block 3
```

The Python interpreter will check each condition from top to bottom. If condition 1 is satisfied, then code block 1 will be executed, and the following conditions will not be checked. Otherwise, the interpreter will continue to check the conditions one by one, with `elif` standing for "`else if`" in the code above. If none of the conditions can be satisfied, we will execute the code block inside the `else` branch. It serves as a back up case, which why we don't specify any condition there.

1.2 Loops

Use loop operations to execute the same block several times! One thing to note is that, while these loops let you repeat some operations, it doesn't limit you to only repeating the same operations. Some variables can be changed in the process, which should be your real purpose to use them.

1.2.1 For Loops

For loops provide a convenient way to scan over a sequence of elements, each of which can be used in the subsequent code block directly.

You can find the basic usage for this kind of loop as following:

```
for x in [e_1, e_2, ..., e_n]:
    # <code block>
```

For loops will help you to check each of the element in the list. This means that `x` will be assigned different values for each circle. At the mean time, you may find it helpful to use `x` in your code. Check this example.

```
for x in range(10):
    y = (x + 2)**2
    print(y)
```

In this example, `x` will be assigned values from 0 through 9. The code block within the loop will help to compute the function $(x + 2)^2$ and then print out correspondingly. We will get a result like this.

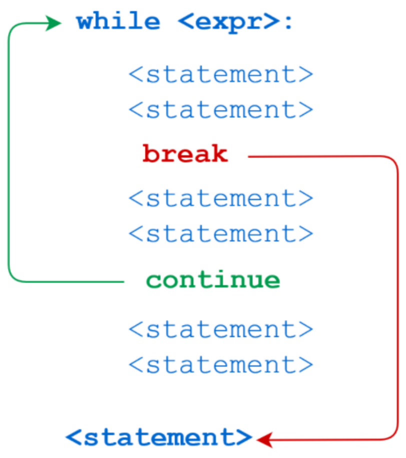
```
4
9
16
25
...
121
```

As you may guess, the list can even contain elements that are not numbers at all. If you have a list containing all the months in a year, like `["January", "February", ..., "December"]`, a for loop will let you consider each element in turn. One last thing to note is a special use for this functionality. If you only care about the number of loops itself, rather than the exact numbers in your list, you can easily use _ to replace `x` there in the code.

1.2.2 While Loops

While loops allow a block be executed over and over again as long as the condition is satisfied.

In the picture below, the code block will be executed while the `expr` evaluates to `True`. In addition to that, we have `break` and `continue` there in the loop as well. As you may guess, `break` gives you the option to jump out of the loop at any time. When combined with `if`, you can terminate the loop under any condition you specify. Unlike `break`, `continue` will bring you back to the head of the loop, which means that the loop can be executed again if the condition is still met.



1.3 Functions

Functions in Python basically name a block of code so that you can call it later instead of typing the same code over and over again. For example, suppose you want to compute the `x`th term of a Fibonacci sequence. You can achieve that by constructing a function with the following code.

```python
def Fibonacci(x):
    number_1 = 0
    number_2 = 1
    i = 1
    while i < x - 1:
        number_1 = number_1 + number_2
        number_2 = number_1 + number_2
        i = i + 2
    if x == i:
        return number_1
    elif x == i + 1:
        return number_2
```

No matter what `x` you have, you can simply call `Fibonacci(x)`, and Python will pass the value to the function, returning the result of the computation.

You can even embed your function into other code structures!

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

As you may have guessed, the result of the code block will be the Fibonacci sequence up to the 9th term.

Now, let's see the formal definition and usage of functions in Python.

In order to define a function, you always need to use the keyword `def` (short for "define") in front of your function name. In addition, you should also specify the arguments you will use within the bracket after your function name. Note that we call these arguments as formal arguments, since they are used as the representatives of the actual arguments that you will pass in later when the function is called somewhere else.

```
def your_function(x, y, z, ...):
    code block 1
    code block 2
    ...
    return a, b, c
```

The `x, y, z, ...` above will have no actual values at this point, they are only used for the purpose of expressing your code logic. When I call the function in another place with real values in the position, namely `your_function(1, 2, 3,...)`, the actual values will be passed to the formal arguments, which then will be used as a part of the computation with corresponding results returned in the end.

2 Useful libraries

In this section, we will introduce the basic usage for the most important library you will use in this course! Python is famous for the richness of libraries it can provide to meet all kinds of needs in your coding life. In AI fields specifically, some of the most useful and actively used libraries may include NumPy (matrix operations), SciPy (machine learning tools), PyTorch (deep learning backbone by Facebook), TensorFlow (deep learning backbone by Google), Pandas (data analysis), along with some visualization tools like Matplotlib and Seaborn. We will not cover all of them here and will focus on NumPy, which we will use the most in this course. Feel free to search them up if you're interested.

2.1 NumPy

To start using NumPy in your code, type `import numpy as np` at the top of your file!

2.1.1 The NumPy Array

The data structure used by NumPy is something called an array, which is an generalized matrix with n dimensions. You may know that we call a one-dimensional number list as vector, a two-dimensional one as matrix. Beyond that, we call any n-dimensional number lists as an array.

One unique property about NumPy is broadcasting. When you do element-wise operations between two arrays of different shapes, broadcasting will replicate some of the elements to make sure that the operations can be executed. For example, for two arrays of the shape `(3, 2)` and `(3, 1)`, NumPy will automatically replicate the column of the second array to convert it into the shape `(3, 2)`, which then can be used to do element-wise computation like matrix summation or element-wise multiplication. The underlying mechanism works as follows: NumPy will start to check the pairs of the dimensions in a manner from right to left. Two situations will be considered as compatible, which are 1) the pair has the same dimensionality 2) one of the dimensions is one. NumPy will replicate elements in the second situation to make the matrices match with each other.

2.1.2 Constructing Arrays

You can use the following methods to construct an array.

1. `np.arange(start, stop, step)`
   Create a one-dimensional NumPy array ranging from `start` to `stop` (exclusive) in increments of `step`.
2. `np.zeros(shape)`
   Create a NumPy array with all 0's in it. The shape will conform to that you give in the inner blanket.
3. `np.ones(shape)`
   Similar to `np.zeros()`, with the only difference being that the elements now become 1's.
4. `np.eye(rows, cols)`
   Create a 2-dimensional NumPy array in shape `(rows, cols)`, with the leading diagonal filled with 1's and all the other places as 0's.
5. `np.array(x)`
   Convert `x` into a NumPy array. `x` is commonly a Python list. For example, with

   ```
   np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
   ```

   you will get

   ```
   [[1, 2, 3, 4],
    [5, 6, 7, 8]]
   ```

2.1.3 Indexing Arrays

Suppose we have a NumPy array `array` as following.

```
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

We have several ways to index elements here.

`array[1, 2]` returns `6`, which is the element located at the second row and the third column. This way of indexing is similar to that you may use in a normal matrix.

`array[-1, 2]` returns `9`. Since `-1` means "the last" in Python, we will get the element at the last row and the third column.

`array[:, :]` returns the array itself. Since "`:`" stands for all elements across a given axis, and we use "`:`" for both the rows and columns, we will index all elements in the array, which is just the array itself.

Just like with a Python list, using "`start:stop`" in place of "`:`" will let us index elements from `start` to `stop` (exclusive) on a given axis. So if we do `array[0:2, :]`, we will get

```
[[1, 2, 3],
 [4, 5, 6]]
```

Similarly, if we use `array[1, 1:]`, we will get

```
[5, 6]
```

2.1.4 Common Operations

1. `array.shape` and `array.reshape()`
   For an existing array `array`, we can access `array.shape` to get its exact shape in the form `(s_1, s_2, ..., s_n)`, where `s_i` is the size of dimension `i`. You can also call `array.reshape()` to convert the array into a totally new shape. Note that the total number of elements before and after reshaping must stay the same. In addition, `np.transpose()` will transpose an array as its name suggests. This operation converts an array with shape `(x, y, z)` into one with shape `(z, y, x)`.

2. element-wise multiplication and matrix multiplication
   Use `np.multiply()` to do element-wise multiplication, while `np.matmul()` can help to do matrix multiplication. As in linear algebra, for matrix multiplication, you have make sure that the second dimension of the first matrix matches the first dimension of the second matrix. Two matrices with shapes like `(3, 2)` and `(2, 4)` can be used for this operation, while `(3, 3)` and `(2, 4)` will be prohibited.

3. summation
   `np.sum()` will execute summation along specific axis you assigned in the arguments. In detail, if we do `np.sum(array, axis=0)`, we sum up all the elements along the first dimension of `array`, by which we will get `[12, 15, 18]` that is the result of 1+4+7, 2+5+8 and 3+6+9 correspondingly. Similarly, `np.sum(array, axis=1)` will get the result as `[6, 15, 24]`. The trick is to find the direction that you use to count the number of rows and columns. For example, since you count the number of rows vertically from top to down, `axis=0` will follow the same direction. And

similarly, counting columns horizontally will accord to that `axis=1` follows the horizontal direction.

3 Basic Linear Algebra

3.1 Matrix

A matrix is an arrangement of numbers in a rectangular form. It takes its origin in linear equations, where people were trying to find a more compact form with all variables omitted and only coefficients reserved. As a result, it's found that this form can not only be used to find the solutions of linear equations, but also be used to represent linear computations themselves, making the matrix a very common tool in STEM fields.

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

The matrix above is called an $m \times n$ matrix, meaning that there are m rows and n columns in the matrix. Each row could represent a linear equation, with n numbers in the row being the coefficients of the linear equation. For now, we don't need to worry about that. Knowing the appearance will be enough for our purposes.

3.2 Linear Operations—Addition, Subtraction, Multiplication, Division

3.2.1 Matrix and Matrix

The first thing to note is that only matrices of the same shape can be added or subtracted. We can do these operations in an element-wise manner for any number of matrices.

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} + \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix}$$
$$= \begin{bmatrix} x_{11}+y_{11} & x_{12}+y_{12} & \cdots & x_{1n}+y_{1n} \\ x_{21}+y_{21} & x_{22}+y_{22} & \cdots & x_{2n}+y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1}+y_{m1} & x_{m2}+y_{m2} & \cdots & x_{mn}+y_{mn} \end{bmatrix}$$

Substituting all $+$'s for $-$, $\times$, or $/$'s in the equation above will yield the other element-wise operations.

3.2.2 Matrix and Scalar

A scalar will be distributed across all elements of a matrix, with regard to either multiplication or division you are doing.

So for example,

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} \times a = \begin{bmatrix} ax_{11} & ax_{12} & \cdots & ax_{1n} \\ ax_{21} & ax_{22} & \cdots & ax_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ ax_{m1} & ax_{m2} & \cdots & ax_{mn} \end{bmatrix}.$$

Mathematically speaking, we usually don't add or subtract a matrix with a scalar. However, since we have the concept of "broadcasting" in NumPy, we allow these two operations to happen in our code. But remember, even though this is allowed, what really happens is that the scalar is repeated to construct a matrix of the same shape as the matrix involved, after which the operations will be executed.

3.3 Matrix Multiplication

Now, imagine we have a bunch of linear expressions as the following.

$$\begin{gathered} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_m \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_m \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_m \end{gathered} \tag{1}$$

One thing to notice is that all of the expressions share the same variables of x_1 through x_n. Do we have a way to simplify the expressions like we can with the distributive property over addition? The answer is yes, and the resulting operation is what we call a matrix multiplication. You may split the expressions like so.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Notice that, x_1 through x_n have been extracted as a column vector (or 1-dimensional matrix), whose number of rows equals the number of columns of the matrix in front. Matrix multiplication takes each row of the first matrix, multiplies it by the column vector in a element-wise manner, and sums the resulting terms, helping us to retrieve the original linear expressions.

The above example is about the situation between one matrix and one vector. How about the case where we want to multiply two matrices together? The answer is to consider it as several sets of linear expressions, each of which is following the same rule we've shown above. So, with

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix}.$$

what we do is apply the previous rule separately to each combination between the first matrix and one of the column vectors of the second matrix, then organize them together into a new matrix.

If we let A represent the matrix with scalars a_{ij}, and $\mathbf{x_j}$ represent column j of the matrix with scalars x_{ij}, then the resulting matrix will be

$$\begin{bmatrix} A \cdot \mathbf{x_1} & A \cdot \mathbf{x_2} & \cdots & A \cdot \mathbf{x_k} \end{bmatrix},$$

where each $A \cdot \mathbf{x_j}$ is a column vector that looks like

$$\begin{bmatrix} a_{11}x_{1j} + a_{12}x_{2j} + \cdots + a_{1n}x_{nj} \\ a_{21}x_{1j} + a_{22}x_{2j} + \cdots + a_{2n}x_{nj} \\ \vdots \\ a_{m1}x_{1j} + a_{m2}x_{2j} + \cdots + a_{mn}x_{nj} \end{bmatrix},$$

just like the list of linear expressions in (1).

With just two matrices, we can express k sets of linear expressions with m expressions and n variables in each of them!

The only requirement for matrix multiplication is that the second dimension of the first matrix matches the first dimension of the second matrix. Namely, the shapes of the matrices involved should be $m \times n$ and $n \times k$, sharing the same number in between. From a linear equation point of view, this constraint can be explained by the fact that we need to make sure that the number of the coefficients is the same as the number of the variables.