

BIOS-584 Python Programming (Non-Bios Student)

Week 03

Instructor: Tianwen Ma, Ph.D.

Department of Biostatistics and Bioinformatics,
Rollins School of Public Health,
Emory University

Lecture Overview

- [Clarify on pd.describe\(\)](#) (3-5)
- [Recap of Matplotlib](#) (6-7)
- [NumPy: Numerical Python](#) (8-22)
- [Random numbers with Python \(*random*\)](#) (23-30)
- [Boolean variables](#) (31-48)
- [Control flow with If/elif/else statements](#) (49-56)

pd.describe() function

- The `pd.describe()` function provides summary statistics of continuous variables.
- It also provides a summary of categorical variables.
- In terms of string or texts, it automatically excludes variables with text values. But you can specify to output categorical variables.

pd.describe() function

```
carfeatures.describe()
```

	mpg	cylinders	displacement	weight	acceleration
count	398.000000	398.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.427136	2970.424623	15.568090
std	7.815984	1.701004	104.268683	846.841774	2.757689
min	9.000000	3.000000	68.000000	1613.000000	8.000000
25%	17.500000	4.000000	104.250000	2223.750000	13.825000
50%	23.000000	4.000000	148.500000	2803.500000	15.500000
75%	29.000000	8.000000	262.000000	3608.000000	17.175000
max	46.600000	8.000000	455.000000	5140.000000	24.800000

```
carfeatures['cylinders_cat'] = carfeatures['cylinders'].astype('str')
carfeatures.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	vehicle id	cylinders_cat
0	18.0	8	307	130	3504	12.0	C-1689780	8
1	15.0	8	350	165	3693	11.5	B-1689791	8
2	18.0	8	318	150	3436	11.0	P-1689802	8
3	16.0	8	304	150	3433	12.0	A-1689813	8
4	17.0	8	302	140	3449	10.5	F-1689824	8

Only display results of continuous variables

Add a new column of type string and rename it to “cylinders_cat”

pd.describe() function

```
carfeatures['cylinders_cat'].describe()
```

```
count      398
unique       5
top         4
freq       204
Name: cylinders_cat, dtype: object
```

Not very informative, recommend using [pd.crosstab\(\)](#) to see the entire frequency table for categorical variables.

- The outputs include
 - Total count
 - Number of unique elements
 - The content of the most common element
 - The frequency of the most common element
 - Name and the type.

Visualization with matplotlib

- Use *Matplotlib* to create plots
- The `hist()` function creates a histogram
- Pass a list as an argument to the function
- Customize the plot
 - See additional notes on week-02 module

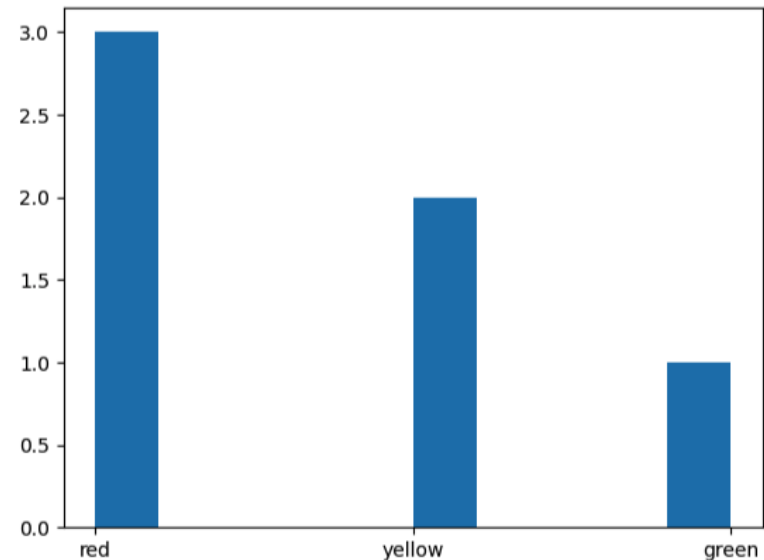
Remember to use `show()` function in the end.

```
import matplotlib.pyplot as plt  
list_2 = ["red", "yellow", "yellow", "green", "red", "red"]
```

```
print(list_2)
```

```
['red', 'yellow', 'yellow', 'green', 'red', 'red']
```

```
# This creates a histogram with the "list_colors"  
plt.hist(x = list_2)  
plt.show()
```



Questions?

NumPy

- Short for “Numerical Python”, a library that provides support for large, multi-dimensional arrays and matrices
- An array is a collection of numbers that are arranged in a regular grid (vector, matrix, high-dimensional array – tensors)
- One of the most important and fundamental libraries in Python

NumPy

- *NumPy* is the backbone of many other libraries in Python, such as *Pandas*, *scikit-learn*
- In general, *NumPy* is already installed when you create a new project using PyCharm.
 - Double check! Sometimes, the default version is a little behind.
 - If not, recall the steps to install or upgrade the packages.
- The alias for *NumPy* is `np`. You can write in Python: `import numpy as np`

Import and basic operations

- Always import packages first!
- Basic operations such as log, exp, sin, cos, $\sqrt{\cdot}$.
 - Shortcut of exponentiation is `**`
- A list of NumPy functions [here](#).

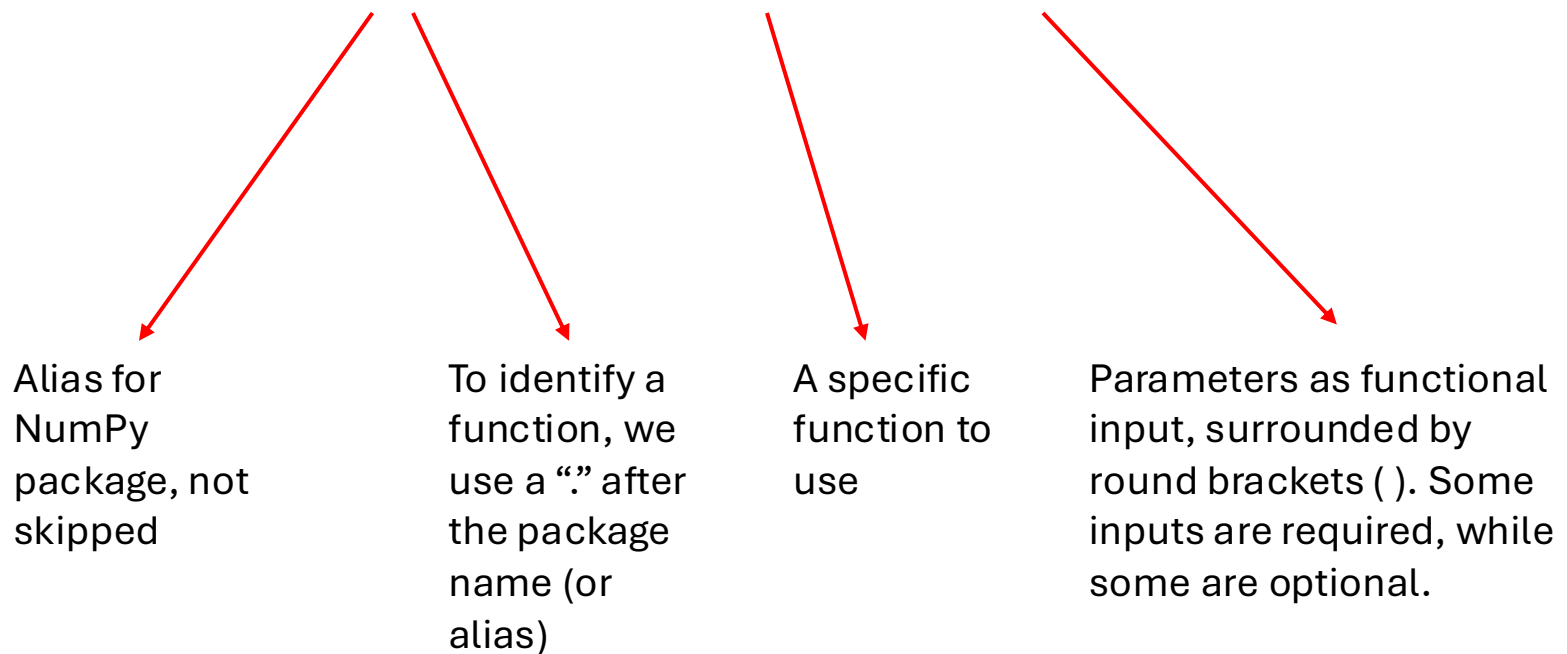
```
# log(x) computes the logarithm with base "e" (Euler constant)
# exp(x) compute the Euler constant raised to the power of "x"
# sin(x) computes the sine of x
# cos(x) computes the cosine of x
# In this example, we're substituting x = 2
```

```
print(np.log(1))
print(np.exp(1))
print(np.sin(1))
print(np.cos(1))
print(np.sqrt(2))
```

```
0.0
2.718281828459045
0.8414709848078965
0.5403023058681398
1.4142135623730951
```

Call and use a function in general

`np.function_name(input1, input2)`



Alias for
NumPy
package, not
skipped

To identify a
function, we
use a “.” after
the package
name (or
alias)

A specific
function to
use

Parameters as functional
input, surrounded by
round brackets (). Some
inputs are required, while
some are optional.

If you have questions, always check their API for detailed tutorials.

Vector arrays with NumPy

- Creating arrays from lists
- Accessing an element of an array
- Operations with a single array and a scalar
- Element-by-element addition between two arrays of the same size (element-wise operations)
- Summary statistics

Creating arrays from lists

- *NumPy* arrays are created using the `np.array()` function.
- We can create arrays from lists
- We can create arrays with a sequence of numbers using `np.arange()` (Yes, only one “r”)
- We can create zero or one arrays using `np.zeros()` and `np.ones()` (Plural forms!)
- `np.eye()` for identity matrix

Examples

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.zeros(3)
```

```
array([0., 0., 0.])
```

```
np.ones(5)
```

```
array([1., 1., 1., 1., 1.])
```

```
np.zeros([2,2])
```

```
array([[0., 0.],  
       [0., 0.]])
```

```
np.ones([3,2])
```

```
array([[1., 1.],  
       [1., 1.],  
       [1., 1.]])
```

```
np.eye(3)
```

```
array([[1., 0., 0.],  
       [0., 1., 0.],  
       [0., 0., 1.]])
```

Creating arrays from lists

- $a = (1, 2, 3)^T$
- $b = (0, 1, 0)^T$
- $(\cdot)^T$ is transpose.

Try

- $c = (1, 10, 10^2, 10^3, 10^4)^T$
- $d = (4, 2)^T$

```
vec_a = np.array([1,2,3])  
vec_b = np.array([0,1,0])  
print(vec_a)  
print(vec_b)
```

```
[1 2 3]  
[0 1 0]
```

- The square brackets `[]` are required because we create an array from a list.
- The round brackets `()` are required because it is an input of an NumPy function.

Accessing an element of an array

- We can access elements of an array using square brackets `[]`
 - Similar to accessing elements of a list
- The index starts from 0.
- Access the first and the third element of a

```
print(vec_a[0])  
print(vec_a[2])
```

```
1  
3
```


Operations with a single array and a scalar

- A scalar refers to either an int or float
- We can add or multiply a scalar to an array

- $a + 3 = \begin{pmatrix} a_1 + 3 \\ a_2 + 3 \\ a_3 + 3 \end{pmatrix}$

- What if 3 becomes an array?

```
print(vec_a)
```

```
[1 2 3]
```

```
print(vec_a * 3)
```

```
print(vec_a / 3)
```

```
print(vec_a + 3)
```

```
print(vec_a ** 3)
```

```
[3 6 9]
```

```
[0.33333333 0.66666667 1. ]
```

```
[4 5 6]
```

```
[ 1  8 27]
```

Element-wise addition

- Two arrays have the same size

$$\bullet \ a + b = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix}$$

```
print(vec_a)
print(vec_b)
```

```
[1 2 3]
[0 1 0]
```

```
# When you add two arrays of the same size,
# Python adds the individual elements
# in each position
```

```
print(vec_a + vec_b)
```

```
[1 3 3]
```

Check the dimension!

```
# it will yield an error!  
vec_c = np.array([1, 10, 100, 1000, 10000])  
print(vec_a + vec_c)
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[12], line 3  
      1 # it will yield an error!  
      2 vec_c = np.array([1, 10, 100, 1000, 10000])  
----> 3 print(vec_a + vec_c)  
  
ValueError: operands could not be broadcast together with shapes (3,) (5,)
```

Other element-wise operations

- $a - b = \begin{pmatrix} a_1 - b_1 \\ a_2 - b_2 \\ a_3 - b_3 \end{pmatrix}$
- $a * b = \begin{pmatrix} a_1 * b_1 \\ a_2 * b_2 \\ a_3 * b_3 \end{pmatrix}$
- $a / b = \begin{pmatrix} a_1 / b_1 \\ a_2 / b_2 \\ a_3 / b_3 \end{pmatrix}$
 - b_1, b_2, b_3 are not zero.

```
print(vec_a - vec_b)
print(vec_a * vec_b)
print(vec_a / vec_b)
```

```
[1 1 3]
[0 2 0]
[inf 2. inf]
```

For the division,
Python will yield a warning.

Summary Statistics

- NumPy provides useful functions to compute summary statistics of an array
- Mean, median, standard deviation, variance, minimum, maximum, quantile
- Sum, product, and cumulative sum
- ...

```
print(vec_a)
print(np.mean(vec_a))
print(np.std(vec_a))
print(np.min(vec_a))
print(np.median(vec_a))
print(np.max(vec_a))
print(np.cumsum(vec_a))
print(np.cumprod(vec_a))
```

```
[1 2 3]
2.0
0.816496580927726
1
2.0
3
[1 3 6]
[1 2 6]
```

Questions?

Random numbers with Python

- The random module is part of the Python Standard Library, no need to install it.
- `import random` It does not have an alias.

Why do we need randomness?

- Simulate different scenarios: high risk vs low risk; different patient subpopulation
- Study proportions of a complex system and/or estimator
- Randomly assign subjects to treatment or control in clinical trials
- Simulate stock prices in finance
- Simulate outcomes of games in sports
- ...

Example Code

- This code creates a vector of random variables generated from a univariate Gaussian distribution
- The distribution has the mean “loc” and standard deviation “scale” parameters
- The length of the vector is specified by “size”

```
randomvar_a = np.random.normal(loc=0, scale=1, size=10)  
print(randomvar_a)
```

```
[ 1.65320155 -1.57004811  0.73274065  1.32487642 -1.47358059  1.10615628  
 -0.53474472 -2.04801587  0.16418893 -0.09359722]
```

Random numbers can be different

- If nothing is specified, the outputs differ each time.
- Add a seed to make it reproducible and the numbers are drawn from a “pre-generated” set.

```
np.random.seed(612)
```

```
random_var_b = np.random.normal(loc=0, scale=1, size=10)
```

```
print(random_var_b)
```

```
[-0.01337706 -1.16289877 -0.22487308  1.11562916  0.5083097  -0.1479853  
 0.2678837  -0.67999971 -0.29333966 -0.38372966]
```

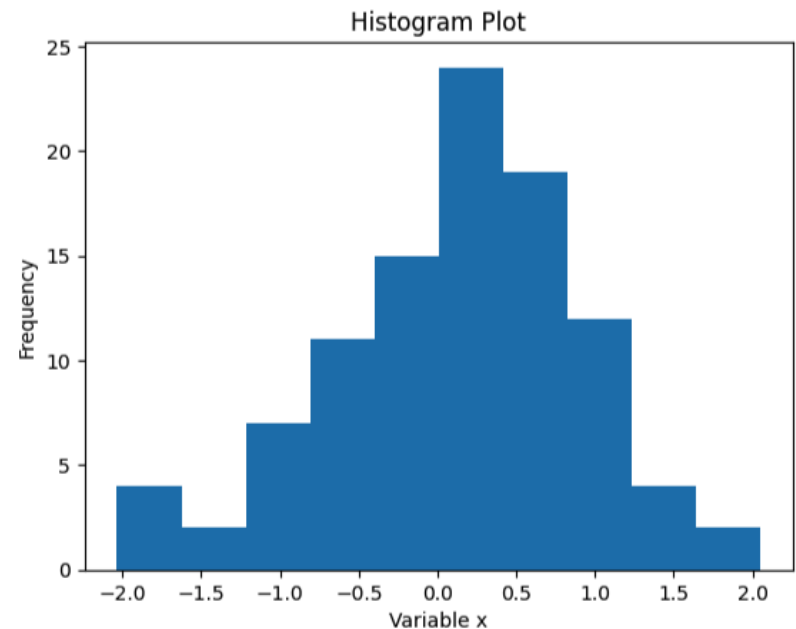
- If you repeatedly running the code chunk, the output of “random_var_b” does not change.

Histogram

- We can use the `plt.hist()` to compute a histogram
- Recall how to specify labels of x-, y-axes, and title.
- Compute mean and standard deviation of the generated vector and compare it to the true value.

```
# Compute a histogram
np.random.seed(612)
randomvar_x = np.random.normal(loc=0, scale=1, size=100)

plt.hist(x = randomvar_x)
plt.xlabel("Variable x")
plt.ylabel("Frequency")
plt.title("Histogram Plot")
plt.show()
```



```
mean_val = np.mean(randomvar_x)
sd_val = np.std(randomvar_x)
print(mean_val)
print(sd_val)
```

```
0.09808633464506501
0.8071588837565997
```

Try it with larger samples

- Change size to 500, 1000, and 5000
- Compute the summary statistics again
- What do you find?

Try it with larger samples

- Change size to 500, 1000, and 5000
- Compute the summary statistics again
- What do you find?
- When sample size is larger, the sample means and sample standard deviations are closer to the true values.
 - The parameter estimation gets more precise.

Questions?

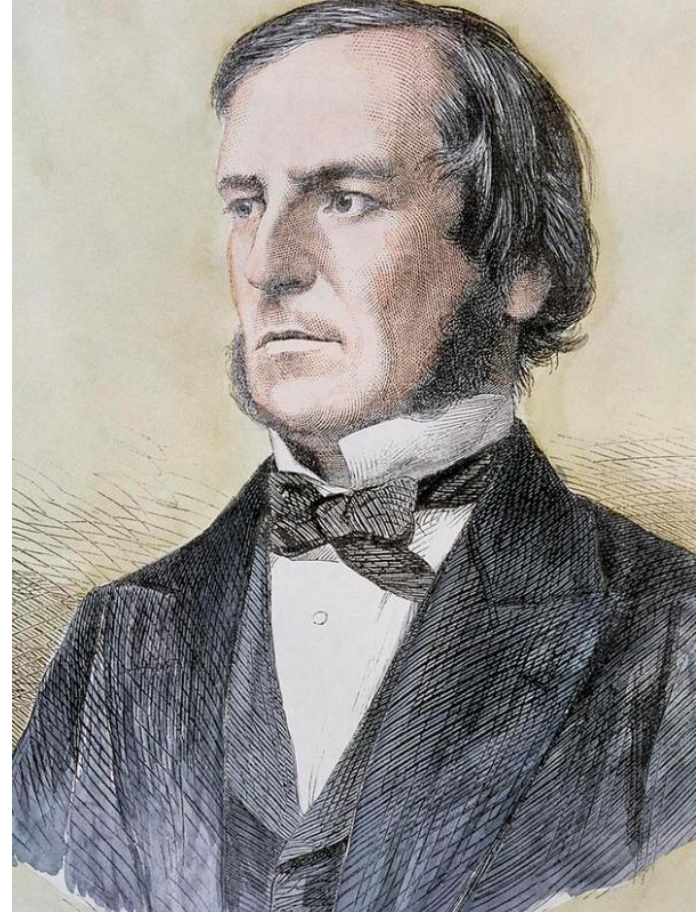
- Practice week-03-maths-arrays-random.ipynb

Boolean Logic and Conditional Statement

- Learn about Boolean logic
- Learn conditional statement in programming
- Learn how to implement conditional statement in Python
 - if
 - elif
 - else
- Practice writing conditional statements in Python

Boolean logic

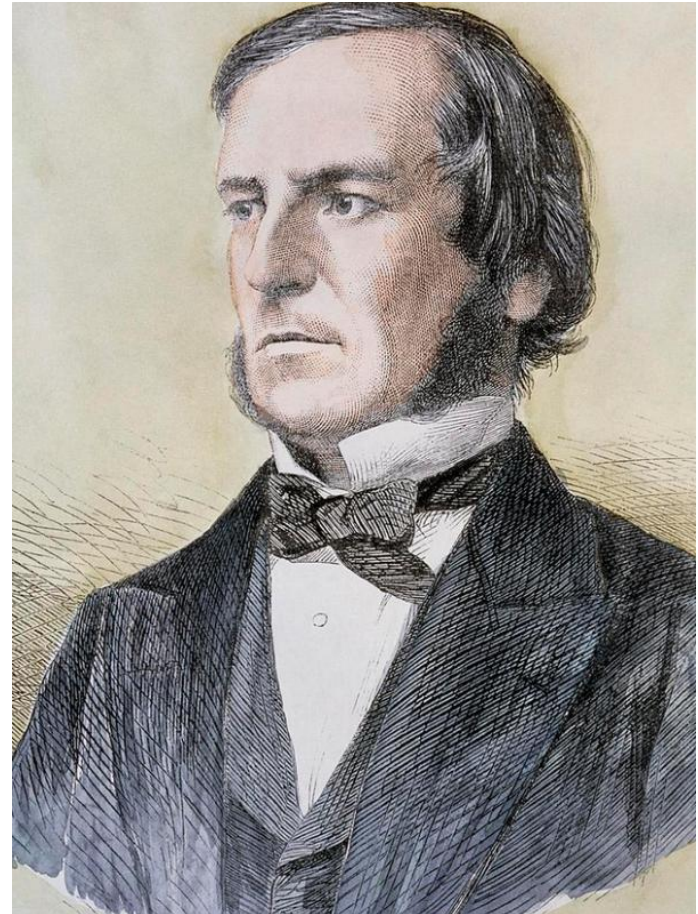
- Named after George Boole, a British mathematician and philosopher
- Boole's work on logic laid foundations for modern computer science
- Boolean logic is a branch of algebra dealing with true and false values



<https://www.elephantlearning.com/post/george-boole-the-father-of-binary-logic>

Boolean logic

- Useful for computer programming b/c it is based on *binary* values
- In Python, Boolean values are **True** and **False**.
 - Reserved keywords
- Use them to make decisions in the codes



Testing Boolean Expressions

- Use “==” operator to test whether two values are equal
- Use “!=” operator to test whether two values are not equal
 - In new PyCharm versions, “!=” will be displayed as “≠”

```
"Is this the real life?" == "is this just fantasy?"
```

False

```
# The order of strings matters
```

```
'ab' == 'ba'
```

False

```
'ab' != 'ba'
```

True

```
Python Console In [2]: 'ab' ≠ 'ba'
```

```
Out[2]: True
```

Testing Boolean Expressions

- Equality of strings is most useful when we compare an unknown variable to a benchmark.
- For example:

```
# Equality of strings is most useful when you're comparing an unknown variable  
# to a benchmark.
```

```
# Below, try switching the value of "any_questions"
```

```
any_questions = "no"  
print( any_questions == "no" )
```

True

- Try changing the values of “[any_questions](#)” to something else.

Testing Boolean Expressions

- We can test whether a keyword is present in a sentence or a list using the “in” operator
- For example, “apple” in “I like apples” return True

```
"apple" in "I like apples"
```

True

```
# The first way to use the "in" command is to check whether a word is contained  
# in a sentence. This can be useful if you're trying to search for patterns
```

```
keyword = "economic"  
sentence = "The Federal Reserve makes forecasts about many economic outcomes"  
keyword in sentence  
# Try changing the keyword!
```

True

Testing Boolean Expressions

- We can test whether a keyword is present in a list

```
current_month = "September"
list_summer_months = ["June", "July", "August"]

print(current_month in list_summer_months)
print('June' in list_summer_months)
```

False

True

Common Pitfalls

- Be careful when testing expressions with texts

- Python is case-sensitive:

```
"apple" == "Apple"
```

False

- To avoid this, you can use `lower()` method to convert all characters to lowercase
- Single equal sign “=” is DIFFERENT from double equal sign “==”

```
"apple".lower() == "Apple".lower()
```

True

```
# A single vs double equality sign makes a BIG difference  
# When you have a single equality (=) you are assignning the variable  
# When you have a double equality (==) you are comparing two values  
message_hello = "hello"  
print( message_hello == "hello" )
```

True

Testing expressions with numbers

- Strictly less than (<)
- Less than or equal (<=)
- Equal (==)
- Strictly more than (>)
- Greater than or equal to (>=)
- Not equal (!=)

```
# We can check equality and inequality constraints  
# Try changing $x$ and see what happens!  
x = 5  
print(x < 5)  
print(x <= 5)  
print(x == 5)  
print(x >= 5)  
print(x > 5)  
print(x != 4)
```

```
False  
True  
True  
True  
False  
True
```

Validate a data type

- We can test whether a variable is of a certain data type using `isinstance()` function
- Common data types include `int`, `float`, `str`, `list`, `tuple`, `dict`, `set`, `bool`, ...
- Specific data types with advanced packages

```
# The isinstance command
y = 10
print(isinstance(y,int))
print(isinstance(y,float))
print(isinstance(y,str))
# Okay, but not recommended.
# In older Python version, it may yield an error.
print(type(y) == int)
print(type(y) == 'int')
```

```
True
False
False
True
False
```


Equality of vectors

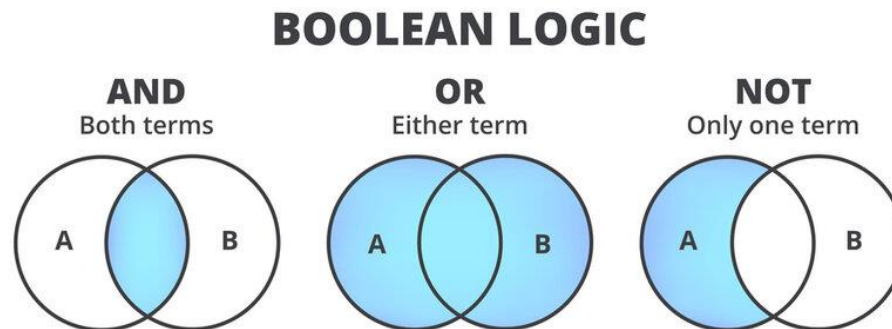
- We can test whether two vectors are equal using “`==`” operator
- The equality of vectors is **done element-wise**.
- `np.any()` or `np.all()` functions may be helpful.

```
vec_a = np.array([1,2,3])
vec_b = np.array([1,2,4])
vec_eval = (vec_a == vec_b)
# brackets are optional but recommended!
print(vec_eval)
print(np.any(vec_eval))
print(np.all(vec_eval))
```

```
[ True  True False]
True
False
```

Testing multiple conditions

- We can test multiple conditions using the **and** and **or** operators
- The **and** operator returns **True** if both conditions are **True**
- The **or** operator returns **True** if at least one conditions is **True**



not: the negation operator

- We can negate a condition using the **not** operator
- The **not** operator returns **True** if the condition is **False** and vice versa.

```
age = 22

# Can this person legally vote in the US?
not (age < 18)

# Note: This expression evaluates whether a person
# is not underage

# The "not" word can be separated by a space and the parentheses are not necessary
# but the parentheses can be helpful to organize your code logically
not age < 18
```

True

Condition A and B satisfied: &

- We can test multiple conditions using the & operator
- The & operator returns **True** if both conditions are **True**
- **()** is optional but always recommended!

```
print(5>3 & 6<10)  
print((5>3) & (6<10))
```

True

True

Condition A or B: | operator

- We test whether at least one condition is **True** with **|** operator

```
# We use the "|" symbol to separate "OR" conditions.  
age = 31
```

```
# Is this age higher than 20 or lower than 30?  
( age >= 20 ) | ( age <= 30 )
```

True

```
# Another example  
student_status = "freshman"  
  
# Is the student in the first two years of undergrad?  
(student_status == "freshman") | (student_status == "sophomore")
```

True

and and &; or and |

- “and” and “or” are logical operators to evaluate Boolean expressions
- “&” and “|” are bitwise operators
 - It involves converting values to binary codes
 - Beyond the scope of our class
- If you are comparing Boolean expressions, and/& and or/| are equivalent.

```
print(5>3 & 6<10)  
print((5>3) & (6<10))
```

True
True

```
print(5>3 and 6<10)  
print((5>3) and (6<10))
```

True
True

Relationship between 0/1 and True/False

- When an integer (or floating) value is 0, it is considered as False.
- When an integer (or floating) value is 1, it is considered as True.

```
1== True
```

True

```
1.0==True
```

True

```
0==False
```

True

```
0.0==False
```

True

```
1.1==True
```

False

```
1.1==False
```

False

Questions?

Flow Control: Conditional Statements

- Conditional statements are used to make decisions
- We use this a lot in data cleaning or data analysis to filter out data!

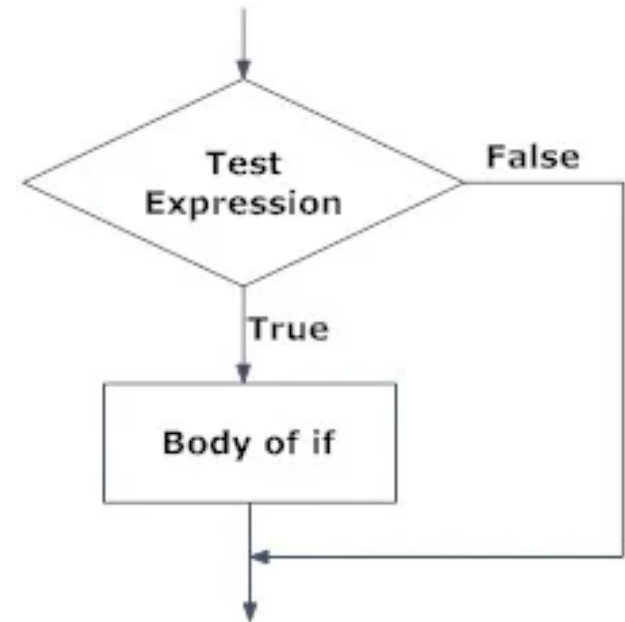


Fig: Operation of if statement

Flow Control: Conditional Statements

- The **if** statement is used to execute a block of code if a condition is **True**
- The **elif** statement is used to execute a block of code if the first condition is **False** and the second condition is **True**
- The **else** statement is used to execute a block of code if all other conditions are **False**

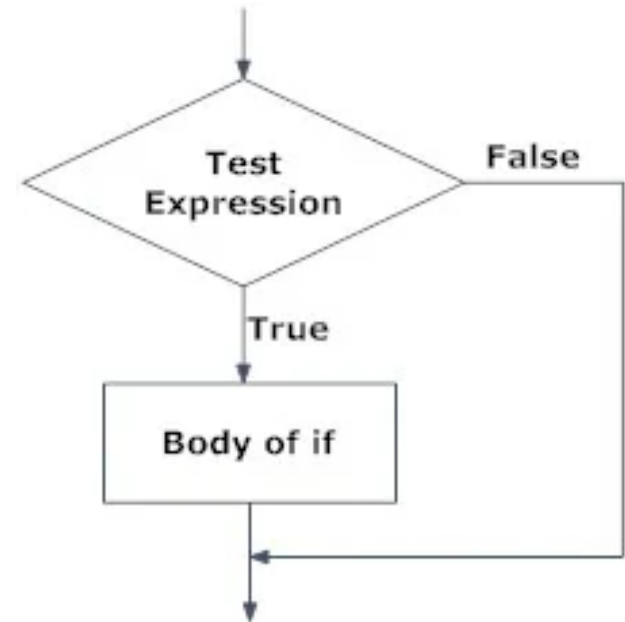


Fig: Operation of if statement

If statements

```
if condition:  
    body
```

- Type “if” followed by a logical condition and the “:” symbol
 - The “:” says: run the following command
- Body of expression
 - The “body” of the “if” statement needs to be indented with four spaces
 - Should be automatically indented when PyCharm detects “:”, otherwise, press the “tab” button on the keyboard.
- Another example:

```
# We start by defining a string  
any_questions = "yes"  
  
if any_questions == "yes":  
    print("Need to give more explanations")
```

Need to give more explanations

If/else statements

- What if we have two outcomes?
- The **else** statement is used to execute codes if the condition is **False**.
- The **else** is always used in conjunction with the **if** statement.

```
if condition:  
    body1  
else:  
    body2
```

- General syntax ->
- Another example using voting eligibility
 - Four different inputs

If/else statement

Q: Can you figure out why?

```
age = 17
national = True

if national & (age >= 18):
    print('This person is eligible to vote in US')
else:
    print('This person is NOT eligible to vote in US')
```

This person is NOT eligible to vote in US

```
age = 18
national = False

if national & (age >= 18):
    print('This person is eligible to vote in US')
else:
    print('This person is NOT eligible to vote in US')
```

This person is NOT eligible to vote in US

```
age = 22
national = False

if national & (age >= 18):
    print('This person is eligible to vote in US')
else:
    print('This person is NOT eligible to vote in US')
```

This person is NOT eligible to vote in US

```
age = 22
national = True

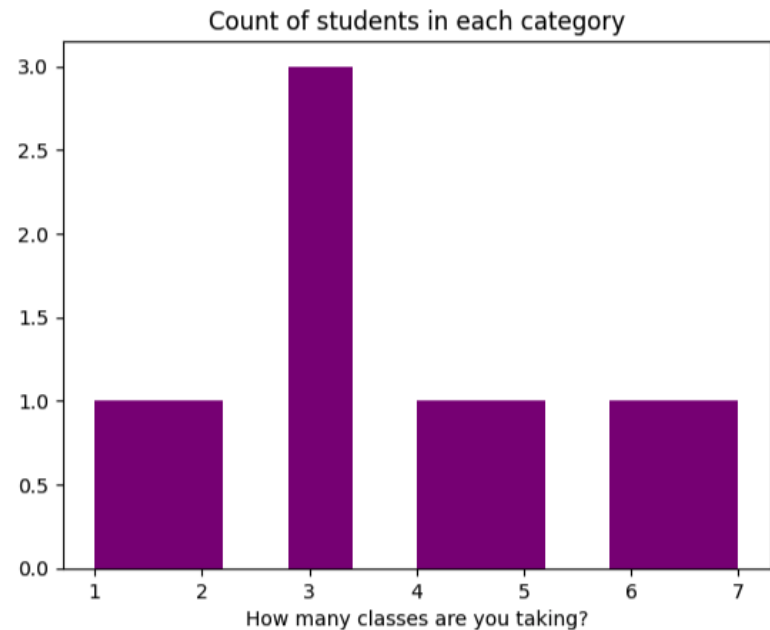
if national & (age >= 18):
    print('This person is eligible to vote in US')
else:
    print('This person is NOT eligible to vote in US')
```

This person is eligible to vote in US

Another example with plotting

```
is_graph_red = False
how_many_classes = np.array([7,1,2,3,3,3,4,5,6])

if is_graph_red:
    plt.hist(x = how_many_classes, color="red")
    plt.title("Count of students in each category")
    plt.xlabel("How many classes are you taking?")
    plt.show()
else:
    plt.hist(x = how_many_classes, color="purple")
    plt.title("Count of students in each category")
    plt.xlabel("How many classes are you taking?")
    plt.show()
```



What if `is_graph_red = True`?

...

If/elif/else statements

- The last conditional statement is the **elif** statement
- The **elif** statement is used to execute a block of code if the first condition is **False** but the second condition is **True**.
- It is like another **if** statement, but with more than two outcomes.

General Syntax:

```
if test_expression:  
    Body  
elif test_expression:  
    Body  
else:  
    Body
```

Okay to have multiple **elif** statements

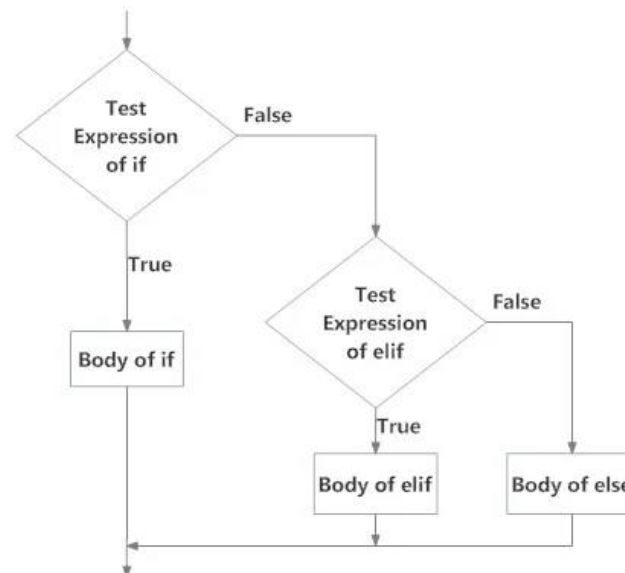


Fig: Operation of if...elif...else statement

Example

```
years_in_program = 1

if years_in_program == 1:
    print("This student is a freshman")
elif years_in_program == 2:
    print("This student is a sophomore")
elif years_in_program == 3:
    print("This student is a junior")
else:
    print("This student is a senior")

# Try changing the initial input
```

This student is a freshman

Questions?

- Practice week-03-boolean-if-else.ipynb