Programming for Essential Digital Skills, Part 1

Christoph Walsh

2024

What is Python?

- Python is a *programming language*.
- A programming language is a way to instruct a computer to perform operations via written text.
- When programming, we need to be very exact. Otherwise the computer will throw and error (*syntax error*) or perhaps even do something we didn't intend it to do (*semantic error*)!
- To code Python we will use *Spyder*. This is a desktop application where you can write Python code and execute Python programs.

Why Learn Python?

Python has many benefits over some alternatives:

- It is free and open source.
- It has a large active community creating packages and providing support.
- Easier to learn than some alternatives.
- It is extremely versatile. For example, the online book was made with Python!
- It is also the most common language used in data science.

Installing Python

- To download and install Python and Spyder, go to https://www.anaconda.com/download.
- Follow all the default settings.
- Afterwards, open Spyder.

The IPython Console

- We can perform calculations in the IPython Console tab in Spyder.
- At the most basic level, we can use it as a simple calculator:
  - ▶ Add: 2 + 3
  - ▶ Subtract: 5 - 3
  - ▶ Multiply: 2 * 3
  - ▶ Divide: 3 / 2
  - ▶ Combining operations: (2 + 4) / (4 * 2)
 - ▶ Exponentiation: 2 ** 3

Python Functions

- Just like in Excel, Python has functions.
- These functions work in a very similar way:
  - ▶ Functions have names, and we provide *arguments* to functions inside parentheses.
- In the next few slides we will see some examples of mathematical functions and how to evaluate them in Python.

Absolute Value

- The absolute value function turns negative numbers into positive ones and has no effect on zero or positive numbers.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$
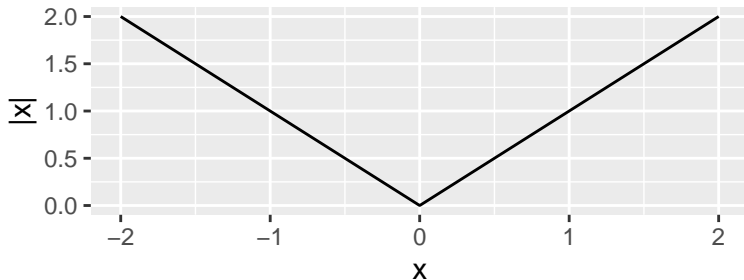
Absolute Value in Python

- The absolute value function in Python is called abs. We can use it as follows:
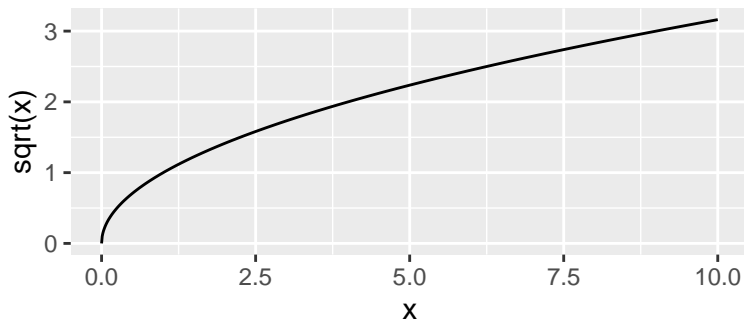
```python
abs(-2)
```

2

```python
abs(3)
```

3

- We can see help about a function using help(abs)

Principal Square Roots

- The square root of a number is the y that solves $y^2 = x$.
- If $x = 4$, both $y = -2$ and $y = 2$ solve $y^2 = x$.
- The principal square root is the positive y solving this.

Principal Square Roots in Python

- Python doesn't have a square root function built in. But we can take a number to the power of $\frac{1}{2}$ to take the square root:

```
9 ** (1/2)
```

```
3.0
```

- The cubed root is the y that solves $y = x^3$. We can also calculate this in Python by taking the power of $\frac{1}{3}$.

```
8 ** (1/3)
```

```
[1] 2
```

The math Module in Python

- Although there is no square root function built in, there is one in the math module.
- Modules are ways we can load more functionality into Python, on top of the built-in Python functions.
- We can load this module and all its functions using import math. We can then using its square root function with:
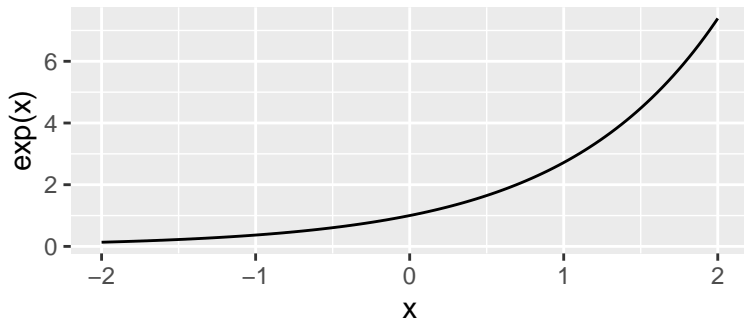
```python
import math
math.sqrt(9)
```

```
3.0
```

- We write math.sqrt() because sqrt() is a function in the module math.

The Exponential Function

The exponential function is a very common function in statistics:

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n$$

The Exponential Function in Python

- In Python we use the `math.exp()` function to calculate the exponential of a number:

```
import math
math.exp(0)
```

```
1.0
```

```
math.exp(1)
```

```
2.718281828459045
```
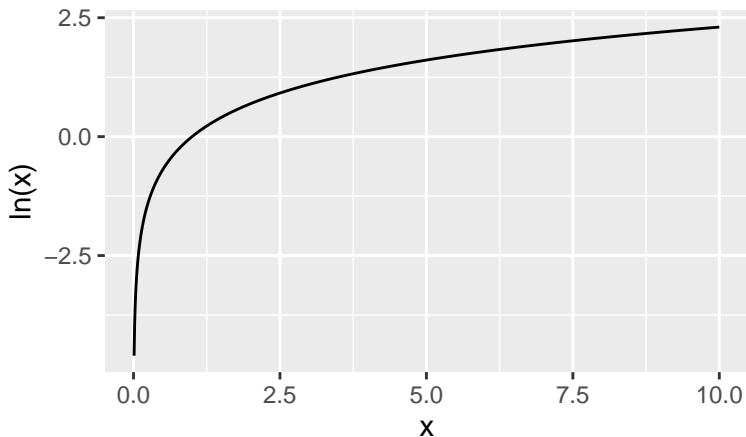
The Logarithm

- Another common mathematical function is the logarithm, which is like the reverse of exponentiation.
- The log of a number x to a base b, denoted $\log_b(x)$, is the number of times we need to multiply b by itself to get x.
- For example, $\log_{10}(100) = 2$, because $10 \times 10 = 100$. We need to multiply the base $b = 10$ by itself twice to get to $x = 100$.

The Natural Logarithm

- A special logarithm is the natural logarithm, $\log_e(x)$, which is the logarithm to the base $\exp(1) = e^1 \approx 2.7183$. This is also written as $\ln(x)$.

The Logarithm in Python

- In Python we use the `math.log()` function to calculate the natural logarithm:

```
math.log(1)
```

```
0.0
```

- We can calculate the logarithm of a number to a different base using the `base` argument. For example, for $\log_{10}(100)$:

```
math.log(100, 10)
```

```
2.0
```

- We didn't provide the base argument to calculate $\log_e(1)$, because e is the default base in the `math.log()` function.

Integer Division and the Modulus Operator

- Dividing $\frac{7}{3}$ can also be expressed as "2 with remainder 1". We can get this in Python with *integer division* (with //) and the *modulus operator* (with %).
- The integer division operator always rounds down to the nearest whole number:

```
7 // 3
```

2

- The modulus operator gives the remainder when dividing two numbers:

```
7 % 3
```

1

- We'll see that the modulus operator is extremely useful in programming.

Assigning Values to Variables

- We can assign *values* to *variables* using *assignment*.
- For example:

```
a = 2
b = 3
```

- a and b are then visible in the **Variable Explorer** tab in Spyder.
- We can then use a and b for calculations:

```
a + b
```

5

- There are rules for naming variables:
  - ▶ They must start with either a letter or _ and contain no spaces.
  - ▶ Variable names cannot be a keyword in Python.

Data Types for Single Values

- Integers (`int`): whole numbers, positive or negative (e.g. 1 or -2).
- Floating-Point Numbers (`float`): real-valued numbers (e.g. 1.234).
  - ▶ Be careful with floating-point precision!
- Strings (`str`): sequences of characters (e.g. `'abc'`).
  - ▶ Addition combines strings: `'abc' + 'def'`
  - ▶ Multiplication by an integer repeats strings: `3 * 'abc'`
- Boolean (`bool`): The Logical constants `True` and `False`.

Logical Operators

There are 3 logical operators: and, or and not:

- a and b is True if both a and b are True. Otherwise it is False.
- a or b is True if either a or b are True. Otherwise it is False.
- not a is True if a is False. Otherwise it is True.

```
a = True
b = False
a and b
```

False

```
a or b
```

True

```
not a
```

False

Comparison Operators

We can also check weak/strict (in)equalities in Python:

```python
a = 2
b = 3
a == b  # is a equal to b?
```

```
False
```

```python
a != b  # is a not equal to b?
```

```
True
```

```python
a > b  # is a strictly greater than b?
```

```
False
```

```python
a <= b  # is a weakly less than b?
```

```
True
```

Type Conversion

We can sometimes convert between `int`, `float`, `str` and `bool`, but we don't always get intuitive results!

```python
float('1.0')
```

```
1.0
```

```python
str(1)
```

```
'1'
```

```python
int(True)
```

```
1
```

```python
bool('False')
```

```
True
```

Assigning Multiple Values to a Variable

- Python has a number of ways you can assign multiple values to a single variable:
  - ▶ Lists
  - ▶ Tuples
  - ▶ Dictionaries
  - ▶ Sets (and Frozen Sets).

Each of these have different properties, which we will learn about.

Lists

Lists can contain any sequence of values, which could be integers, floats, strings, Booleans, or any combination of them. They can even contain lists as elements.

```python
a = [2, 4, 6]
a = [1, 1.1, True, 'hello']
a = [1, 2, [3, 4]]
```

Arithmetic Operations on Lists

+ combines lists and * repeats lists (like with strings):

```
a = [2, 4, 6]
b = [8, 10, 12]
a + b
```

```
[2, 4, 6, 8, 10, 12]
```

```
3 * a
```

```
[2, 4, 6, 2, 4, 6, 2, 4, 6]
```

– and / do not work on lists.

Indexing

- We can extract elements of a list with indexing, which uses the order the elements appear.
- In Python (and many other languages like C), indexing starts at 0.
  - This means the 1st element has index 0, the 2nd element has index 1, and so on.

```
a = [2, 4, 6]
a[0]
```

2

```
a[1]
```

4

```
a[2]
```

6

Mutability of Lists

We can change the value of an element its index:

```
a = [2, 4, 6]
a[0] = 8
a
```

```
[8, 4, 6]
```

Because lists have this property we say they are *mutable*.

List Slicing

To get a range of elements in a list we can using *slicing*:

```python
a = [1, 2, 3, 4, 5]
a[1:3]
```

```
[2, 3]
```

```python
a[:2]
```

```
[1, 2]
```

```python
a[2:]
```

```
[3, 4, 5]
```

```python
a[:]
```

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

List Methods

Lists have *methods*, which are like functions but instead of providing the list as an argument to the function, we apply the method directly to the object.

We can add an element at the end of a list using the append( ) method:

```python
a = [2, 4, 6]
a.append(8)
a
```

```
[2, 4, 6, 8]
```

Notice how the method changes a *in place*.

List Methods

Removing an element (by index):

```python
a = [2, 4, 6]
a.pop(1)
```

```
4
```

```python
a
```

```
[2, 6]
```

Sorting a list:

```python
a = [6, 2, 4]
a.sort()
a
```

```
[2, 4, 6]
```

Iterating over Items in a List

- A useful thing we can do with lists is iterate over each element and perform an operation on those elements.
- For example, if we want to square each number:

```python
a = [2, 4, 6]
for i in a:
    print(i ** 2)
```

```
4
16
36
```

Indentation Matters!

The following two code snippets only differ by indentation, but produce different output:

Snippet 1:

```
for i in a:
    print(i ** 2)
    print('hello')
```

Snippet 2:

```
for i in a:
    print(i ** 2)
print('hello')
```

print('hello') is part of the loop in case 1, and after the loop in case 2.

For Loops

If we want to save each squared number we could do:

```
a = [2, 4, 6]
b = []
for i in a:
  b.append(i ** 2)
b
```

```
[4, 16, 36]
```

List Comprehensions

However, an easier way to do this is to use *list comprehensions*:

```
a = [2, 4, 6]
b = [i ** 2 for i in a]
b
```

```
[4, 16, 36]
```

List Membership

We can check if a value is contained somewhere in a list using the `in` operator:

```
a = [2, 4, 6]
4 in a
```

```
True
```

```
5 in a
```

```
False
```

Copying Lists

Consider the following:

```
a = [2, 4, 6]
b = a
a[0] = 8
b
```

[8, 4, 6]

A change in a automatically changes b!

Copying Lists

To avoid this we can do:

```python
a = [2, 4, 6]
b = a[:]
a[0] = 8
b

[2, 4, 6]
```

Two Types of Copying

- *Deep copy:* This copies a to b and recursively copies all of its elements, resulting in a completely independent object.
- *Shallow copy:* This copies a to b but does not recursively copy its elements. Instead, it only copies the references to the elements in a (like the address for where in the computer's memory those elements are stored). This means that changes to elements of a will affect the elements of b.

Tuples

- A tuple is like a list with the important difference that they are *immutable*. Once we create them we cannot change them.
- We can create tuples with parentheses instead of square brackets:

```
a = (2, 4, 6)
```

- Indexing, iterating and other operations for lists also work with tuples.
- However, appending, popping, or altering values is not possible.

Tuple Assignment

Suppose we wanted to do the following:

```
x = ['a', 'b', 'c']
x_0 = x[0]
x_1 = x[1]
x_2 = x[2]
```

We can shorten this code by using *tuple assignment:*

```
(x_0, x_1, x_2) = x
```

Dictionaries

- A dictionary is a collection that maps *keys* to *values*.
- The keys can be strings or integers, and the values can be any type.

```
suppliers = {100001 : 'ABC Ltd.', 100002 : 'EFG Ltd.'}
```

Extracting Values from a Dictionary

- Dictionaries are unordered, so we cannot extract elements with indexing, e.g. with `suppliers[0]`.
- Instead we extract elements using the keys:

```
suppliers[100001]
```

```
'ABC Ltd.'
```

Adding or Modifying Values to a Dictionary

We can add keys and values to a dictionary:

```python
suppliers[100003] = 'HIJ Ltd.'
```

We can also modify values:

```python
suppliers[100003] = 'KLM Ltd.'
```

Getting all Keys or Values

We can get all the keys or values with:

```
suppliers.keys()
```

```
dict_keys([100001, 100002, 100003])
```

```
suppliers.values()
```

```
dict_values(['ABC Ltd.', 'EFG Ltd.', 'KLM Ltd.'])
```

These can be converted to a list. For example:

```
list(suppliers.keys())
```

```
[100001, 100002, 100003]
```

Iterating Over Keys

Iterating over a dictionary implicitly iterates over the keys:

```
for key in suppliers:
    print('Supplier with ID ' + str(key) + ' is ' + suppliers[key])
```

```
Supplier with ID 100001 is ABC Ltd.
Supplier with ID 100002 is EFG Ltd.
Supplier with ID 100003 is KLM Ltd.
```

Sets

- A set is another way to store multiple items into a single variable.
- Sets are unordered and unindexed. This means you cannot extract individual elements using their index like a list, nor by their key like a dictionary.
- We create sets using {}:

```
myset = {"apple", "banana", "cherry"}
myset
```

```
{'apple', 'cherry', 'banana'}
```

Sets

Duplicates are not possible:

```
myset = {"apple", "banana", "cherry", "cherry"}
myset
```

```
{'apple', 'cherry', 'banana'}
```

But adding elements is:

```
myset = {"apple", "banana", "cherry"}
myset.add("pear")
myset
```

```
{'apple', 'pear', 'cherry', 'banana'}
```

We can remove with myset.remove("apple").

Sets

We can use sets to get all the unique elements in a list:

```
fruits = ["apple", "apple", "apple", "banana", "cherry", "banana"]
set(fruits)
```

```
{'apple', 'cherry', 'banana'}
```

We can also iterate over sets just like with lists and tuples.

Set Operations

We can also do mathematical set operations like ∩ (intersection), ∪ (union) and set differences:

```python
set_a = {1, 2, 4, 6, 8, 9}
set_b = {2, 3, 5, 7, 8}
set_a.intersection(set_b)
```

```
{8, 2}
```

```python
set_a.union(set_b)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```python
set_a.difference(set_b)
```

```
{1, 4, 6, 9}
```

Frozen Sets

You want to make an immutable set, you can use the frozenset() command:

```python
myset = frozenset([1, 2, 3])
```

Looking back at the objects we've seen, we can summarize:

- list: mutable and ordered.
- tuple: immutable and ordered.
- set: mutable and unordered.
- frozenset: immutable and unordered.

Defining Functions

If we wanted to create our own function that adds one to its input, i.e.:

$$f(x) = x + 1$$

We can do:

```python
def addone(x):
    y = x + 1
    return y
addone(2)
```

3

Again, indentation matters!

Local vs Global Variables

- The y that we create inside the function is not available outside it.
- If we were to run the code below, we would get an error saying y is not defined.
- The y is only *local* to the function, and is not available globally.

```
x = 2
def addone(x):
    y = x + 1
    return y
addone(2)
y
```

Local vs Global Variables

- If we wanted to access y outside the function, we could do the following.
- Generally this is not recommended practice, but it can be useful for debugging.

```python
x = 2
def addone(x):
    global y
    y = x + 1
    return y
addone(2)
```

3

```python
y
```

3

If-Else Statements

- Suppose we wanted to create our own abs() function from scratch:

$$|x| = \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

- We can use an if-else statement to do this:

```python
def my_abs(x):
    if x < 0:
        return -x
    else:
        return x
[my_abs(i) for i in [-2, 0, 3]]
```

```
[2, 0, 3]
```

If-Else If-Else Statements

- Now let's make the $sgn(x)$ function which gives the sign in front of a number:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{otherwise} \end{cases}$$

```python
def sign(x):
    if x < 0:
        return -1
    else:
        if x == 0:
            return 0
        else:
            return 1
[sign(i) for i in [-2, 0, 3]]
```

```
[-1, 0, 1]
```

If-Else If-Else Statements

- Instead of nesting multiple if-else statements, we can use `elif`:

```python
def sign(x):
    if x < 0:
        return -1
    elif x == 0:
        return 0
    else:
        return 1
[sign(i) for i in [-2, 0, 3]]
```

```
[-1, 0, 1]
```

While Loops

- When you want to repeat commands for an indeterminate amount of time we can use a `while` loop.
- A while loop repeats a command as long as a condition is `True`:

```python
counter = 0
while counter < 3:
    print('hello')
    counter += 1
```

```
hello
hello
hello
```

- Using `while True:` repeats the commands forever.

Using While Loops to Approximate

- The Ancient Greeks used the following method to find the square root of a number.
- They began with an initial guess $y_0 = \frac{x}{2}$ for $\sqrt{x}$.
- They then updated the guess repeatedly where the nth guess is:

$$y_n = \frac{1}{2}\left(y_{n-1} + \frac{x}{y_{n-1}}\right)$$

- For $x = 2$ (where $\sqrt{2} = 1.414214$), this approach yields:

$$(y_0, y_1, y_2, y_3) = (1, 1.5, 1.416667, 1.414216)$$

- It converges very quickly to very close to the solution!
- We can use a while loop to approximate square roots, where we let the loop run until the answer stops changing very much.

Using While Loops to Approximate: The Square Root

```python
def my_sqrt(x, tol=0.000001):
    y = x / 2                          # Initial guess
    dist = tol + 1                     # Initialize distance
    while dist > tol:                  # Repeat until dist <= tol
        y_old = y                      # Previous guess
        y = (y_old + x / y_old) / 2    # Update guess
        dist = abs(y - y_old)          # Distance from previous guess
    return y
my_sqrt(2, 0.1)   # Loose tolerance
```

1.4166666666666665

```python
my_sqrt(2)          # Default (tighter) tolerance
```

1.414213562373095

Numpy

- The built-in Python functions are not well suited to performing many mathematical operations we often need to do.
- The operations can be done, but often require several lines of code or complicated list comprehensions.
- The numpy module contains many functions that can easily do these common operations.
- The conventional way to load numpy is with:

```python
import numpy as np
```

- We will do some examples to show how useful numpy is.

Doubling All Numbers in a List

```python
x = [2, 4, 8]
y = []                  # For loop approach
for i in x:
    y.append(2 * i)
y
```

```
[4, 8, 16]
```

```python
[2 * i for i in x]    # List comprehension approach
```

```
[4, 8, 16]
```

```python
2 * np.array(x)       # Numpy approach
```

```
array([ 4,  8, 16])
```

Element-By-Element Multiplication

Suppose we have two lists of equal length:

$$x = (x_1, x_2, \dots, x_n)$$
$$y = (x_1, x_2, \dots, x_n)$$

and we want to calculate z from this which is:

$$z = (x_1 \times y_2, x_2 \times y_2, \dots, x_n \times y_n)$$

Element-By-Element Multiplication

```python
x = [2, 4, 8]
y = [3, 2, 2]
z = []                          # Iterating over the indices of x and
for i in range(len(x)):
    z.append(x[i] * y[i])
z
```

```
[6, 8, 16]
```

```python
z = []
for i, j in zip(x, y):          # Iterating over x and y with zip
    z.append(i * j)
z
```

```
[6, 8, 16]
```

Element-By-Element Multiplication

```
# List comprehension approach:
[i * j for i, j in zip(x, y)]
```

```
[6, 8, 16]
```

```
# Numpy approach:
x = np.array(x)
y = np.array(y)
x * y
```

```
array([ 6,  8, 16])
```

The Median of a List of Numbers

- When you sort a list of n numbers, the median is:
  - ▶ The number in the middle when n is odd
  - ▶ The average of the two middle numbers when n is even

```python
def median(x):
    y = sorted(x)
    if len(y) % 2 == 0:
        return (y[len(y) // 2 - 1] + y[len(y) // 2]) / 2
    else:
        return y[len(y) // 2]
median([2, 6, 4])
```

4

```python
median([2, 6, 4, 3])
```

3.5

The Median of a List of Numbers

In numpy, we can do this much more easily:

```python
x = np.array([2, 6, 4])
np.median(x)
```

```
4.0
```

```python
x = [2, 6, 4, 3]
np.median(x)
```

```
3.5
```

Matrix Multiplication

- Suppose we want to calculate $C = AB$, where:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 3 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 1 & 2 \\ 3 & 2 & 1 \\ 1 & 3 & 1 \end{pmatrix}$$

- Row i and column j of C is given by:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} b_{kj}$$

For example:

$$c_{21} = \sum_{k=1}^{3} a_{2k} b_{k1} = a_{21} b_{11} + a_{22} b_{22} + a_{23} b_{33} = 2 \times 2 + 3 \times 3 + 1 \times 1 = 14$$

Defining Matrices with Lists:

- We can define matrices as nested lists.
- Each element of the list is a list which represents a row of the matrix.

```
A = [
    [1, 2, 3],
    [2, 3, 1],
    [3, 1, 3]
]

B = [
    [2, 1, 2],
    [3, 2, 1],
    [1, 3, 1]
]
```

Matrix Multiplication without Numpy

```python
C = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]
]
for i in range(3):
    for j in range(3):
        for k in range(3):
            C[i][j] += A[i][k] * B[k][j]
for row in C:
    print(row)
```

```
[11, 14, 7]
[14, 11, 8]
[12, 14, 10]
```

Matrix Mutliplication with Numpy

```
import numpy as np
A = np.array(A)
B = np.array(B)
np.dot(A, B)
```

```
array([[11, 14,  7],
       [14, 11,  8],
       [12, 14, 10]])
```