33721 lines (33721 loc) · 1.68 MB

Help method

- help(function) will return documentation of functions
- help(numpy) will return documentation of numpy library

Dynamic Typing

- variable type is determined only during runtime (execution of program)
- Python will infer the last line of code block as the output
- Need to specify output if we need to capture specific code output by different names

In [2]:
```
x = 3
x = 3.14
x = 'apple'
x
```

Out[2]: 'apple'

In [3]:
```
type(x)
```

Out[3]: str

Python Data Type

- int: 1
- float: 1.0
- bool: True, False
- str: 'need'
- None: None

In [4]:
```
a = 1
b = 1.0
c = False
d = 'Love'
e = None

type(a)
```

Out[4]: int

In [5]:
```
type(b)
```

Out[5]:
```
float
```

In [5]:
```
type(c)
```

Out[5]:
```
bool
```

In [6]:
```
type(d)
```

Out[6]:
```
str
```

In [7]:
```
type(e)
```

Out[7]:
```
NoneType
```

Python Functions

- format: def func(a, b, c = 0, d = 1):
- a and b are positional argument
- c and d are keyword argument with default values

In [8]:
```
def add_two(x):
  return x + 2

add_two(2)
```

Out[8]:
```
4
```

- positional arguments must be entered in order

In [9]:
```
def substract(x, y):
  return x - y

substract(3, 1)
```

Out[9]:
```
2
```

- keyword arguments must follow positional argument

In [10]:
```
def proportion(a, b, precision = 2):
  return round(a/b, precision)

proportion(2, 3, precision=2)
```

Out[10]:
```
0.67
```

String Formatting

- str(value) converts value into a string

In [11]:
```
x = 3.14
type(str(x))
```

Out[11]: str

- %0.2f will convert the float with 2 decimal places

In [12]:
```
x = 3.1415926
print('%0.2f' % x)
type(x)
```

```
3.14
```

Out[12]: float

In [13]:
```
"{:0.10f}".format(x) # convert x into a string with 10 decimal place
```

Out[13]: '3.1415926000'

In [14]:
```
f'{x:0.10f}' # alternatively
```

Out[14]: '3.1415926000'

In [15]:
```
f'x = {x:0.2f}' # insert expression by a string
```

Out[15]: 'x = 3.14'

In [16]:
```
f'{x = :0.2f}' # alternatively
```

Out[16]: 'x = 3.14'

- multi-line string: use /n to first line will be continued to the next line
- use print() to resolve /n

In [17]:
```
"""The answer is this:
x is {}""".format(x)
```

Out[17]: 'The answer is this:\nx is 3.1415926'

In [18]:
```
print("""The answer is this:
x is {}""".format(x))
```

```
The answer is this:
x is 3.1415926
```

List

- any elements in list doesn't have to be of the same type

In [19]:
```
x = [42, 'e', 2.0]
x
```

Out[19]:
```
[42, 'e', 2.0]
```

- list indexing refer the first element as 0

In [20]:
```
x[0]
```

Out[20]:
```
42
```

- the last element of list can be indexed by [-1]

In [21]:
```
x[-1]
```

Out[21]:
```
2.0
```

- x[index] = value
- convert a new value to the specific index in the list

In [22]:
```
x[2] = 4 # convert the element of 2nd index into 4
x
```

Out[22]:
```
[42, 'e', 4]
```

- list.append(value)
- add value to the end of list

In [23]:
```
x.append('a')
x
```

Out[23]:
```
[42, 'e', 4, 'a']
```

- list.pop(index_number): remove speficic value by its index in the list
- returned value indicate as value to be removed later in the list

In [24]:
```
x.pop(1)
```

Out[24]:
```
'e'
```

In [25]:
```
x
```

Out[25]:
```
[42, 4, 'a']
```

Dictionary

- {key: value} pairs
- key can be string
- value can be any type of object and value

In [26]:
```
y = {'b': [2,1], 'a': 1, 'c': 4}
y
```

Out[26]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

- dict['key'] can return the value corresponding to the key in original dictionary

In [27]:
```
y['b']
```

Out[27]:
```
[2, 1]
```

- dict['key'] = value
- add a new key with corresponding to new key into the dictionary

In [28]:
```
y['d'] = 3
y
```

Out[28]:
```
{'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}
```

- dict.pop(key, None)
- if the key exists in the original dictionary, then it will be returned a value to be removed

In [29]:
```
y.pop('d', None)
```

Out[29]:
```
3
```

In [30]:
```
y
```

Out[30]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

- if they key doesn't exist, it will be returned as None (default format)

In [31]:
```
y.pop('x', None) # it will return nothing, since key 'x' is not in the di
```

In [32]:
```
y
```

Out[32]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

- dict.keys()
- it returns a set of keys in dictionary

In [33]:
```
y.keys()
```

Out[33]:
```
dict_keys(['b', 'a', 'c'])
```

- dic.values()
- it returns values in the dictionary

In [34]:
```
y.values()
```

Out[34]:
```
dict_values([[2, 1], 1, 4])
```

- dict.items()
- it returns (key, value) pairs in the dictionary

In [35]:
```
y.items()
```

Out[35]:
```
dict_items([('b', [2, 1]), ('a', 1), ('c', 4)])
```

In [36]:
```
list(y.items()) # it converts dictionary into a list of (key, values) pai
```

Out[36]:
```
[('b', [2, 1]), ('a', 1), ('c', 4)]
```

- create a dict of mid-term and final grade as list

In [37]:
```
grades_list = {'a': [90, 100], 'b': [80, 90], 'c': [70, 80], 'd': [60, 70
grades_list
```

Out[37]:
```
{'a': [90, 100], 'b': [80, 90], 'c': [70, 80], 'd': [60, 70], 'f': [0, 6
```

```
0]}
```

- query for grade of student 'a'

In [38]:
```
grades_list['a'] # query for the mid and final grades
```

Out[38]:
```
[90, 100]
```

- query for mid-term grade of student 'a'

In [39]:
```
grades_list['a'][0] # since value is list, use [0] as mid-term grade for
```

Out[39]:
```
90
```

- create a dict of mid-term and final grade as dictionary embedded
- under embedded dict, mid and final grades formatted as {mid: final} where mid will be index for final

In [40]:
```
grades_dict = {'a': {90: 100},'b': {80: 90}, 'c': {70: 80}, 'd': {60: 70},
grades_dict
```

Out[40]:
```
{'a': {90: 100}, 'b': {80: 90}, 'c': {70: 80}, 'd': {60: 70}, 'f': {0: 6
0}}
```

- query for grades of student 'a'

In [41]:
```
grades_dict['a']
```

Out[41]:
```
{90: 100}
```

- query for final grades of student 'a'

In [42]:
```
grades_dict['a'][90] # we need to use mid-term as index key to query for
```

Out[42]:
```
100
```

Tuples

- tuples cannot be changed inside
- tuple indexing = list indexing
- cannot convert or assign a new value inside the tuple
- cannot add or remove the element from tuple

In [43]:
```
# inside the tuple, any element cannot be changed
z = (2, 'need', 3, 1.0)
z
```

Out[43]:
```
(2, 'need', 3, 1.0)
```

- tuple indexing

In [44]:
```
z[1]
```

Out[44]:
```
'need'
```

In [46]:
```
z[1] = 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-119466b456d8> in <cell line: 1>()
----> 1 z[1] = 3

TypeError: 'tuple' object does not support item assignment
```

In [47]:
```
z.remove(2)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-47-baeee67eb7b3> in <cell line: 1>()
----> 1 z.remove(2)

AttributeError: 'tuple' object has no attribute 'remove'
```

- list element inside tuple can be modified

In [48]:
```
a = (1, 2, [10, 12])
a
```

Out[48]:
```
(1, 2, [10, 12])
```

In [49]:
```
a[2][0] = 15 # convert first element in the list to be modified to 15
a
```

Out[49]:
```
(1, 2, [15, 12])
```

In [50]:
```
a[2].append(100) # add or remove any list elements by appending or removi
a
```

Out[50]:
```
(1, 2, [15, 12, 100])
```

- If there is tuple inside the list, we can add or remove tuple but cannot change

element inside the tuple

In [51]:
```
b = [1, (2, 'need', 3, 1.0), 8, 9]
b
```

Out[51]:
```
[1, (2, 'need', 3, 1.0), 8, 9]
```

In [52]:
```
b.pop(1) # remove tuple inside list
b
```

Out[52]:
```
[1, 8, 9]
```

In [53]:
```
b.append((10, 1000)) # add a new tuple inside the list
b
```

Out[53]:
```
[1, 8, 9, (10, 1000)]
```

Set

- set is collection of unique elements
- unordered collection of elements that can be any data type
- set cannot be treated as list for indexing

In [54]:
```
# convert list into set
lst = [1,1,1,1,1,1,5,5,'Jack']
lst = list(set(lst))
```

In [55]:
```
lst
```

Out[55]:
```
[1, 'Jack', 5]
```

- also can use dict form for set expression

In [56]:
```
# {value} can be used as set
x = {2, 'e', 'e'}
x
```

Out[56]:
```
{2, 'e'}
```

- alternatively

In [57]:
```
# set([value]) also can be used as set
x = set([2, 'e', 'e'])
x
```

Out[57]: {2, 'e'}

- we can insert a new element into set
- set_name.add(value)

In [58]:
```
x.add(1)
x
```

Out[58]: {1, 2, 'e'}

- we can remove element from list
- set_name.remove(value)

In [59]:
```
x.remove('e')
x
```

Out[59]: {1, 2}

- Set operation: Intersection
- find the common values between sets

In [60]:
```
x.intersection({2, 3})
```

Out[60]: {2}

- Set operation: Difference
- find element in the former set that are not in latter set

In [61]:
```
x.difference({2,3})
```

Out[61]: {1}

- Set property
- set cannot be indexed as list

In [62]:
```
x[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-62-2f755f117ac9> in <cell line: 1>()
----> 1 x[0]

TypeError: 'set' object is not subscriptable
```

Length in any entities

- len(list) returns the number of elements in the list

In [63]:
```
len([1, 2, 3])
```

Out[63]: 3

- len(dict) returns the number of key-value pairs in the dict

In [64]:
```
len({'a':1,'b':2,'c':3})
```

Out[64]: 3

- len(string) returns the number of characters in the string

In [65]:
```
len('apple')
```

Out[65]: 5

Exception

- exception method is used when the "unknown" error occurs, the code will bypass the error and continue executing following code

In [66]:
```
try:
    eval('x === y')  # Invalid syntax
except SyntaxError as e:
    print(f"There's a syntax error!: \n{e}")
```

```
There's a syntax error!:
invalid syntax (<string>, line 1)
```

- f"expression of error: \n{e}"
- use e to unfold specific error from system

In [67]:
```
try:
    exec('def my_func(): \nprint("Hello")')  # Missing indentation (it sh
except IndentationError:
    print("There is an indentation error!")
```

```
There is an indentation error!
```

In [68]:
```
try:
```

```
    int('abc')  # Cannot convert 'abc' to an integer (it should be int('1
except ValueError:
    print("There was a ValueError!")
```

```
There was a ValueError!
```

In [69]:
```
try:
    'a' + 2 # Cannot add a string and an integer (it should be 'a' + str(
except TypeError as e:
    print(f"We did this on purpose, and here's what's wrong:\n{e}")
```

```
We did this on purpose, and here's what's wrong:
can only concatenate str (not "int") to str
```

In [70]:
```
try:
    my_list = [1, 2, 3]
    print(my_list[5])  # Index 5 doesn't exist
except IndexError:
    print("Index out of range!")
```

```
Index out of range!
```

In [71]:
```
try:
    my_dict = {'a': 1, 'b': 2}
    print(my_dict['c'])  # 'c' key doesn't exist
except KeyError:
    print("Key not found!")
```

```
Key not found!
```

In [72]:
```
# we can loop through except arguments to catch specific errors
try:
    set([1,2,3])[0]
except SyntaxError as e:
    print(f"Print this if there's a syntax error")
except Exception as e:
    print(f"Print this for any other error")
```

```
Print this for any other error
```

Truthiness

- use boolean to determine: True (1) and False (0)

Comparison Operators

- equality: ==

In [73]:
```
3 == 3
```

Out[73]: True

- inequality: !=

In [74]:
```
3 != 4
```

Out[74]:
```
True
```

- less than: <

In [75]:
```
3 < 4
```

Out[75]:
```
True
```

- greater than >

In [76]:
```
3 > 4
```

Out[76]:
```
False
```

Logical Operator

- True: 1
- False: 0
- or: addition
- and: multiplication
- not: negation

In [77]:
```
((3 > 5) or ((3 < 4) and (5 > 4))) and not (3 == 5)

# (True or (False and True)) and not (False)
# (1 or (0 and 1)) and not (0)
# (1 + (0 * 1)) * 1
# 1 and 1 => 1 => True
```

Out[77]:
```
True
```

List operator

- any(): at least one element in the list is true

In [78]:
```
any([0, 0, 0])
```

Out[78]:
```
False
```

- all(): all element should be true

In [79]:
```
all([True, 1, 0])
```

Out[79]:
```
False
```

Assert

- assert function, 'error message'
- use assert to test anything we know should be true
- it will raise exception when assertion is false, otherwise shows nothing if true

In [80]:
```
assert 2 + 2 != 4
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-80-033180eb2704> in <cell line: 1>()
----> 1 assert 2 + 2 != 4

AssertionError:
```

In [81]:
```
assert 1 == 0, "1 does not equal 0"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-81-6f246726711b> in <cell line: 1>()
----> 1 assert 1 == 0, "1 does not equal 0"

AssertionError: 1 does not equal 0
```

In [82]:
```
assert 1 == 1 # function is true, then no error message from assertion
```

Control Flow

if-else flow

- if expression 1: elif expression 2: else expression

In [83]:
```
x = 3
if x > 0:
  print('x > 0')
elif x < 0:
  print('x < 0')
else:
  print('x == 0')
```

```
x > 0
```

- convert 'if else' into a single line
- print (expression 1) if (condition for expression 1 is true) else print (expression 2)

In [84]:
```
print('x < 0') if (x < 0) else print ('x >= 0')
```

```
x >= 0
```

In [85]:
```
if x < 0:
  print('x < 0')
else:
  print('x >= 0')
```

```
x >= 0
```

for loop

- for loop can iterate over a sequence of values (list, string)
- use for loop when you want to repeat instructions for a fixed number of times
- use for loop when iterating over a sequence of values and perform the same operation on each element

In [87]:
```
a = []
for x in [0, 1, 2]: # iterate over each element in the list [0, 1, 2]
  a.append(x) # append each element to the list a
a

# loop stops automatically when it reaches to the end of list
# append each element iterables to the list a
```

Out[87]:
```
[0, 1, 2]
```

while loop

- while loop can run indefinitely until condition is no longer true
- while loop can run as long as a specific conditions are true
- use while when # of iterations is unknown or when loop should continue until a certain condition is met

In [88]:
```
x = 0
while x < 3: # it continues loop till x < 3 where x can be 0, 1, 2
  x += 1 # starting from x = 0, it will add 1 to x in each iteration
# when x reach 2, loop will stop and perform 2 += 1 => 3
x
```

Out[88]:
```
3
```

break statement

- break: break out of current loop
- once the certain condition is met, the loop will stop there

In [89]:
```
x = 0
while True: # infinite loop due to condition is always true
  x += 1
  if x == 10:
    print(x)
    break # once x = 10 by x += 1, loop will stop by using break
    # No any statement below break
```

```
10
```

continue statement

- continue is used to skip current iteration and move to next iteration
- current iteration is skipped to the rest

In [90]:
```
for x in range(5):
  if x == 1:
    print(x)
```

```
1
```

In [91]:
```
for x in range(5):
  if x == 1:
    continue
  print(x)
```

```
0
2
3
4
```

In [92]:
```
for name in ['John', 'Ben', 'Julia']:
    if name == 'Ben':
```

EODS_FALL_2024 / Mid-Term / EODS_Mid_Review.ipynb ↑ Top

Preview Code Blame Raw

```
    if name == 'Ben':
        continue # once the expression is met, the loop will skip to the
    print(name)
```

```
John
Julia
```

range

- range(n) indicates from 0 to (n-1), the list values inside
- range(4) indicates from 0 to 3

In [94]:
```
a = []
for i in range(4):
  a.append(i)
a
```

Out[94]:
```
[0, 1, 2, 3]
```

In [95]:
```
# alternatively

list(range(4))
```

Out[95]:
```
[0, 1, 2, 3]
```

- range (n, m) indicates range of values from n to m + 1
- start from element with index n (nth element) to element with index m - 1 (mth element)

In [96]:
```
list(range(3, 5))

# [0, 1, 2, 3 (3rd index, 3rd element), 4 (4th index, 5th element), 5]
```

Out[96]:
```
[3, 4]
```

In [97]:
```
list(range(0, 10, 2))
```

Out[97]:
```
[0, 2, 4, 6, 8]
```

enumerate

- enumerate(list)
- add index counter to iterable (list, tuple, etc) to track list index or for-loop iteration
- return both index and items in each iteration of loop

In [98]:
```
for i, x in enumerate(['a', 'b', 'c']):
  print(i, x)
```

```
0 a
1 b
2 c
```

In [99]:
```
list(enumerate(['a', 'b', 'c']))
```

Out[99]:
```
[(0, 'a'), (1, 'b'), (2, 'c')]
```

Sorting

- list.sort()
- sort list in ascending order
- such method only modify original list but doesn't create a new list
- list.sorted(reverse = True), soring list in descending order

In [100…
```
x = [4, 1, 2, 3]
x.sort()
```

In [101…
```
x
```

Out[101…
```
[1, 2, 3, 4]
```

In [102…
```
x.sort(reverse = True)
```

In [103…
```
x
```

Out[103…
```
[4, 3, 2, 1]
```

- sorted(list)
- funtion returns a new sorted list from original list
- it sorts list in ascending order
- sorted(list, reverse = True), sorting list in descending order

In [104…
```
y = [4, 1, 2, 3]
sorted(y)
```

Out[104…
```
[1, 2, 3, 4]
```

In [105…
```
sorted(y, reverse = True)
```

Out[105…
```
[4, 3, 2, 1]
```

Lambda function

- for ('a', 3), lambda x: x[1] returns 3
- for ('b', 5), lambda x: x[1] returns 5

- for ('c', 1), lambda x: x[1] returns 1
- use lambda function to sort the dictionary by keys in x[0] or values in x[1], the output will be sorted by keys or values in ascending order.

In [106…
```
d = {'a':3,'b':5,'c':1}

s = sorted(d.items(), key = lambda x: x[1])
s
```

Out[106…
```
[('c', 1), ('a', 3), ('b', 5)]
```

In [107…
```
s = sorted(d.items(), key = lambda x: x[0])
s
```

Out[107…
```
[('a', 3), ('b', 5), ('c', 1)]
```

List Comprehension

- expression for item in iterable if condition
- expression: operation performed on each item in the iterable
- for item in iterable: loop that iterates over each item in the iterate

In [108…
```
# which integers between 0 and 3 inclusive are divisible by 2?
result = []
for i in range(0, 4):
  result.append(i % 2 == 0)
result
```

Out[108…
```
[True, False, True, False]
```

In [109…
```
[i % 2 == 0 for i in range(0, 4)]
```

Out[109…
```
[True, False, True, False]
```

In [110…
```
# what are the indices of the vowels in 'apple'?
vowels = ['a','e','i','o','u']
[i for i,x in enumerate('apple') if x not in vowels]
```

Out[110…
```
[1, 2, 3]
```

In [111…
```
# using both for and if-else condition without list comprehension
a = []
for x in range(5):
    if x % 2 == 0:
        a.append(x * 2)
    else:
```

```
        a.append(x)
print(a)
```

```
[0, 1, 4, 3, 8]
```

In [112…

```
# use the same condition in list comprehension
a = [x * 2 if x % 2 == 0 else x for x in range(5)]
print (a)
```

```
[0, 1, 4, 3, 8]
```

Dictionary Comprehension

- {key_expression: value_expression for item in iterable if condition}
- key expression: expression that generates key for dictionary
- iterable: an iterable object (list, tuple, set, etc)
- dictionary comprehension works as list comprehension but with key-value pairs

In [113…

```
pairs = [(1,'e'),(2,'f'),(3,'g')]
dict(pairs)
```

Out[113… {1: 'e', 2: 'f', 3: 'g'}

In [114…

```
# modify value and only include odd keys
{key: val for key, val in pairs if key % 2 == 1}
```

Out[114… {1: 'e', 3: 'g'}

In [115…

```
# Creating a Dictionary from Two Lists

keys = ['a', 'b', 'c']
values = [1, 2, 3]

combined_dict = {k: v for k, v in zip(keys, values)}
print(combined_dict)
```

```
{'a': 1, 'b': 2, 'c': 3}
```

In [116…

```
# traditional for loop with dictionary
even_squares = {}
for x in range(1, 6): # iterate over numbers 1 to 5
    if x % 2 == 0: # x is key in even_squares dictionary
        even_squares[x] = x**2 # even_squares[x] indicate when if-conditi

print(even_squares)
```

```
{2: 4, 4: 16}
```

In [117…

```
# Alternatively, we can use dict comprehension to achieve the same result

even_squares = {x: x**2 for x in range(1, 6) if x % 2 == 0}
# x: x**2 indicate the key and value expression in the dict to be returned
# range(1, 6) is the range of numbers to be used as keys => in this case,
# if x % 2 == 0 is the condition to filter out odd numbers. If the condit

print(even_squares)
```

```
{2: 4, 4: 16}
```

Strong typing

- variables do have a type and that type matters when performing operations on a variable

In [118…

```
x, y = 5, 'five'
x + y # since x and y share different data type, then they cannot operate
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-118-3ea7a61dfaa7> in <cell line: 2>()
      1 x, y = 5, 'five'
----> 2 x + y # since x and y share different data type, then they cannot o
perate

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Numpy array VS Python list

Numpy array

- it contains a single pointer to one continguous block of data
- it can access or operate individual elements in a Numpy array much faster than Python list
- single pointer can have numpy array more memory-efficient than Python list

Python List

- it contains a pointer to a block of pointers, each of which points to a full python object (int, str, bool, etc)
- process much slower than numpy array
- multiple pointers indicate list is more flexible than numpy arrays since each element is a full structure with data and type info
- list can be filled with any data types whereas numpy lacks flexibility but can be more efficient for storing and manipulating data

In [119…
```
import numpy as np
```

Numpy datatypes

- bool_ Boolean (True or False) stored as a byte
- int_ Default integer type (same as C long; normally either int64 or int32)
- intc Identical to C int (normally int32 or int64)
- intp Integer used for indexing (same as C ssize_t; normally either int32 or int64)
- int8 Byte (-128 to 127)
- int16 Integer (-32768 to 32767)
- int32 Integer (-2147483648 to 2147483647)
- int64 Integer (-9223372036854775808 to 9223372036854775807)
- uint8 Unsigned integer (0 to 255)
- uint16 Unsigned integer (0 to 65535)
- uint32 Unsigned integer (0 to 4294967295)
- uint64 Unsigned integer (0 to 18446744073709551615)
- float_ Shorthand for float64.
- float16 Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
- float32 Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
- float64 Double precision float: sign bit, 11 bits exponent, 52 bits mantissa

Numpy arrays

- np.array() function
- it is used to create a numpy array from a python list or tuple
- it takes a single argument (list or tuple)

In [120…
```
# convert a list into array
x = np.array([1, 2, 3])
x
```

Out[120… `array([1, 2, 3])`

In [121…
```
type(x)
```

Out[121… `numpy.ndarray`

In [122…
```
# since the list contains all int, datatype is uniformed
x.dtype
```

Out[122… `dtype('int64')`

- array only can contain one datatype otherwise it will give unassigned datatype

In [123…
```
x = np.array([1, 'two', 3])
x
```

Out[123…
```
array(['1', 'two', '3'], dtype='<U21')
```

- np.zeros() creates an array of zeros with specified size and datatype
- np.zeros(# of zeros, dtype = data type)

In [124…
```
np.zeros(5, dtype = float)
```

Out[124…
```
array([0., 0., 0., 0., 0.])
```

Array Indexing

- np.arange(start (inclusive), stop (exclusive), step (steps to skip)
- array indexing works as python list

In [125…
```
x = np.arange(1, 6)
x
```

Out[125…
```
array([1, 2, 3, 4, 5])
```

In [126…
```
x[0], x[-1], x[3]
```

Out[126…
```
(1, 5, 4)
```

Array Slicing

In [127…
```
x = np.arange(5)
x
```

Out[127…
```
array([0, 1, 2, 3, 4])
```

- return first 2 items, start: end(exlusive)
- x[index of first item (inclusive) : index of last item + 1 (exclusive)]

In [128…
```
# x[0 (index of first item => 0) : 2 (index of second item + 1 => 1)]
x[0: 4]
```

Out[128…
```
array([0, 1, 2, 3])
```

- missing value before or after :

In [129…
```
# x[: index of last item + 1] => it will return all the elements from the
x[:4]
```

Out[129…
```
array([0, 1, 2, 3])
```

In [130…
```
# x[index of first item: ] => it will return all the elements from the in
x[2:]
```

Out[130…
```
array([2, 3, 4])
```

In [131…
```
# x[-3:] returns array without first 3 items
x[-3:]
```

Out[131…
```
array([2, 3, 4])
```

In [132…
```
# x[:-2] returns array without last 2 elements
x[:-2]
```

Out[132…
```
array([0, 1, 2])
```

In [133…
```
# x[:] returns all elements
x[:]
```

Out[133…
```
array([0, 1, 2, 3, 4])
```

Array slicing with step size

- x[start (first item index): end (last item index + 1): step_size]

In [134…
```
x
```

Out[134…
```
array([0, 1, 2, 3, 4])
```

In [135…
```
x[2: 4: 1]
```

Out[135…
```
array([2, 3])
```

reverse array with -1 step size

- x[::-(# of step size)] reverses order of elements with specified steps

In [136…
```
# x[::-1] means to reverse the order of the elements in the array x with s

x[::-1]
```

Out[136…
```
array([4, 3, 2, 1, 0])
```

In [137…
```
x[::-2]
```

Out[137…
```
array([4, 2, 0])
```

Numpy fancy indexing

- x[[item 1 index, item 2 index .....]] => it can extract a sublist from the numpy array based on the given index range and step.
- it will return the multiple but non-consecutive indices at once using list
- there is not index rule

In [138…
```
x = np.arange(5, 10)
x
```

Out[138…
```
array([5, 6, 7, 8, 9])
```

In [139…
```
x[0:3]
```

Out[139…
```
array([5, 6, 7])
```

In [140…
```
# list of indices

x[[0, 3]]
```

Out[140…
```
array([5, 8])
```

In [141…
```
x[[0, 2, -1]]
```

Out[141…
```
array([5, 7, 9])
```

In [142…
```
x[[0,2,-2,-1]]
```

Out[142…
```
array([5, 7, 8, 9])
```

Boolean Indexing using a Boolean Mask

- it selects everything in array and returns a new array with only values that corresponds to true values in the mask
- the mask is a boolean array with same shape as original array

In [143…
```
x
```

Out[143…
```
array([5, 6, 7, 8, 9])
```

In [144…
```
# it will return the boolean array with True for indices where the conditi
x % 2 == 0
```

Out[144…
```
array([False,  True, False,  True, False])
```

- x[condition expression] will return the array of values of x where the condition expression is True.

In [145…
```
x[x%2 == 0]
```

Out[145…
```
array([6, 8])
```

In [146…
```
x[x >= 7]
```

Out[146…
```
array([7, 8, 9])
```

Bitwise Operators

- (condition expression) will return boolean array with True for elements where condition is True, and False for elements where condition is False.

In [147…
```
x
```

Out[147…
```
array([5, 6, 7, 8, 9])
```

In [148…
```
(x%2 == 0)
```

Out[148…
```
array([False,  True, False,  True, False])
```

In [149…
```
(x > 6)
```

Out[149…
```
array([False, False,  True,  True,  True])
```

- comparison operator (and, or, not) cannot be applied to the numpy boolean

- comparison operator (and, or, not) cannot be applied to the numpy boolean indexing
- comparison operator expect both elements to be boolean for comparison, not array of boolean

In [150…
```
(3 > 2)
```

Out[150… True

In [ ]:
```
(5 > 4)
```

Out[ ]: True

In [151…
```
# both boolean can be compared via comparison operator
(3 > 2) and (5 > 4)
```

Out[151… True

In [152…
```
(x%2==0)
```

Out[152… array([False, True, False, True, False])

In [153…
```
(x > 6)
```

Out[153… array([False, False, True, True, True])

In [154…
```
# both boolean arrays cannot be compared via comparison operator
(x%2 == 0) and (x > 6)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-154-016fe7c1f1c4> in <cell line: 2>()
      1 # both boolean arrays cannot be compared via comparison operator
----> 2 (x%2 == 0) and (x > 6)

ValueError: The truth value of an array with more than one element is ambig
uous. Use a.any() or a.all()
```

- boolean array cannot be combined by "and" "or" "not"
- boolean array can be combined by "&" "|" "~"

In [155…
```
x
```

Out[155… array([5, 6, 7, 8, 9])

- (condition1 & condition2) acts as boolean numpy array and returns True or False for each element in array.

In [156…
```
(x%2 == 0) & (x > 6)

# it can return a boolean array with True for each element that satisfies
```

Out[156…
```
array([False, False, False,  True, False])
```

- Apply boolean mask will return item array satisfies with conditions

In [157…
```
x[(x%2 == 0) & (x > 6)]
```

Out[157…
```
array([8])
```

and: &

- & is the intersection operator
- it returns only the values that are present in both arrays.

In [158…
```
x[(x%2 == 0) & (x > 6)]
```

Out[158…
```
array([8])
```

or: |

- | is the OR operator
- it returns the values in whole set

In [159…
```
x[(x%2 == 0) | (x > 6)]
```

Out[159…
```
array([6, 7, 8, 9])
```

not: ~

- ~ is NOT operator
- it returns the boolean value of the opposite of the expression inside the parentheses

In [160…
```
( (x%2 == 0) | (x > 6) )
```

Out[160…
```
array([False,  True,  True,  True,  True])
```

In [161…
```
~( (x%2 == 0) | (x > 6) )
```

Out[161… `array([ True, False, False, False, False])`

In [162…
```
x[~( (x%2 == 0) | (x > 6) )]
```

Out[162… `array([5])`

Multidimensional Lists

- [[list1], [list2]] represent 2 sub-lists in the original list

In [163…
```
x = [[1,2,3],[4,5,6]]
x
```

Out[163… `[[1, 2, 3], [4, 5, 6]]`

list indexing:

- x[index for rows, index for columns]
- index rule strictly follows list indexing

In [164…
```
# return 1st row
x[0]
```

Out[164… `[1, 2, 3]`

In [165…
```
# 2nd row, 3rd column

x[1][2]
```

Out[165… `6`

In [166…
```
# return 2nd column
[i[1] for i in x]
```

Out[166… `[2, 5]`

Multidimensional arrays

- To convert the list of lists into a 2D numpy array, we can use the numpy.array() function:

In [167…
```
x = np.array([1,2,3])
x
```

Out[167… array([1, 2, 3])

- to convert 2-D array, we need to shift lists from 1-d by single [ ] to double [[ ]] for how many lists inside are

In [168…
```
x = np.array([[1, 2, 3], [4, 5, 6]])
x
```

Out[168…
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [169…
```
y = np.array([[1,2,3,],[4,5,6,],[7,8,9]])
y
```

Out[169…
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

- x[index for row item, index for column item]

In [170…
```
x[0, 1]
```

Out[170… 2

- x[:, column indexing] will return all the rows with the specified column index.

In [171…
```
# return 2nd column
x[:, 1]
```

Out[171… array([2, 5])

- x[row indexing, :] will return all the columns of the specified row.

In [172…
```
# return first row
x[0, :]
```

Out[172… array([1, 2, 3])

Array attributes

In [173…
```
x = np.array([1,2,3])
x.ndim
```

Out[173… 1

In [174…
```
x = np.array([[1,2,3],[4,5,6]])
x
```

Out[174…
```
array([[1, 2, 3],
       [4, 5, 6]])
```

- array.ndim is the number of dimensions of the array.

In [175…
```
# number of dimensions
x.ndim
# array have 2 dimensions
```

Out[175…
```
2
```

- array.shape returns (row x column) dimensions

In [176…
```
# shape in dimension
x.shape
```

Out[176…
```
(2, 3)
```

- array.size return # of elements

In [177…
```
x.size
```

Out[177…
```
6
```

Numpy operations (UFuncs)

- list operation can only contatenate different lists without arithmetic operation

In [178…
```
x = [1,2,3]
y = [4,5,6]

x + y
```

Out[178…
```
[1, 2, 3, 4, 5, 6]
```

- Once we convert the list to a numpy array, we can perform mathematical operations on it (one to one) into a single row.

In [179…
```
x = np.array([1,2,3])
y = np.array([4,5,6])
```

```
x + y
```

Out[179… array([5, 7, 9])

Numpy Broadcasting

- a python list will not be operated by the ** operator

In [180…
```
# square every element in a list
x = [1,2,3]
x ** 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-180-1436245b91ac> in <cell line: 3>()
      1 # square every element in a list
      2 x = [1,2,3]
----> 3 x ** 2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

- numpy array allows vectorized computation on arrays of different sizes

In [181…
```
x = np.array([1,2,3])
x ** 2
```

Out[181… array([1, 4, 9])

- array is a high-dimensional object used to operate with another array of same or different shape or scalar value

In [182…
```
a = np.array([1, 2, 3])
b = 2 # scalar value
a * b
```

Out[182… array([2, 4, 6])

In [183…
```
b = np.array([2, 5, 8])
a + b
```

Out[183… array([3, 7, 11])

In [184…
```
b = np.array([[2, 5, 8],[1,3,9]])
a + b
```

Out[184…
```
array([[ 3,  7, 11],
       [ 2,  5, 12]])
```

numpy array summary

- array can deal with numerical data
- array can define matrix
- array can generate n-dimensional array with random numbers and linear algebra operations

Pandas

- series: 1-D array with flexible index
- dataframe: 2-D matrix with index and column names

In [185…
```
import pandas as pd
import numpy as np
```

Pandas series

- pandas series can translate data structures between 1-D array and 2-D table
- it can store data of any type (unlike array), including numbers, strings, booleans.
- series can be created using various methods like list, array, dict and panda dataframes

In [186…
```
s = pd.Series(np.random.rand(4)) # random.rand() generates a list of 4 ra
s # pandas series stored it into 1-D array in s with corresponding index.
```

Out[186…

	0
0	0.419522
1	0.472708
2	0.300702
3	0.093004

dtype: float64

In [187…
```
# returns values of series in form of 1-D numpy array
s.values
```

Out[187…
```
array([0.41952217, 0.47270821, 0.30070187, 0.09300415])
```

In [188…
```
# returns index of series
s.index
```

Out[188…
```
RangeIndex(start=0, stop=4, step=1)
```

- series index can be flexibly defined into anything hashable
- pd.series ([list], index =[str or int], 'name', dtype = data type)

In [189…
```
# create series from array and set index
a = pd.Series([1,2,3], index=['a','b','c'], name = 'number', dtype = floa
a
```

Out[189…

	number
a	1.0
b	2.0
c	3.0

dtype: float64

In [190…
```
a.ndim
```

Out[190…
```
1
```

- array["index_name"] to access a single value via index label => it is similar to dictionary["key"]

In [191…
```
a['a']
```

Out[191…
```
1.0
```

array.index[] returns index name by position

In [192…
```
a.index[2]
```

Out[192…
```
'c'
```

- create series with index from a dictionary

In [193…
```
b = pd.Series({'a':1, 'b':2, 'c':3, 'd':4}, name = 'dictionary', dtype =
b
```

Out[193…

	dictionary
a	1.0

b	2.0
c	3.0
d	4.0

dtype: float64

pandas dataframe

- convert 2-D dictionary into tabular datastructure
- each column has a single and unique datatype
- contains both row and column indices
- single column = series

In [194…
```
df = pd.DataFrame({'Year':[2017,2018,2018,2019],
                   'Semester':['Fall','Fall','Spring','Fall'],
                   'Measure_1':[2.1,3.0,2.4,1.9]
                  })
df
```

Out[194…

	Year	Semester	Measure_1
0	2017	Fall	2.1
1	2018	Fall	3.0
2	2018	Spring	2.4
3	2019	Fall	1.9

- pandas dataframe is built on basis of dictionary consisting of set of keys and lists of values
- pd.DataFrame({"key": [value1, value2, value3], "key2": [value4, value5, value6]})
- it converts 2-D array dictionary into a table where

1. columns are keys and rows are values
2. default index is 0, 1, 2... for each key

In [195…
```
print(df)
```

```
   Year Semester  Measure_1
0  2017     Fall        2.1
1  2018     Fall        3.0
2  2018   Spring        2.4
3  2019     Fall        1.9
```

In [196…
```
display(df)
```

	Year	Semester	Measure_1
0	2017	Fall	2.1
1	2018	Fall	3.0
2	2018	Spring	2.4
3	2019	Fall	1.9

collection of series

- dataframe is collection of series in pandas
- DataFrame convert the list into a 2-D dataframe with columns and index labels

In [197…
```
# 2-D multidimensional list
data = [[2017,'Fall',2.1],
        [2018,'Fall',3.0],
        [2018,'Spring',2.4],
        [2019,'Fall',1.9]]
```

In [198…
```
# dataframe convert 2-D list into table with column and index
df = pd.DataFrame(data,
                  columns=['Year','Semester','Measure_1'],
                  index=['001','002','003','004'])
df
```

Out[198…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4
004	2019	Fall	1.9

Pandas Attributes

- get shape of DataFrame: shape
- it will return the number of rows and columns in the dataframe

In [199…
```
df.shape
```

Out[199…
```
(4, 3)
```

- get column values

In [200…

In [200…
```
df.columns
```

Out[200… `Index(['Year', 'Semester', 'Measure_1'], dtype='object')`

In [201…
```
df.ndim
```

Out[201… 2

Pandas Indexing

In [202…
```
df
```

Out[202…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4
004	2019	Fall	1.9

.loc[ ]

- .loc[ ] can query row and column by index name and column name
- .loc[ ] is used for selecting rows/columns by label (label-based)

In [203…
```
# df.loc['index_name'] will return the row with the index name 'index_name
df.loc['001']
```

Out[203…

	001
Year	2017
Semester	Fall
Measure_1	2.1

dtype: object

In [204…
```
df.loc['001', :]
```

Out[204…

	001
Year	2017
Semester	Fall

Measure_1	2.1

dtype: object

In [205…
```
# df.loc[:, 'column_name'] will return the column with the column name 'c

df.loc[:, 'Year']
```

Out[205…

	Year
001	2017
002	2018
003	2018
004	2019

dtype: int64

In [206…
```
assert df.loc['Year'], 'error'
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in get_
loc(self, key)
   3804         try:
-> 3805             return self._engine.get_loc(casted_key)
   3806         except KeyError as err:

index.pyx in pandas._libs.index.IndexEngine.get_loc()

index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectH
ashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectH
ashTable.get_item()

KeyError: 'Year'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
<ipython-input-206-aa51c4d00c7d> in <cell line: 1>()
----> 1 assert df.loc['Year'], 'error'

/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in __getite
m__(self, key)
   1189             maybe_callable = com.apply_if_callable(key, self.obj)
   1190             maybe_callable = self._check_deprecated_callable_usage
(key, maybe_callable)
-> 1191             return self._getitem_axis(maybe_callable, axis=axis)
   1192
   1193     def _is_scalar_access(self, key: tuple):
```

```
/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in _getitem
_axis(self, key, axis)
   1429         # fall thru to straight lookup
   1430         self._validate_key(key, axis)
-> 1431         return self._get_label(key, axis=axis)
   1432
   1433     def _get_slice_axis(self, slice_obj: slice, axis: AxisInt):

/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in _get_lab
el(self, label, axis)
   1379     def _get_label(self, label, axis: AxisInt):
   1380         # GH#5567 this will fail if the label is not present in the
axis.
-> 1381         return self.obj.xs(label, axis=axis)
   1382
   1383     def _handle_lowerdim_multi_index_axis0(self, tup: tuple):

/usr/local/lib/python3.10/dist-packages/pandas/core/generic.py in xs(self,
key, axis, level, drop_level)
   4299                     new_index = index[loc]
   4300         else:
-> 4301             loc = index.get_loc(key)
   4302
   4303             if isinstance(loc, np.ndarray):

/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in get_
loc(self, key)
   3810             ):
   3811                 raise InvalidIndexError(key)
-> 3812             raise KeyError(key) from err
   3813         except TypeError:
   3814             # If we have a listlike key, _check_indexing_error will
raise

KeyError: 'Year'
```

In [207…

```
# df.loc['index_name']['column_name'] returns the value of a specific cel

df.loc['001','Measure_1']
```

Out[207… 2.1

.iloc[ ]

- .iloc[ ] can retrieve info (single value or range of values) based on row and column index
- .iloc[ ] can be used for selecting row/column by index position (integer-based), considering 0 index

In [208…

```
df
```

Out[208…

	Year	Semester	Measure_1
001	2017	Fall	2.1

002	2018	Fall	3.0
003	2018	Spring	2.4
004	2019	Fall	1.9

In [209…

```
# df.iloc[index of row] returns the row with all columns

df.iloc[0]
```

Out[209…

	001
Year	2017
Semester	Fall
Measure_1	2.1

dtype: object

In [210…

```
df.iloc[0, :]
```

Out[210…

	001
Year	2017
Semester	Fall
Measure_1	2.1

dtype: object

In [211…

```
 # df.iloc[:, column_index] will return the column with column index with

 df.iloc[:, 2]
```

Out[211…

	Measure_1
001	2.1
002	3.0
003	2.4
004	1.9

dtype: float64

In [212…

```
# df.iloc[index of row, index of column]
df.iloc[0, 2]
```

Out[212… 2.1

indexing with multiple individual rows and columns by .loc[[x,y],[a,b]]

In [213…
```
df
```

Out[213…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4
004	2019	Fall	1.9

- select rows by index names
- df.loc[['row_index1, 'row_index2']]

In [214…
```
df.loc[['002', '004']]
```

Out[214…

	Year	Semester	Measure_1
002	2018	Fall	3.0
004	2019	Fall	1.9

In [215…
```
df.loc[['002','004'], :]
```

Out[215…

	Year	Semester	Measure_1
002	2018	Fall	3.0
004	2019	Fall	1.9

- select columns by column names and return all rows
- df.loc[:,['column_name1', 'column_name2']]

In [216…
```
df.loc[:, ['Year', 'Semester']]
```

Out[216…

	Year	Semester
001	2017	Fall
002	2018	Fall
003	2018	Spring

004	2019	Fall

- select rows and columns by corresponding names
- df.loc[[row_name1: row_name2], [column_name1: column_name2]]

In [217…
```
df.loc[['002', '004'], ['Year', 'Semester']]
```

Out[217…

	Year	Semester
002	2018	Fall
004	2019	Fall

indexing with multiple consecutive rows and columns using .loc[x:y, a:b]

- df.loc['row_indexer1':'row_indexer2'] returns sepcified rows from the dataframe including all columns
- we need to consider the returned output is 1-D or 2-D
- x:y = from x to y

In [218…
```
# get the last 2 rows
df.loc['003':, :]
```

Out[218…

	Year	Semester	Measure_1
003	2018	Spring	2.4
004	2019	Fall	1.9

In [219…
```
# alternatively

df.loc['003':]
```

Out[219…

	Year	Semester	Measure_1
003	2018	Spring	2.4
004	2019	Fall	1.9

In [220…
```
# get the first 2 rows

df.loc[:'002', :]
```

Out[220…

	Year	Semester	Measure_1
001	2017	Fall	2.1

002	2018	Fall	3.0

- df.loc[:, 'column_name1': 'column_name2'] will return a new dataframe with the all rows and specified columns inclusively.

In [221…
```
# get last 2 columns

df.loc[:, 'Semester':]
```

Out[221…

	Semester	Measure_1
001	Fall	2.1
002	Fall	3.0
003	Spring	2.4
004	Fall	1.9

In [222…
```
# get first 2 columns

df.loc[:, :'Semester']
```

Out[222…

	Year	Semester
001	2017	Fall
002	2018	Fall
003	2018	Spring
004	2019	Fall

- df.loc['row_indexer1': 'row_indexer2', 'column_name1': 'column_name2']
- return a new dataframe with the specified rows and columns inclusively.

In [223…
```
# get 2nd to 4th row and last 2 columns

df.loc['002':'004', 'Semester':]
```

Out[223…

	Semester	Measure_1
002	Fall	3.0
003	Spring	2.4
004	Fall	1.9

indexing with multiple individual rows and columns by .iloc[[x,y], [a,b]]

by .iloc[[x,y], [a,b]]

- df.iloc[[x,y],[a,b]]
- for each index for row and columns, all follow list indexing rule

- df.iloc[[x, y], :]
- returns specific rows with all columns

In [224…
```
# get 2nd and 3rd rows and all columns
df.iloc[[1, 2]]
```

Out[224…

	Year	Semester	Measure_1
002	2018	Fall	3.0
003	2018	Spring	2.4

In [225…
```
# alternatively
df.iloc[[1, 2], :]
```

Out[225…

	Year	Semester	Measure_1
002	2018	Fall	3.0
003	2018	Spring	2.4

- df.iloc[:, [a, b]]
- returns specific columns with all rows

In [226…
```
# get 1st and 2nd columns with all rows

df.iloc[:, [0, 1]]
```

Out[226…

	Year	Semester
001	2017	Fall
002	2018	Fall
003	2018	Spring
004	2019	Fall

- df.iloc[[x:y], [a:b]]
- returns specific rows and columns

In [227…
```
# get 2nd and 3rd row and 1st and last column
df.iloc[[1, 2], [0, 2]]
```

Out[227…

	Year	Measure_1
002	2018	3.0
003	2018	2.4

indexing with multiple consecutive rows and columns by .iloc[x:y, a:b]

In [228…

```
df
```

Out[228…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4
004	2019	Fall	1.9

- df.iloc[# of rows: # of columns, # of rows to skip]

In [229…

```
# df.iloc[# of rows to start (inclusively): ]
# return the last # of rows specified in the brackets and everything insi
# 行列都是左闭右开

# get last 2 rows with all columns for consecutive rows/columns
df.iloc[-2:]
```

Out[229…

	Year	Semester	Measure_1
003	2018	Spring	2.4
004	2019	Fall	1.9

In [230…

```
# alternatively

df.iloc[-2:, :]
```

Out[230…

	Year	Semester	Measure_1
003	2018	Spring	2.4
004	2019	Fall	1.9

In [231…

```
# df.iloc[row_index] for individual rows/columns
# return the last second rows with all columns

df.iloc[-2]
```

Out[231…

	003
Year	2018
Semester	Spring
Measure_1	2.4

dtype: object

In [232…

```
# df.iloc[: # of rows to end (exclusively)]

# get first 3 rows with all columns
df.iloc[:3]
```

Out[232…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4

In [233…

```
# alternatively

df.iloc[:3,:]
```

Out[233…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4

In [234…

```
# df.iloc[:, # of columns from start (inclusively):]

# get last 2 columns with all rows
# 左闭右开
df.iloc[:, 1:]
```

Out[234…

	Semester	Measure_1
001	Fall	2.1
002	Fall	3.0
003	Spring	2.4
004	Fall	1.9

In [235…

```
# df.iloc[:, : # of columns to the end (exclusively)]
```

```
# get the first columns with all rows
df.iloc[:, :1]
# it will give 2-D dataframe
```

Out[235…

	Year
001	2017
002	2018
003	2018
004	2019

In [236…

```
df.iloc[:, 0]
# it will give 1-D series
```

Out[236…

	Year
001	2017
002	2018
003	2018
004	2019

dtype: int64

- technique
- when asking first few rows or columns, use df.iloc[:x, :y]
- where x and y are exclusive, so we need to element + 1
- x + 1 y + 1

In [237…

```
# Get first two rows and first two columns

df.iloc[:2, :2]
```

Out[237…

	Year	Semester
001	2017	Fall
002	2018	Fall

- technique
- when asking last few rows or columns, use df.iloc[a:, b:]

- where a and b inclusive, so no need to add one
- a b

In [238…

```
# Get last 3 rows and last 2 columns

df.iloc[1:, 1:]
```

Out[238…

	Semester	Measure_1
002	Fall	3.0
003	Spring	2.4
004	Fall	1.9

Panda Selection Chaining

In [239…

```
df
```

Out[239…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4
004	2019	Fall	1.9

rule of selection chaining

.iloc[]:

- Primarily used for integer-location based indexing (position-based).
- When you select a single row or column, it returns a 1-D Series.
- When you select multiple rows and columns (using slices), it returns a 2-D DataFrame.

.loc[]:

- Primarily used for label-based indexing (using row and column labels).
- When you select a single row or column by label, it returns a 1-D Series.
- When you select multiple rows and columns (using slices or lists of labels), it returns a 2-D DataFrame.

In [240…
```
df.iloc[:3].ndim
```

Out[240… 2

In [241…
```
df.loc[:, 'Semester':].ndim
```

Out[241… 2

In [242…
```
# get last 2 columns and first 3 rows

df.iloc[:3].loc[:,'Semester':]
```

Out[242…

	Semester	Measure_1
001	Fall	2.1
002	Fall	3.0
003	Spring	2.4

In [243…
```
# alternatively
df.loc[:, 'Semester':].iloc[:3]

# both iloc and loc can swap since 2 return 2-D dataframe
```

Out[243…

	Semester	Measure_1
001	Fall	2.1
002	Fall	3.0
003	Spring	2.4

In [244…
```
# get 1st and 2nd rows and last 2 columns
df.loc[['001', '003'], :].iloc[:, 1:]
```

Out[244…

	Semester	Measure_1
001	Fall	2.1
003	Spring	2.4

In [245…
```
df.loc[['001', '003']].shape
```

Out[245… (2, 3)

In [246…
```
df.iloc[:, 1:].shape
```

Out[246… (4, 2)

- Techniques
- Series (1-D) cannot be followed by a dataframe (2-D) but a series (1-D)
- dataframe (2-D) can be followed by a Series (1-D) or another dataframe (2-D)

In [247…

```
# get 2nd row and last 2 columns

df.loc['002'].iloc[:, -2:]
```

```
---------------------------------------------------------------------------
IndexingError                             Traceback (most recent call last)
<ipython-input-247-be5bb1084e5a> in <cell line: 3>()
      1 # get 2nd row and last 2 columns
      2
----> 3 df.loc['002'].iloc[:, -2:]

/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in __getite
m__(self, key)
   1182             if self._is_scalar_access(key):
   1183                 return self.obj._get_value(*key, takeable=self._tak
eable)
-> 1184             return self._getitem_tuple(key)
   1185         else:
   1186             # we by definition only have the 0th axis

/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in _getitem
_tuple(self, tup)
   1688
   1689     def _getitem_tuple(self, tup: tuple):
-> 1690         tup = self._validate_tuple_indexer(tup)
   1691         with suppress(IndexingError):
   1692             return self._getitem_lowerdim(tup)

/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in _validat
e_tuple_indexer(self, key)
    960         Check the key for valid keys across my indexer.
    961         """
--> 962         key = self._validate_key_length(key)
    963         key = self._expand_ellipsis(key)
    964         for i, k in enumerate(key):

/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in _validat
e_key_length(self, key)
    999                     raise IndexingError(_one_ellipsis_message)
   1000                 return self._validate_key_length(key)
-> 1001             raise IndexingError("Too many indexers")
   1002         return key
   1003

IndexingError: Too many indexers
```

In [248…

```
df.loc['002'].ndim
```

Out[248… 1

In [249…
```
df.iloc[:, -2:].ndim
```

Out[249… 2

- since df.loc['002'] is 1-D series followed by df.iloc[:, -2:] as 2-D dataframe, the rule doesn't work

In [250…
```
# make the series after the dataframe works
df.iloc[:, -2:].loc['002']
```

Out[250…

	002
Semester	Fall
Measure_1	3.0

dtype: object

- We need to make sure the dimension matches beforing proceeding
- for single individual row or column indexing, keep in mind for both situation

- df.iloc[:, [0]] df.iloc[:, :] Pandas DataFrame
:
shape 4 1

In [251…
```
df.iloc[:, [0]]
```

Out[251…

	Year
001	2017
002	2018
003	2018
004	2019

In [252…
```
df.iloc[:, [0]].ndim
```

Out[252… 2

In [253…
```
df.iloc[:, :1]
```

Out[253…

	Year
001	2017

002	2018
003	2018
004	2019

In [254…
```
df.iloc[:, :1].ndim
```

Out[254… 2

- df.iloc[:, 0] Pandas Series (4,) 4

In [255…
```
df.iloc[:, 0]
```

Out[255…

	Year
001	2017
002	2018
003	2018
004	2019

dtype: int64

In [256…
```
df.iloc[:, 0].ndim
```

Out[256… 1

- series can be followed by a series

In [257…
```
# got the last 2 columns with 2nd row

# this time, .iloc did not directly index on a Series that would result i
df.loc['002'].iloc[1:]
```

Out[257…

	002
Semester	Fall
Measure_1	3.0

dtype: object

In [258…

```
df.loc['002']
```

Out[258…

	002
Year	2018
Semester	Fall
Measure_1	3.0

dtype: object

In [259…

```
df.iloc[:, 2:]
```

Out[259…

	Measure_1
001	2.1
002	3.0
003	2.4
004	1.9

In [260…

```
# series can only follow series or dataframe, so it depends on the previou
# .iloc[1:] is series indexing
# df.iloc[1:] is dataframe indexing since df is 2-d dataframe
df.iloc[:, 1:2].iloc[1:, 0]
```

Out[260…

	Semester
002	Fall
003	Spring
004	Fall

dtype: object

In [261…

```
df.iloc[1].ndim
```

Out[261… 1

In [262…

```
df.iloc[1:].ndim
```

Out[262… 2

Pandas Boolean Mask

- df.loc[row_indexer, column_name] == "value"

- return a boolean mask of the rows where the value in the specified column matches the specified value.

In [263…
```
df
```

Out[263…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4
004	2019	Fall	1.9

In [264…
```
# check each row in which semester is fall
df.loc[:,'Semester'] == 'Fall'
```

Out[264…

	Semester
001	True
002	True
003	False
004	True

dtype: bool

- df[df['column_name'] == 'value']
- return the rows where the value in the 'column_name' column is equal to 'value'

In [265…
```
# give rows where each has fall semester

df[df['Semester'] == 'Fall']
```

Out[265…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
004	2019	Fall	1.9

In [266…
```
# alternatively

df[df.Semester == 'Fall']
```

Out[266…

	Year	Semester	Measure_1

001	2017	Fall	2.1
002	2018	Fall	3.0
004	2019	Fall	1.9

- df.loc[df.column_name1 == 'value', 'column_name2']
- df.column_name == 'value' is row filtering
- 'column_name2' is columns filtering
- return all rows where the value in the column_name1 column is 'value' and return the values in the column_name2 column.

In [267…
```
# Get Measure_1 for all records for Semester 'Fall'

df.loc[df.Semester == 'Fall', 'Measure_1']
```

Out[267…

	Measure_1
001	2.1
002	3.0
004	1.9

dtype: float64

In [268…
```
# Get Measure_1 and Year columns for all records for Semester 'Fall'

df.loc[df.Semester == 'Fall', ['Measure_1', 'Year']]
```

Out[268…

	Measure_1	Year
001	2.1	2017
002	3.0	2018
004	1.9	2019

- by using chaining rule, we can filter more with bitwise operation

In [269…
```
# give rows where semester is fall and prior to 2019

df.loc[(df.Semester == 'Fall') & (df.Year < 2019), ['Measure_1', 'Year']]
```

Out[269…

	Measure_1	Year
001	2.1	2017
002	3.0	2018

- Get all records belonging to a set with `.isin` :
- '.isin' can allow to access specific records in a DataFrame based on a list of values.
- a type of multiple individual filtering

In [270…
```
df
```

Out[270…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0
003	2018	Spring	2.4
004	2019	Fall	1.9

- df.loc[df.column_name.isin (list_of_values)]
- return a new dataframe with only the rows where the column_name is in the list_of_values.

In [271…
```
# give all rows where year is in 2017 and 2019

df.loc[df.Year.isin([2017, 2019])]
```

Out[271…

	Year	Semester	Measure_1
001	2017	Fall	2.1
004	2019	Fall	1.9

Pandas Sorting

- sort_values() sorts data by specific columns
- df.sort_values(by='column_name', ascending=True/False)
- The "by" parameter takes the name of the column to sort by
- the "ascending" parameter is a boolean value that determines whether to sort the data in ascending or descending order.

In [272…
```
df
```

Out[272…

	Year	Semester	Measure_1
001	2017	Fall	2.1
002	2018	Fall	3.0

003	2018	Spring	2.4
004	2019	Fall	1.9

In [273…

```
# sort value for column "Measure_1"

df.sort_values(by=['Measure_1']).head(3)
```

Out[273…

	Year	Semester	Measure_1
004	2019	Fall	1.9
001	2017	Fall	2.1
003	2018	Spring	2.4

In [274…

```
df.sort_values(by=['Measure_1'],ascending=False).head(3)
```

Out[274…

	Year	Semester	Measure_1
002	2018	Fall	3.0
003	2018	Spring	2.4
001	2017	Fall	2.1

In [275…

```
df.sort_values(by=['Year','Measure_1'], ascending=False).head(3)
```

Out[275…

	Year	Semester	Measure_1
004	2019	Fall	1.9
002	2018	Fall	3.0
003	2018	Spring	2.4

Exploratory Data Analysis

In [277…

```
import pandas as pd

df_taxi = pd.read_csv('/content/yellowcab_demo_withdaycategories.csv', sep

# display first 5 rows
df_taxi.head(5)
```

Out[277…

	pickup_datetime	dropoff_datetime	trip_distance	fare_amount	tip_amount	payr
0	2017-01-05 14:49:04	2017-01-05 14:53:53	0.89	5.5	1.26	
1	2017-01-15 01:07:22	2017-01-15 01:26:47	2.70	14.0	0.00	

2	2017-01-29 09:55:00	2017-01-29 10:04:43	1.41	8.0	0.00
3	2017-01-10 05:40:12	2017-01-10 05:42:22	0.40	4.0	0.00
4	2017-01-06 17:02:48	2017-01-06 17:16:10	2.30	11.0	0.00

In [278…
```
# get columns names

df_taxi.columns
```

Out[278…
```
Index(['pickup_datetime', 'dropoff_datetime', 'trip_distance', 'fare_amou
nt',
       'tip_amount', 'payment_type', 'day_of_week', 'is_weekend'],
      dtype='object')
```

In [279…
```
# columns as arrays

df_taxi.columns.values
```

Out[279…
```
array(['pickup_datetime', 'dropoff_datetime', 'trip_distance',
       'fare_amount', 'tip_amount', 'payment_type', 'day_of_week',
       'is_weekend'], dtype=object)
```

In [280…
```
# columns as list

df_taxi.columns.tolist()
```

Out[280…
```
['pickup_datetime',
 'dropoff_datetime',
 'trip_distance',
 'fare_amount',
 'tip_amount',
 'payment_type',
 'day_of_week',
 'is_weekend']
```

In [281…
```
# get column datatypes

df_taxi.dtypes
```

Out[281…

	0
pickup_datetime	datetime64[ns]
dropoff_datetime	datetime64[ns]
trip_distance	float64
fare_amount	float64
tip_amount	float64
payment_type	object

day_of_week	int64
is_weekend	bool

dtype: object

summary of dataframe

- number of rows
- number of columns
- column names, number of filled values, datatypes
- number of each datatype seen
- size of dataset in memory

In [282…

```
# summary for dataframe

df_taxi.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column            Non-Null Count  Dtype         
---  ------            --------------  -----         
 0   pickup_datetime   1000 non-null   datetime64[ns]
 1   dropoff_datetime  1000 non-null   datetime64[ns]
 2   trip_distance     1000 non-null   float64       
 3   fare_amount       1000 non-null   float64       
 4   tip_amount        910 non-null    float64       
 5   payment_type      1000 non-null   object        
 6   day_of_week       1000 non-null   int64         
 7   is_weekend        1000 non-null   bool          
dtypes: bool(1), datetime64[ns](2), float64(3), int64(1), object(1)
memory usage: 55.8+ KB
```

summary statisitcs

- table.describe()

In [283…

```
df_taxi.describe()
```

Out[283…

	pickup_datetime	dropoff_datetime	trip_distance	fare_amount	tip_amo
count	1000	1000	1000.000000	1000.000000	910.000(
mean	2017-01-17 02:02:09.784000	2017-01-17 02:19:25.860999936	2.880010	12.442600	1.766
min	2017-01-01 00:15:27	2017-01-01 00:18:58	0.000000	2.500000	0.000(
25%	2017-01-09 13:56:56.750000128	2017-01-09 14:12:23.750000128	0.950000	6.500000	0.000(

50%	2017-01-17 04:56:38.500000	2017-01-17 05:04:04.500000	1.565000	9.000000	1.350(
75%	2017-01-24 21:32:36	2017-01-24 21:48:19.500000	3.100000	14.000000	2.460(
max	2017-01-31 23:16:30	2017-01-31 23:40:08	32.770000	88.000000	22.700(
std	NaN	NaN	3.678534	10.807802	2.315

variable types

- **Numeric** (eg. weight, temperature)
- usually has a zero value
- describes magnitude
- **Categorical** (eg. class, variety)
- usually a finite set
- no order
- **Ordinal** (eg. Like scale, education level, etc.)
- usually a finite set
- has order
- usually missing zero
- difference between levels may not be the same

Numeric Data range

- table.variable.min() will return the minimum value of the variable in the table.

In [284…

```
# get minimum trip distance value

df_taxi.trip_distance.min()
```

Out[284… `0.0`

- table.variable.max() will return the maximum value of the variable in the table.

In [285…

```
# get maximum trip distance value

df_taxi.trip_distance.max()
```

Out[285… 32.77

- table.min(numeric_only = True)
- return ONLY the columns with numeric value and return the minimum value on that column

In [286…
```
# get minimum values for each variables with numeric values

df_taxi.min(numeric_only=True)
# binary value (True/False) is also returned in False since False is recog
```

Out[286…

	0
trip_distance	0.0
fare_amount	2.5
tip_amount	0.0
day_of_week	0
is_weekend	False

dtype: object

Numeric: Central Tendency with Mean

$$\bar{x} = \frac{1}{n}\sum x_i$$

- table.variable.mean()
- returns the average value under the particular variable in the table

In [287…
```
# get the average value of fare amount

x = df_taxi.fare_amount.mean()
```

In [288…
```
# round it up to 2 decimal

print(f'{x = :0.2f}')
```

```
x = 12.44
```

Numeric: Central Tendency with Median

- Median: the outlier will not affect the median value so large
- Divides sorted dataset into two equal sizes

- 50% of the data is less than or equal to the median

- table.variable.median()
- return the median value of the variable in the table.

In [289…
```
# get the median of fare amount

df_taxi.fare_amount.median()
```

Out[289… 9.0

- Median is *robust* to outliers
- **Robust:** Not affected by outliers

Numeric: Quantiles/Percentiles

- Quantiles
- Quartiles divide the data into four equal parts:
- The 1st quartile (Q1) is the value below which 25% of the data falls.
- The 2nd quartile (Q2) is the median, where 50% of the data falls below it.
- The 3rd quartile (Q3) is the value below which 75% of the data falls.
- Percentiles divide the data into 100 equal parts.

In [290…
```
# .quantile(0.95):
# The quantile() method calculates the value below which 95% of the data
# it finds the 95th percentile of the fare_amount.

df_taxi['fare_amount'].quantile(.95, interpolation = 'linear')
```

Out[290… 33.5

- steps to calculate percentile with quantile()
- sort data in ascending order
- calculate position: P = [(N - 1) * q] + 1
  - N: the number of data in the fare_amount
  - q: the desired quantile (0.95 in this case for the 95th percentile).
  - P: gives data value that corresponds to the 95th percentile.
- This value indicates the point in the distribution where 95% of the data lies below it, and the remaining 5% of the data lies above it

it, and the remaining 5% of the data lies above it.

- linear interpolation represents the mean of the two nearest data points.

- result interpretation
- The code returns the 95th percentile value for the fare_amount column.
- This value represents the threshold below which 95% of the fare amounts in the dataset fall.
- result is 33.5, it means that 95% of the taxi fares are $33.5 or less.
- the value 33 is the fare amount that marks the boundary for the top 5% of fares. Any fare greater than 33.5 would fall into the top 5% (above the 95th percentile).

In [291…

```
# how may values falls within 5% and 95% percentile
df_taxi.fare_amount.quantile([.05,.95], interpolation='linear')
# 90% data falls within 4 and 33.5
```

Out[291…

	fare_amount
0.05	4.0
0.95	33.5

dtype: float64

In [292…

```
df_taxi.fare_amount.quantile([0,.25,.5,.75,1])
```

Out[292…

	fare_amount
0.00	2.5
0.25	6.5
0.50	9.0
0.75	14.0
1.00	88.0

dtype: float64

Numeric: Spread of variance

- To quantify how spread out the data is, we can use the variance.
- The variance measures how far the data points are spread out from the mean.
- A low variance indicates that the data points are close to the mean
- a high variance indicates that the data points are spread out.

- sample variance

sample variance

$$s^2 = \frac{\sum(x - \bar{x})^2}{n - 1}$$

- The term n-1 (degree of freedom) is used instead of n to correct for the fact that the sample mean x is an estimate of the population mean.
- Using n - 1 corrects this bias, ensuring that the sample variance is an unbiased estimator of the population variance.
- using n-1 ensures that the calculated sample variance provides an unbiased estimate of the true population variance by accounting for the fact that the sample mean uses up one degree of freedom.

In [293…
```
# give sample variance in fare amount

round(df_taxi.fare_amount.var(), 3)

# but sample variance doesn't give the same unit as original data
```

Out[293… 116.809

- when the sample variance is larger than the mean, it indicates that the data is more spread out

Numeric: Spread with Standard Deviation

- The standard deviation provides a measure of spread in the same units as the original data (unlike variance, which is in squared units).
- Standard deviation is more interpretable because it gives the typical distance of data points from the mean in the original units of measurement.

- sample standard deviation

$$s = \sqrt{\frac{\sum(x - \bar{x})^2}{n - 1}}$$

In [294…
```
# give standard deviation of fare amount
round(df_taxi.fare_amount.std(), 3)
```

Out[294… 10.808

- both sample variance and standard deviation is sensitive to outliers

Numeric: Explore spread with IQR

- a measure used to explore the spread or variability of a dataset, focusing on the middle 50% of the data.
- helps identify how the data is distributed and is less sensitive to extreme values (outliers) compared to measures like variance or standard deviation.

- Compute the IQR:
- The IQR is the difference between the third quartile and the first quartile:
- IQR = Q3 - Q1
- This range represents the middle 50% of the data (less sensitive to outliers)

In [295…
```
IQR = df_taxi.fare_amount.quantile(.75) - df_taxi.fare_amount.quantile(.25
IQR
```

Out[295… 7.5

Numeric: exploring distribution with skew

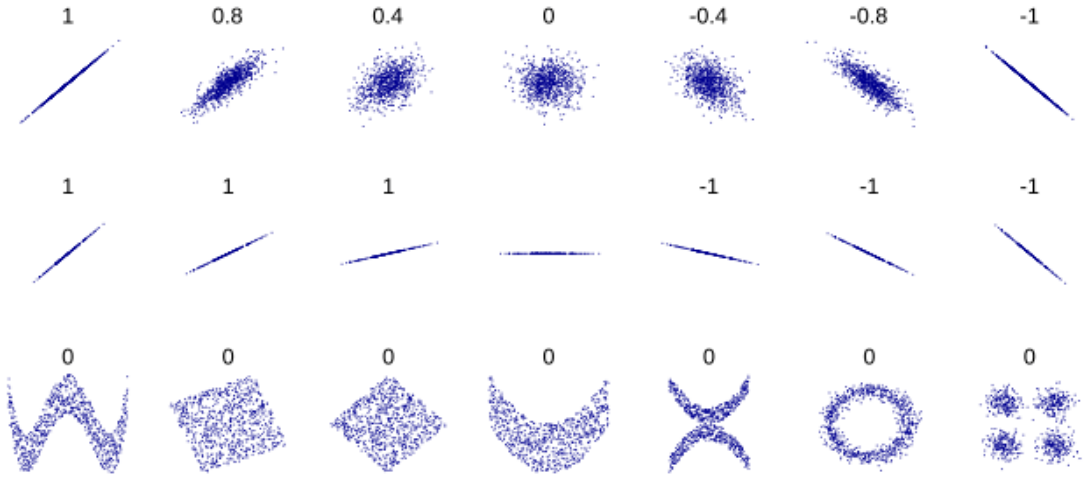
- Negative skewness
- tail to left
- mean < median
- Positive skewness
- tail to right
- median < mean

In [296…
```
# give the skewness of fare amount
df_taxi.fare_amount.skew()
```

Out[296… 2.882730031010152

Evaluation Correlation

- **Correlation:** the degree to which two variables are linearly related
- Takes values between:
- -1 (highly negatively correlated)
- 0 (not correlated)
- 1 (highly positively correlated)



- sample correlation
  - it measures the strength and direction of a linear relationship between two variables among samples
  - sample1.corr(sample2)

In [29…
```
# how trip distance correlates with fare amount
df_taxi.trip_distance.corr(df_taxi.fare_amount).round(2)
```

Out[297…
```
0.95
```

- Pearson correlation
  - The Pearson correlation coefficient (r) measures the strength and direction of a linear relationship between two variables from population
  - r, p = pearsonr(sample1, sample2) will return coefficient (r) and p-value(p)

In [29…
```
from scipy.stats import pearsonr
r, p = pearsonr(df_taxi.trip_distance, df_taxi.fare_amount)
print(f"{r = :.2f}, {p = :.2f}")
```

```
r = 0.95, p = 0.00
```

Categorical: counting values

- table.column.value_counts()

In [29…
```
# for each values under payment type, count each amount

df_taxi.payment_type.value_counts()
```

Out[299…

	count
payment_type	
Credit card	663

Credit card	853
Cash	335
No charge	2

dtype: int64

Categorical: grouping

- table.groupby('column_name').function_name
- group data by different value classes in column and operate those shared classes with functions

In [30…

```
# give me the average of all values with different payment types
df_taxi.groupby('payment_type').mean()
```

Out[300…

payment_type	pickup_datetime	dropoff_datetime	trip_distance	fare_amoun
Cash	2017-01-16 07:38:39.238806016	2017-01-16 07:59:25.835820800	2.732209	11.85671(
Credit card	2017-01-17 11:03:23.547511296	2017-01-17 11:18:55.428355840	2.961870	12.76108(
No charge	2017-01-20 20:22:23.500000000	2017-01-20 20:27:38.500000000	0.500000	5.00000(

- table.groupby('column_name')['column_name2'].mean()
- calculate the mean of 'column_name2' for each group in 'column_name'
- group by 'column_name' first where we are grouping by the values in 'column_name'
- Then select 'column_name2' and apply the mean function to 'column_name2'

In [30…

```
df_taxi.groupby('payment_type')['fare_amount'].mean()
```

Out[301…

payment_type	fare_amount
Cash	11.856716
Credit card	12.761086
No charge	5.000000

dtype: float64

- table.groupby('column_name')['column_name2'].sum()
- calculate the sum of 'column_name2' for each group in 'column_name'

In [30…
```
df_taxi.groupby('payment_type')['fare_amount'].sum()
```

Out[302…

payment_type	fare_amount
Cash	3972.0
Credit card	8460.6
No charge	10.0

dtype: float64

- Applying multiple function by .agg(['funtion1', 'function2', ...])
- table.groupby('column1')['column2'].agg(['count','mean','median'])
- return a table with the count, mean, and median of column2 grouped by column1.

In [30…
```
# give frequency, average and midian of trip distance by each payment cl
df_taxi.groupby('payment_type')['trip_distance'].agg(['count','mean','me
```

Out[303…

payment_type	count	mean	median
Cash	335	2.73	1.37
Credit card	663	2.96	1.70
No charge	2	0.50	0.50

- groupby multiple columns with the aggregation function for other columns

In [30…
```
# consider cash, credit card classes only under payment type, give mean
df_taxi[df_taxi.payment_type.isin(['Cash','Credit card'])].groupby(['pay
```

Out[304…

payment_type	is_weekend	mean	median
Cash	**False**	2.59	1.28
	True	3.51	2.10
	False	2.90	1.70

Credit card	**True**	3.30	1.74

- df_taxi[df_taxi.payment_type.isin(['Cash','Credit card'])] returns all classes fall in cash or credit card classes under payment_type
- df_taxi[] is a boolean masks to return all classes satisfied with those classes.

In [30…
```
df_taxi.payment_type.isin(['Cash','Credit card']).head(3)
```

Out[305…

	payment_type
0	True
1	True
2	True

dtype: bool

In [30…
```
x = df_taxi[df_taxi.payment_type.isin(['Cash','Credit card'])].head(3)
x
```

Out[306…

	pickup_datetime	dropoff_datetime	trip_distance	fare_amount	tip_amount	pa
0	2017-01-05 14:49:04	2017-01-05 14:53:53	0.89	5.5	1.26	
1	2017-01-15 01:07:22	2017-01-15 01:26:47	2.70	14.0	0.00	
2	2017-01-29 09:55:00	2017-01-29 10:04:43	1.41	8.0	0.00	

In [30…
```
y = x.groupby(['payment_type','is_weekend'])['trip_distance']
```

In [30…
```
y.agg(['mean','median']).round(2)
```

Out[308…

		mean	median
payment_type	**is_weekend**		
Cash	**True**	2.06	2.06
Credit card	**False**	0.89	0.89

Hypothesis Testing

In [30…
```
import numpy as np
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('darkgrid')

%matplotlib inline
```

In [31…

```
df_taxi = (
    pd.read_csv('/content/yellowcab_demo_withdaycategories.csv',
                header=1,
                parse_dates=['pickup_datetime','dropoff_datetime'])
    .assign(
        weekpart = lambda df_: df_.is_weekend.apply(lambda x: 'Weekend'
    )
    .loc[:,['trip_distance','is_weekend','weekpart']]
    .dropna()
)
print(df_taxi.shape)
display(df_taxi.head(5))
```

```
(1000, 3)
```

	trip_distance	is_weekend	weekpart
0	0.89	False	Weekday
1	2.70	True	Weekend
2	1.41	True	Weekend
3	0.40	False	Weekday
4	2.30	False	Weekday

Confidence Intervals and Hypothesis Testing

- Random Sampling: is the process of selecting a subset of a population to represent the entire population.
- Confidence Intervals: is the range of values that the true population parameter is likely to fall within given a certain level of confidence.
- Hypothesis Testing: is the process of determining whether a result is statistically significant or not.
- Permutation Tests: is a statistical method used to compare two or more groups or populations.
- A/B Tests: is a statistical method used to compare two versions of a website or app to determine which one performs better.
- p-values: is the probability of obtaining a result at least as extreme as the one observed, assuming that the null hypothesis is true.

- Multi-Armed Bandit: is a machine learning algorithm used to optimize the allocation of resources among different options.

Population Distribution and Sampling Distribution

- **Population Distribution:** The actual distribution out in the world
- **sampling distribution:** the distribution of the sample we take from the population
- We need a large sample size to get a good estimate of the true population distribution
- We need a small sample size to get a good estimate of the sampling distribution
- We need a sample size that is large enough to get a good estimate of the true population distribution, but not too large to get a biased estimate of the sampling distribution.

1. **Population Mean (μ) vs. Sample Mean ($\bar{x}$):**
    - **(μ)**: average of population
    - **($\bar{x}$)**: average of sample
    - Although we often use the sample mean to infer the population mean, they are not always exactly equal, especially with smaller sample sizes.
2. **Population Standard Deviation (σ) vs. Sample Standard Deviation (s):**

- **(σ)**: Measures the dispersion of the entire population. It reflects the true variability in the data.
- **(s)**: The spread of the sample data, used to estimate the population standard deviation.
- use degree of freedom (n-1) to address biased estimate of population variance or standard deviate

3. **Central Limit Therom**

- As the sample size increases, the sampling distribution of the sample mean will approach normality, regardless of the shape of the population distribution.
- larger sample sizes lead to a more accurate representation of the population mean.

4. **Random Sampling**:

- select a subset of data from population where each population sample has an equal chance of being chosen, ensuring that the sample is representative of the population, reducing bias and enabling generalization of results to the entire population.

In [31…
```
df_taxi.trip_distance.head(3)
```

Out[311…

	trip_distance
0	0.89
1	2.70
2	1.41

dtype: float64

Sampling without replacement

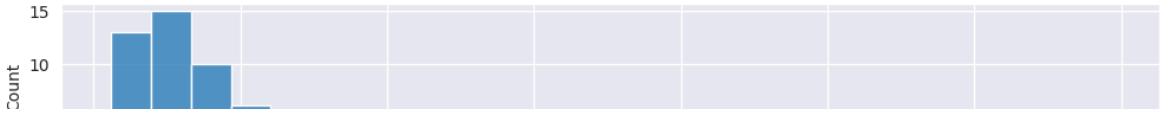
- Small populations: can avoid duplicate
- Surveys and finite populations
- Statistical tests requiring independence

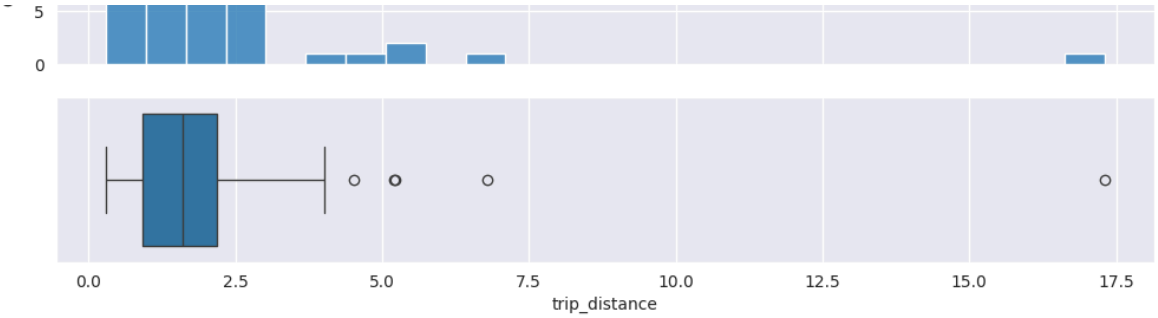
In [31…
```
# randomly sample 50 size without replacement for trip distance

trip_distance_sample = df_taxi.trip_distance.sample(
    n=50,              # our sample size
    random_state=123, # needed for reproducability
    replace=False     # sample without replacement
)

print(trip_distance_sample.describe().round(2))
print()
print(f"sample skew = {trip_distance_sample.skew().round(2)}")
```

```
count    50.00
mean      2.14
std       2.56
min       0.30
25%       0.91
50%       1.60
75%       2.19
max      17.30
Name: trip_distance, dtype: float64

sample skew = 4.55
```

- plot distribution of sample

In [37…
```
fig,ax = plt.subplots(2,1,figsize=(12,4),sharex=True)
sns.histplot(x=trip_distance_sample, ax=ax[0]);
sns.boxplot(x=trip_distance_sample, ax=ax[1]);
```





- evaluate such sampling is a good approximation

In [31…

```
trip_distance_sample_xbar = trip_distance_sample.mean()
print(f'sample mean: {trip_distance_sample_xbar:0.2f}')

# sample mean here indicates the average of the trip distances in the sa
```

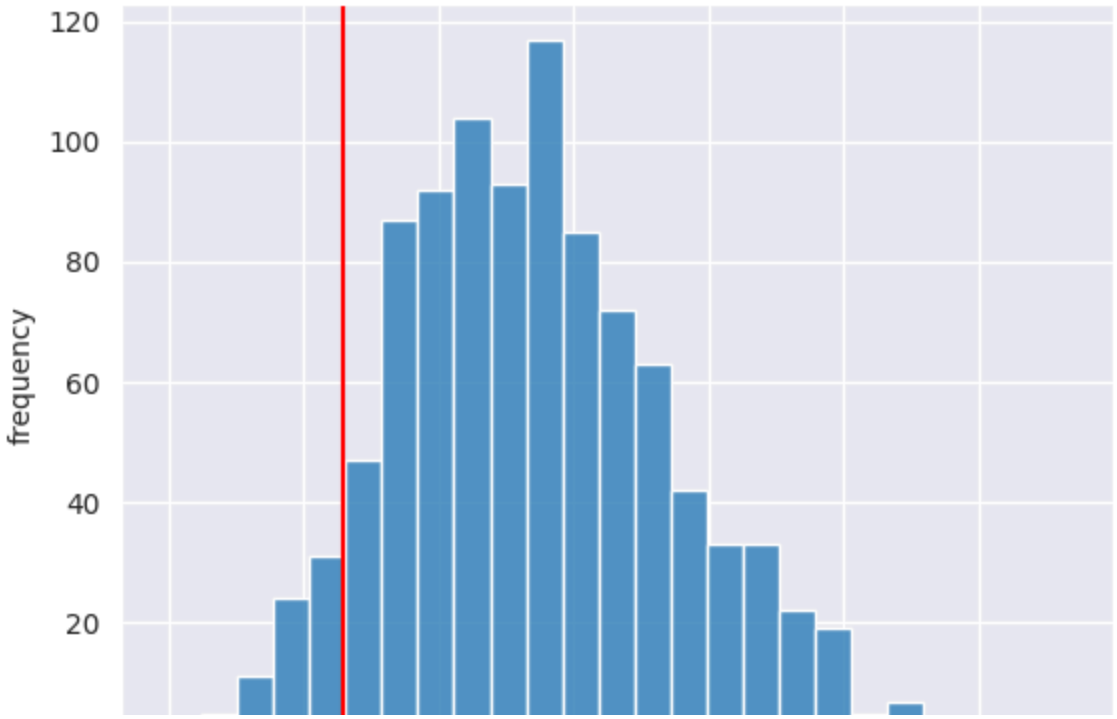
```
sample mean: 2.14
```

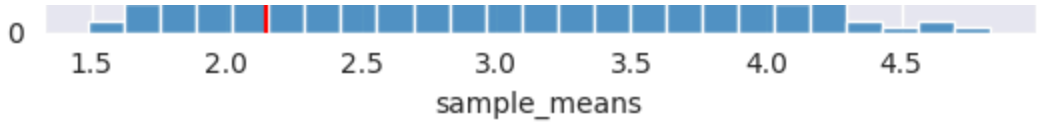
In [31…

```
# get distribution of 50 samples with sample means

sample_means = []
for i in range(1000):
    sample_mean = df_taxi.trip_distance.sample(n=50,random_state=i).mean
    sample_means.append(sample_mean)
```

In [31…

```
# plot mean in the distribution of samples

ax = sns.histplot(x=sample_means)
ax.set_xlabel('sample_means');
ax.set_ylabel('frequency');
ax.axvline(trip_distance_sample_xbar, color='red');
```





Sampling with replacement (bootstrap)

- Large populations distributions
- Bootstrap methods: In resampling techniques like bootstrapping, sampling with replacement is used to create multiple samples from the original data to estimate the sampling distribution of a statistic.
- Situations where repeated selection is possible: For example, in quality control testing where the same item could be tested multiple times.

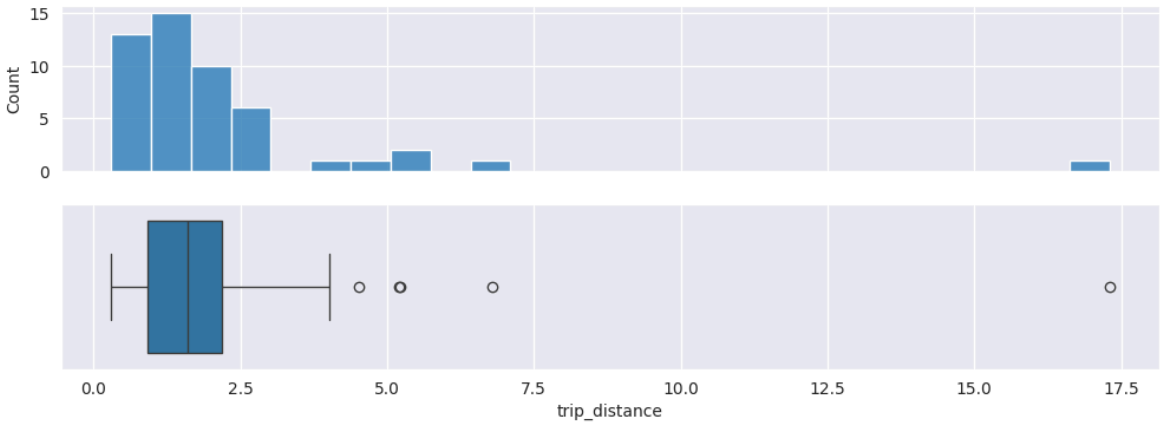
In [31…

```
trip_distance_sample = df_taxi.trip_distance.sample(
    n=50,              # our sample size
    random_state=123, # needed for reproducability
    replace=True      # sample without replacement
)

print(trip_distance_sample.describe().round(2))
print()
print(f"sample skew = {trip_distance_sample.skew().round(2)}")
```

```
count    50.00
mean      2.88
std       4.07
min       0.27
25%       0.82
50%       1.30
75%       2.33
max      17.90
Name: trip_distance, dtype: float64

sample skew = 2.52
```

In [37…

```
fig,ax = plt.subplots(2,1,figsize=(12,4),sharex=True)
sns.histplot(x=trip_distance_sample, ax=ax[0]);
sns.boxplot(x=trip_distance_sample, ax=ax[1]);
```



In [31…

```
trip_distance_sample_xbar = trip_distance_sample.mean()
print(f'sample mean: {trip_distance_sample_xbar:0.2f}')
```

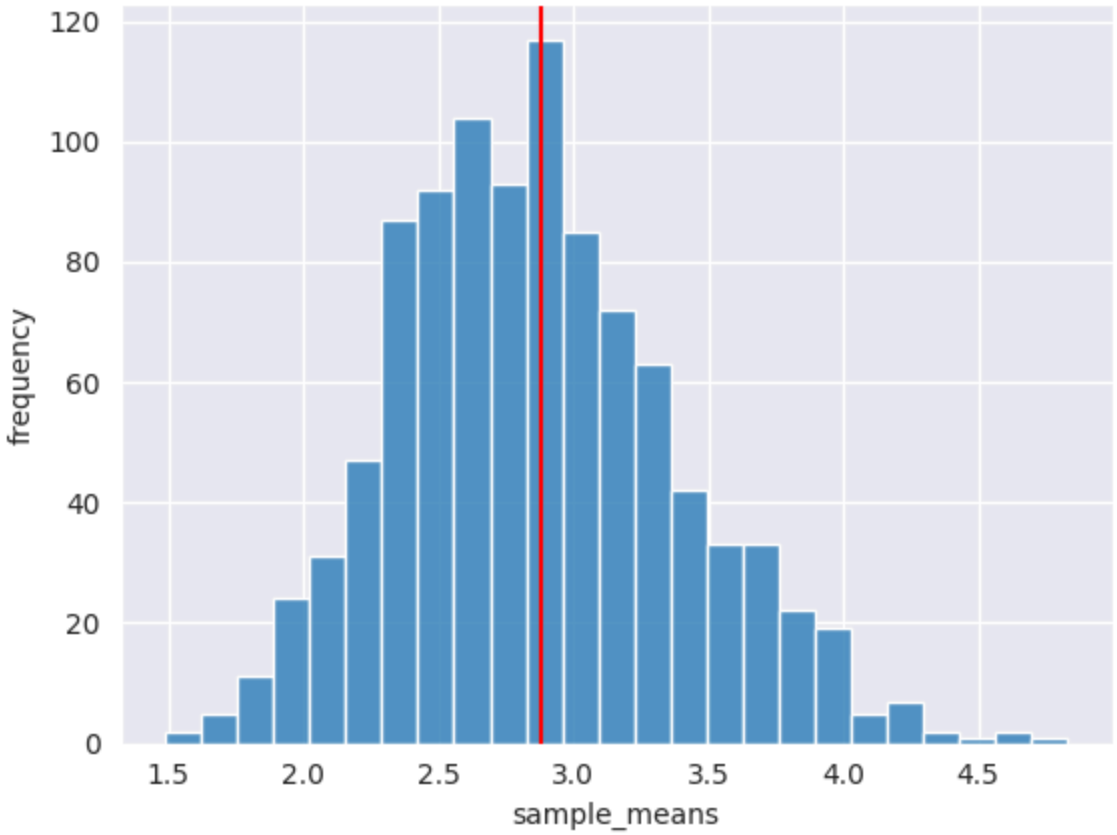
```
sample mean: 2.88
```

In [32…

```
sample_means = []
for i in range(1000):
    sample_mean = df_taxi.trip_distance.sample(n=50,random_state=i).mean
    sample_means.append(sample_mean)
```

In [32…

```
ax = sns.histplot(x=sample_means)
ax.set_xlabel('sample_means');
ax.set_ylabel('frequency');
ax.axvline(trip_distance_sample_xbar,color='red');
```



Confidence Intervals

- **Find a range of values from sample data** that is likely to contain the true population parameter (e.g., mean, proportion) with a specified level of confidence (95%)
- Since we typically only have one sample from a population, use confidence intervals to estimate where the true population parameter lies based on that sample.
- If we survey a group of people to estimate the average height in a population, the

confidence interval provides a range of plausible values for the true average height.

- It quantifies the uncertainty around a sample estimate to make inferences about the population.
- CI provide a way to understand the variability in our estimate. By providing a range instead of a single point estimate, we acknowledge that there is uncertainty in any sample-based estimate.
- A narrower confidence interval indicates more precision, while a wider interval suggests more uncertainty.

In [32…

```
# randomly sample 50 size without replacement for trip distance

trip_distance_sample = df_taxi.trip_distance.sample(
    n=50,              # our sample size
    random_state=123, # needed for reproducability
    replace=False      # sample without replacement
)

print(trip_distance_sample.describe().round(2))
print()
print(f"sample skew = {trip_distance_sample.skew().round(2)}")
```

```
count    50.00
mean      2.14
std       2.56
min       0.30
25%       0.91
50%       1.60
75%       2.19
max      17.30
Name: trip_distance, dtype: float64

sample skew = 4.55
```

In [32…

```
trip_distance_sample_xbar = trip_distance_sample.mean()
print(f'sample mean: {trip_distance_sample_xbar:0.2f}')
```

```
sample mean: 2.14
```

Plot CI with seaborn

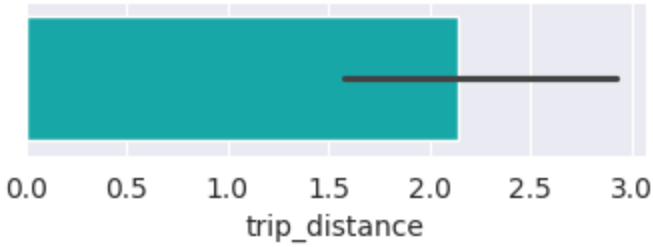
- bootstrapping is applied internally by Seaborn to resample the data with replacement from the original sample of 50 observations.
- For each bootstrap sample, the mean is calculated, and the distribution of these means is used to construct the 95% confidence interval.

In [37…

```
fig,ax = plt.subplots(1,1,figsize=(4,1))

sns.barplot(x=trip_distance_sample,
```

```
            estimator=np.mean, # default sample statistic
            ci=95,             # default 95% CI
            n_boot=1000,       # default number of bootstrap samples
            color='c',
           );
```

```
<ipython-input-374-42680bf29fb0>:3: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=('ci', 95)` for the same e
ffect.

  sns.barplot(x=trip_distance_sample,
```



- the mean generated with CI should be the same as the mean of the original sample of 50 observations.
- the black line on the plot from 1.6 to 2.9 represents the 95% confidence interval (CI) for the mean, indicating the range within which we can be **95% confident that the true population mean falls, based on the sampled data.**

How to generate CI

1. Generate Bootstrap Samples: Repeatedly sample with replacement from the original dataset.
2. Calculate the Statistic for Each Sample: Compute the mean for each bootstrap sample.
3. Construct the Confidence Interval: Use the distribution of bootstrap means to find the desired percentiles (e.g., 2.5th and 97.5th percentiles for a 95% confidence interval).

Creating a bootstrap confidence interval using resampling techniques

1. bootstrapping

- From the original dataset, randomly draw a sample of size (n) with replacement. This means that each data point can be selected more than once.
- This sampling process is known as bootstrapping, and the resampled dataset is called a bootstrap sample.

2. Record the Sample Statistic from This Random Sample

- Calculate the sample statistic (e.g., mean, median, standard deviation) for the bootstrap sample. This statistic will vary because each bootstrap sample is different.
- For example, if you are calculating a confidence interval for the mean, record the mean of the bootstrap sample.

3. Repeat Steps 1 and 2 Many Times

- Perform the resampling and statistic calculation steps multiple times (e.g., 1,000 times). Each time, a new bootstrap sample is drawn, and the sample statistic is recorded.
- This process generates a distribution of the sample statistic based on the bootstrap samples. This distribution is called the bootstrap distribution.

4. Find trim point to remove endpoint from CI

- To create an x% confidence interval (CI), determine how much data needs to be trimmed from each end of the bootstrap distribution.
- $\frac{1}{2}\left(1-\frac{x}{100}\right)$
- x% is CI
- The remaining 95% of the data between these trim points represents the 95% bootstrap confidence interval for the mean.

CI interpretation

- 95% CI:

- When CI is set to 95%, it means that **if we were to repeat the sampling and CI construction process many times, approximately 95% of the calculated confidence intervals would contain the true population mean.**

- Relation to the Sample Mean:

- The center of the confidence interval aligns with the mean of the original sample (point estimate).

- The CI provides a range around this mean to reflect the uncertainty in estimating the true population mean from the sample data.

1. **A 95% confidence interval does not mean there is a 95% probability that the true parameter lies within the interval.** Instead, it means that if we were to construct CI from an infinite number of independent samples, about 95% of those intervals would contain the true parameter.

2. Instead, it tells us that **variability of the statistic** and provides a way to understand **how confident we should be that our parameter lies within the**

given range.

In [37…

```
fig,ax = plt.subplots(1,1,figsize=(4,3))
sns.barplot(x='is_weekend',y='trip_distance',data=df_taxi);
```

- CI interpretation

1. 95% CI

- The black lines indicate the range within which we are 95% confident that the true mean trip distance lies for each group (weekend vs. non-weekend).
- In other words, if we were to repeatedly take samples and compute confidence intervals, approximately 95% of those intervals would contain the true mean trip distance.
- For example, the interval for “False” (non-weekend) extends roughly from about 2.4 to 3.0 on the y-axis. This suggests that the true average trip distance for non-weekend trips is likely to fall within this range.

2. Comparing Weekends (True) vs. Non-Weekends (False):

- The mean trip distance for weekends (“True”) appears to be slightly higher than for non-weekends (“False”). However, the confidence intervals overlap, which implies that there is not a statistically significant difference between the mean trip distances for weekends and non-weekends.
- Overlapping confidence intervals suggest that the observed difference in means may be due to random variation in the data rather than a true underlying difference.

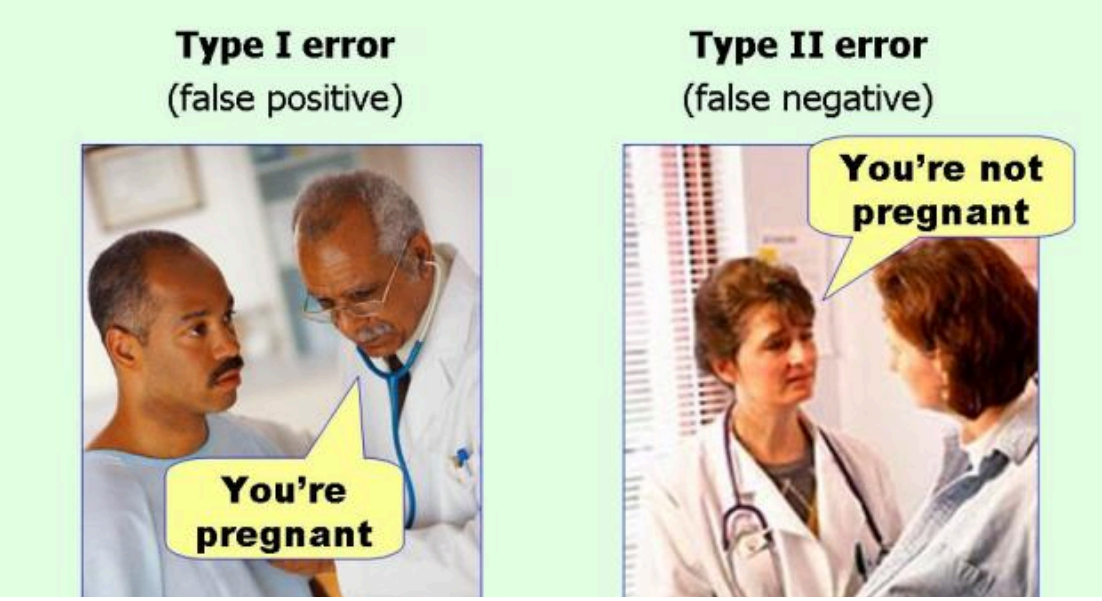
Hypothesis Testing

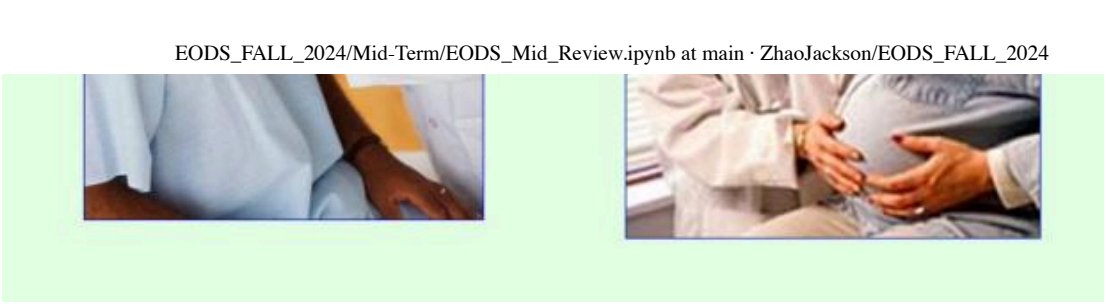
- Hypothesis testing is used to make inferences about a population based on sample data.
- The goal is to determine whether there is enough evidence to reject a null hypothesis (usually a statement of no effect or no difference) in favor of an alternative hypothesis.

- Key Concepts:

    1. Null Hypothesis (H0):
    - the thing we're observing is happening due to random chance (luck)
    - there are no differences between two groups
    2. Alternative Hypothesis (H1 or Ha):
    - The thing we're observing is happening not due to random chance (luck)
    - there is a difference between two groups
    3. Significance Level (α): The threshold for deciding whether to reject the null hypothesis, often set at 0.05.
    - It represents the probability of making a Type I error (rejecting the null hypothesis when it is actually true).
    4. P-value: The probability of obtaining test results at least as extreme as the observed results, under the assumption that the null hypothesis is true.
    - A small p-value (less than α) suggests that the null hypothesis can be rejected.
    5. Test Statistic: A value calculated from the sample data that is used to determine whether to reject the null hypothesis. Common test statistics include the t-statistic, z-statistic, and chi-square statistic.

Errors in Hypothesis Tests





https://flowingdata.com/wp-content/uploads/2014/05/Type-I-and-II-errors1-620x465.jpg

Type I Error (false positive)

- Type I error occurs **when H0 is true, but we incorrectly reject it.**
  - $P\left(\text{reject } H_0 \mid H_0 \text{ true}\right)$ = **Significance** of test or **p-value** (Type I Error)
    - This means that we conclude there is an effect or a difference when, in fact, there isn't one.
    - The probability of making a Type I error is denoted by the significance level, which is typically set at 0.05. This indicates a 5% risk of rejecting the null hypothesis when it is actually true.
    - If a medical test claims a patient has a disease when they are actually healthy, that's a Type I error.

Type II Error (False Negative)

- Type II error occurs when the null hypothesis is false, but we fail to reject it.
- $P\left(\text{reject } H_0 \mid H_1 \text{ true}\right)$ = **Power** of test (Type II Error)
  - **we incorrectly conclude that there is no effect or difference when, in reality, there is one.**
  - The probability of Type II error is denoted by β. The power of the test, which is (1 - β), represents the likelihood of correctly rejecting a false null hypothesis.
  - If a medical test fails to detect a disease that the patient actually has, that's a Type II error.

Permutation Test

- it assesses the significance of an observed effect by comparing the observed data to a distribution of outcomes obtained by randomly rearranging the data labels.
- to test whether the observed data arrangement is not due to luck compared to what might occur by chance.

- permutation test is a non-parametric approach, without replying on normality assumption Instead, it generates an empirical distribution based on the data itself.

why permutation test

1. **Data Flexibility: Numeric or Boolean**

- Permutation tests can be applied to different types of data, such as numeric (e.g., temperature, measurements) or categorical/boolean (e.g., success/failure, conversion rates).
- This flexibility allows permutation tests to be used in various settings without needing to convert data types or rely on specific distributions.

2. **Different Group Sizes**

- Permutation tests do not require equal group sizes, which is often a limitation in some traditional parametric tests.
- we can compare groups of different sizes without any special adjustments, as the permutation process will handle the data accordingly.

3. **No Need for Normality Assumptions**

- Permutation tests are non-parametric, meaning they do not require any specific distributional assumptions.
- With a sufficient number of permutations, the permutation distribution will accurately represent the null hypothesis, regardless of the original data distribution.

- steps of permutation test

1. Combine Groups Together (Assume H_0 is True):
    - Pool all the data together as if there is no difference between the groups.

- This step represents the null hypothesis assumption that any observed effect is due to random chance.

2. Permute (Reorder) Observations:

    - Randomly shuffle the combined dataset to generate a new arrangement of the data. This random reordering helps simulate what the data arrangement might be if H_0 is true.
3. Create New Groups (Same Sizes as Original Groups):

    - Split the shuffled data into groups that match the original group sizes. This step preserves the structure of the experiment while creating new "randomized" group assignments.

randomized "group assignments."

4. Calculate Metric: - Compute the test statistic (e.g., mean difference, correlation, or another measure of interest) for the new groups.

- This step generates a value that represents the effect for the randomly permuted data.

5. Repeat Many Times: - Perform the permutation and metric calculation multiple times (e.g., 1,000 or more).

- Each repetition produces a new test statistic, **creating a distribution of possible outcomes under the null hypothesis**.

6. See Where Our Original Observation Falls in the Distribution of Sample Statistics:

- to compare wether the observed result (original test statistic from the actual data) is typical (more extreme) to the distribution of test statistics generated under the null hypothesis (H_0) through permutations.

- Think of the permutation distribution as simulating the "randomness" of the data under the assumption that there is no real effect.

- If the observed result falls in the tail of this distribution, it indicates that such an outcome is rare under the null hypothesis, thus giving evidence to reject H_0.

7. **If the observed test statistic falls in the tail of the distribution (suggesting a small p-value), it indicates that the observed effect is unlikely to have occurred by chance, and we may reject the null hypothesis.**
    - A low p-value ($p < 0.05$) suggests that the observed test statistic is unlikely to occur if the null hypothesis is true. This provides evidence against H_0, indicating that the observed effect may be real.

    - A high p-value indicates that the observed test statistic is consistent with what could happen by random chance, supporting the null hypothesis.

permutation for coding

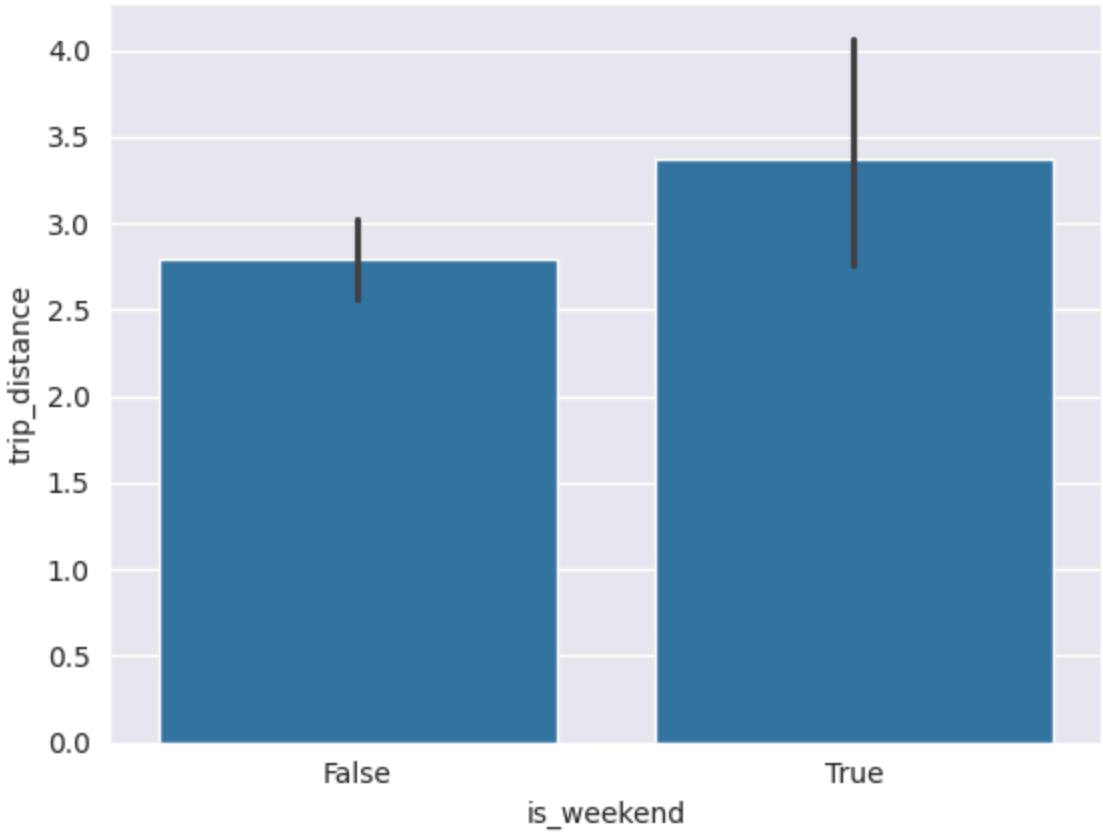
- Reminder of Permutation Test:

0. get group sizes
1. combine groups together
2. permute observations
3. create two new groups (same sizes as originals)
4. calculate metric
5. repeat many times
6. see where our original observation falls

- Question: Is the average trip_distance different on weekdays vs weekends?

In [32…
```
# tqdm gives us a progress bar when looping
from tqdm.notebook import tqdm
```

In [32…
```
sns.barplot(x='is_weekend',y='trip_distance',data=df_taxi);
```



In [32…
```
# Metric: the measure we're interested in
## We're interested in a difference of means: Weekday - Weekend

mean_weekend = df_taxi.loc[df_taxi.is_weekend,'trip_distance'].mean()
mean_weekday = df_taxi.loc[~df_taxi.is_weekend,'trip_distance'].mean()
observed_trip_metric = mean_weekend-mean_weekday
print(f'observed metric: {observed_trip_metric.round(2)}')
```

```
observed metric: 0.58
```

In [32…
```
# 0. get group sizes
n_weekend = df_taxi.is_weekend.sum()
n_weekday = (~df_taxi.is_weekend).sum()
print(f'{n_weekend=} {n_weekday=}')
assert n_weekday + n_weekend == df_taxi.shape[0]
```

```
n_weekend=150 n_weekday=850
```

In [33…
```
# 1. combine groups together (assume H0 is true)
trip_distances = df_taxi.trip_distance
trip_distances[:2]
```

Out[330…

	trip_distance
0	0.89
1	2.70

dtype: float64

In [33…

```
# 2. permute observations
permuted_trip_distances = trip_distances.sample(frac=1,replace=False,ran
permuted_trip_distances[:2]
```

Out[331…

	trip_distance
131	2.13
203	2.15

dtype: float64

In [33…

```
# 3. create new groups

rand_mean_weekend = permuted_trip_distances[:n_weekend].mean()
rand_mean_weekday = permuted_trip_distances[n_weekend:].mean()

# 4. calculate metric

rand_mean_trip_diff = (rand_mean_weekend - rand_mean_weekday)
print('{:.2f}'.format(rand_mean_trip_diff))
```

```
-0.03
```

In [33…

```
# 5. repeat many times

rand_mean_trip_diffs = []
iterations = 10_000

for i in tqdm(range(iterations)):
    permuted_trip_distances = trip_distances.sample(frac=1,replace=False

    rand_mean_weekend = permuted_trip_distances[:n_weekend].mean()
    rand_mean_weekday = permuted_trip_distances[n_weekend:].mean()

    rand_mean_trip_diffs.append(rand_mean_weekend - rand_mean_weekday)

rand_mean_trip_diffs = np.array(rand_mean_trip_diffs) # convert list to

rand_mean_trip_diffs[:5].round(2)
```

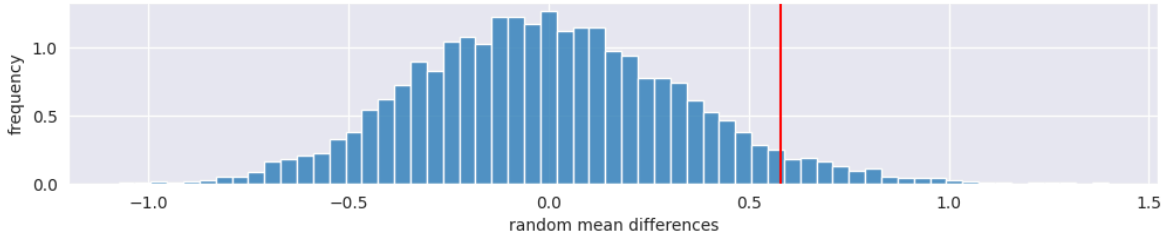
```
  0%|          | 0/10000 [00:00<?, ?it/s]
```

Out[333…

```
array([-0.49, -0.21,  0.58, -0.09, -0.37])
```

In [37…

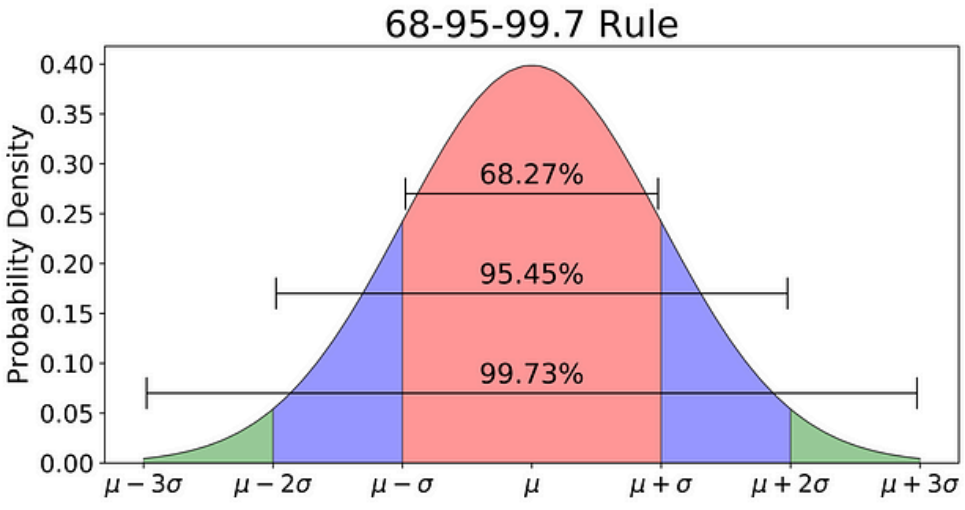
```
# b. see where our original observation falls

fig,ax = plt.subplots(1,1,figsize=(12,2))
ax = sns.histplot(x=rand_mean_trip_diffs, stat='density')
ax.set_xlabel('random mean differences');ax.set_ylabel('frequency');
ax.axvline(observed_trip_metric, color='r');
```



Central Limit Theorem (CLT)

- If all samples are randomly drawn from the same sample population:
- For reasonably large samples (usually $n \geq 30$), the distribution of sample mean $\bar{x}$ is normal regardless of the distribution of X.
- The sampling distribution of $\bar{x}$ becomes approximately normal as the the sample size n gets large.

Properties of Normal Distribution



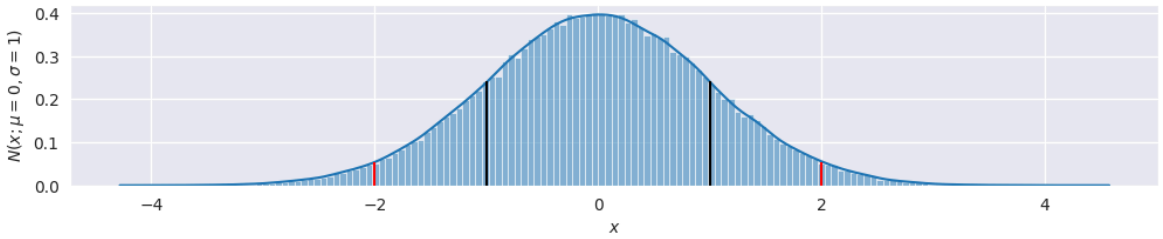
https://towardsdatascience.com/understanding-the-68-95-99-7-rule-for-a-normal-distribution-b7b7cbf760c2

Plotting a Standard Normal Distribution

Plotting a Standard Normal Distribution

In [37…

```
import scipy as sp

x = np.random.normal(0,1,size=100_000)                          # generate
fig,ax = plt.subplots(1,1,figsize=(12,2))
ax = sns.histplot(x=x,stat='density',kde=True);                  # using den
ax.set_xlabel('$x$');ax.set_ylabel('$N(x;\mu=0,\sigma=1)$'); # using lat
ax.vlines([-1,1],0,sp.stats.norm.pdf(1), colors='k');           # 1 standar
ax.vlines([-2,2],0,sp.stats.norm.pdf(2), colors='r');           # 2 standar
```



Normalization: z-score

- Applying normalization (z-score) to the permutation test context can help in comparing the test statistics on a standardized scale.
- applying z-score normalization to the **observed and permuted test statistics** can help interpret the extremeness of the observed statistic on a common scale.

1. Centering Around the Null Hypothesis:

- The permutation distribution generated under H_0 will typically be centered around a mean (often 0) with standard deviation of 1
- By using z-scores, you are effectively centering the test (observed) statistic distribution around this mean and scaling by the standard deviation.

2. Measuring Extremeness:

- A higher absolute z-score implies that the observed test statistic is more extreme relative to the permuted distribution. $\mathcal{Z} = \frac{x-\bar{x}}{s}$

- normalization steps

In [33…

```
rand_mean_trip_diffs_xbar = np.mean(rand_mean_trip_diffs)
rand_mean_trip_diffs_s = np.std(rand_mean_trip_diffs)

rand_mean_trip_zscores = (rand_mean_trip_diffs - rand_mean_trip_diffs_xb
list(zip(rand_mean_trip_diffs[:3].round(2),rand_mean_trip_zscores[:3].ro
```

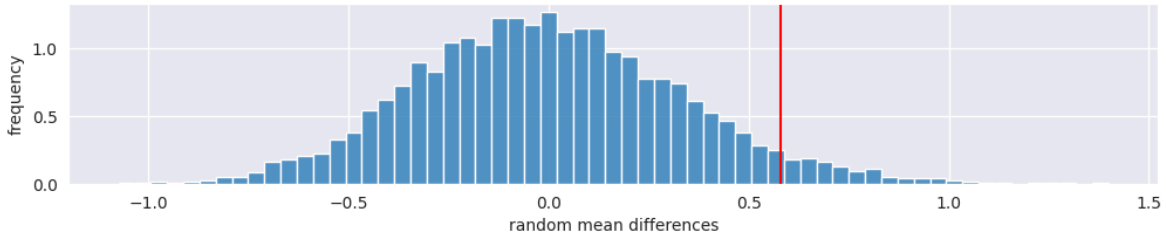
Out[336… `[(-0.49, -1.47), (-0.21, -0.64), (0.58, 1.77)]`

In [33…
```
observed_trip_metric_zscore =  (observed_trip_metric - rand_mean_trip_di
observed_trip_metric.round(2),observed_trip_metric_zscore.round(2)
```

Out[337… (0.58, 1.76)

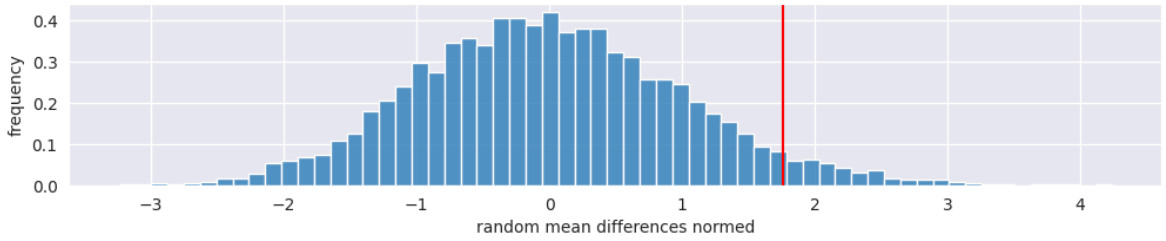
- previously on distribution

In [34…
```
# 6. see where our original observation falls

fig,ax = plt.subplots(1,1,figsize=(12,2))
ax = sns.histplot(x=rand_mean_trip_diffs, stat='density')
ax.set_xlabel('random mean differences');ax.set_ylabel('frequency');
ax.axvline(observed_trip_metric, color='r');
```



- current distribution

In [33…
```
# 6. see where our original observation falls (normalized)
fig,ax = plt.subplots(1,1,figsize=(12,2))
ax = sns.histplot(rand_mean_trip_zscores, stat='density')
ax.set_xlabel('random mean differences normed');ax.set_ylabel('frequency
ax.axvline(observed_trip_metric_zscore,color='r');
```



A/B Test

- A/B test is used to compare two versions (A and B) of a variable (such as a web page, product, or marketing campaign) to determine which one performs better.
- The goal is to identify the version that yields the most favorable outcome based on a specific metric (e.g., conversion rate, click-through rate, revenue).

- Ex: Webpages and Sales

- Question: Which webpage leads to more sales?
- Potential Issue: what if sales are large but infrequent?
- **Proxy Variable:** stand in for true value of interest
- Ex: Assume 'time on page' is correlated with sales

In [34…

```
session_times = pd.read_csv('/content/web_page_data.csv')
print(session_times.shape)

session_times.head(3)
```

```
(36, 2)
```

Out[342…

	Page	Time
0	Page A	12.6
1	Page B	151.7
2	Page A	21.0

In [34…

```
fig,ax = plt.subplots(1,3,figsize=(12,4),sharey=True)
sns.boxplot(x='Page',y='Time',data=session_times,ax=ax[0]);
sns.swarmplot(x='Page',y='Time',data=session_times,ax=ax[1]);
sns.barplot(x='Page',y='Time',data=session_times,ax=ax[2]);
```

- Define a metric

In [34…

```
# Ex: Webpages and Sales, Define the Metric
## We're interested in a difference of means (Page A - Page B)

mean_a = session_times.loc[session_times.Page == 'Page A','Time'].mean()
mean_b = session_times[session_times.Page == 'Page B'].Time.mean()
observed_ad_metric = mean_a-mean_b
print('observed metric: {:0.2f}'.format(observed_ad_metric))
```

```
observed metric: -49.77
```

- Website and Sales, Permutation Test

- Websites and Sales, Permutation Test

In [34…
```
# 0. get group sizes
n_a = (session_times.Page == 'Page A').sum()
n_b = session_times.shape[0] - n_a
print(f'{n_a=} {n_b=}')
```

```
n_a=21 n_b=15
```

In [34…
```
# 1. combine groups together (assume H0 is true)
session_times.Time[:2]
```

Out[346…

	Time
0	12.6
1	151.7

dtype: float64

In [34…
```
# 2. permute observations
session_times_permuted = session_times.Time.sample(frac=1,replace=False,
session_times_permuted[:2]
```

Out[347…

	Time
6	50.5
8	79.2

dtype: float64

In [34…
```
# 3. create new groups
rand_mean_a = session_times_permuted[:n_a].mean()
rand_mean_b = session_times_permuted[n_a:].mean()

# 4. calculate metric
rand_mean_ad_diff = (rand_mean_a - rand_mean_b)
print('{:.2f}'.format(rand_mean_ad_diff))
```

```
11.89
```

In [34…
```
# 5. repeat many times
rand_mean_ad_diffs = []
iterations = 10_000

for i in tqdm(range(iterations)):
    session_times_permuted = session_times.Time.sample(frac=1,replace=Fa

    rand_mean_a = session_times_permuted.iloc[:n_a].mean()
    rand_mean_b = session_times_permuted.iloc[n_a:].mean()

    rand_mean_ad_diffs.append(rand_mean_a - rand_mean_b)
```

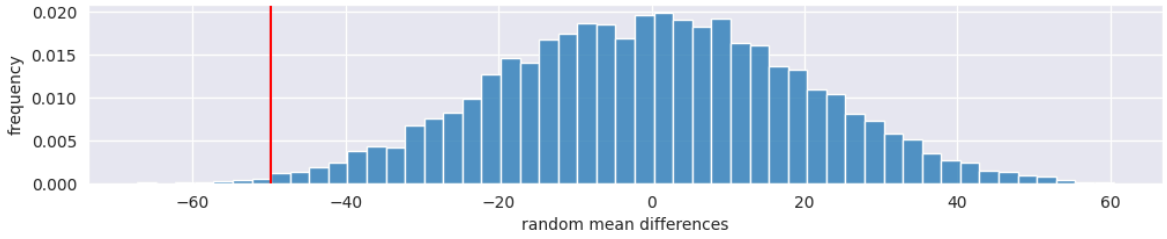
```
rand_mean_ad_diffs = np.array(rand_mean_ad_diffs)
rand_mean_ad_diffs[:5].round(2)
```

```
0%|          | 0/10000 [00:00<?, ?it/s]
```

Out[349… `array([  9.79, -13.71, -15.83,  34.99,  -4.67])`

In [35…

```
# 6. see where our original observation falls
fig,ax = plt.subplots(1,1,figsize=(12,2))
ax = sns.histplot(x=rand_mean_ad_diffs, stat='density')
ax.set_xlabel('random mean differences');ax.set_ylabel('frequency');
ax.axvline(observed_ad_metric, color='r');
```



In [35…

```
# Normalize our values
rand_mean_ad_diffs_xbar = np.mean(rand_mean_ad_diffs)
rand_mean_ad_diffs_s    = np.std(rand_mean_ad_diffs)

rand_mean_ad_zscores = (rand_mean_ad_diffs - rand_mean_ad_diffs_xbar) /
list(zip(rand_mean_ad_diffs[:3].round(2),rand_mean_ad_zscores[:3].round(
```

Out[352… `[(9.79, 0.5), (-13.71, -0.69), (-15.83, -0.8)]`

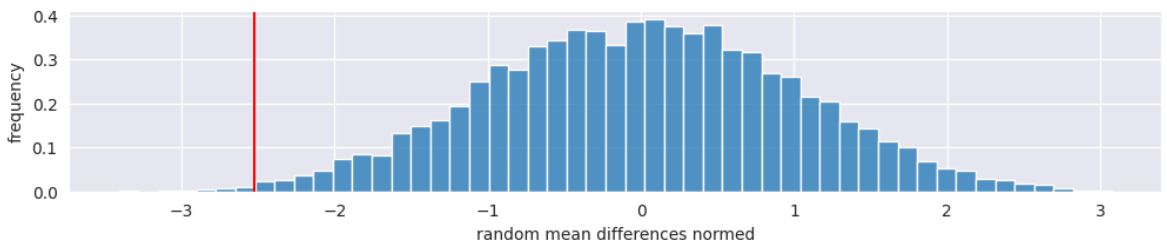
In [35…

```
observed_ad_metric_zscore =  (observed_ad_metric - rand_mean_ad_diffs_xb
observed_ad_metric.round(2),observed_ad_metric_zscore.round(2)
```

Out[353… `(-49.77, -2.52)`

In [35…

```
# 6. see where our original observation falls (normalized)
fig,ax = plt.subplots(1,1,figsize=(12,2))
ax = sns.histplot(rand_mean_ad_zscores, stat='density')
ax.set_xlabel('random mean differences normed');ax.set_ylabel('frequency
ax.axvline(observed_ad_metric_zscore,color='r');
```



One-Tailed vs Two-Tailed Tests

- One-Tailed Test
- alternative hypothesis specifies a direction of the effect (e.g., greater than or less than a certain value).
- if you are testing whether a new drug increases recovery rate compared to a placebo, you would use a one-tailed test for “greater than” (positive effect).

- Two-Tailed Test
- alternative hypothesis does not specify a direction, meaning it tests for the possibility of an effect in either direction.
- If you are testing for any significant difference, regardless of the direction, such as testing whether a new teaching method affects test scores (either increases or decreases them).

Significance Level Allocation:

- In a one-tailed test, the entire α is placed on one tail of the distribution.
  - For example, if α = 0.05 , all 5% of the probability is on one side.
- In a two-tailed test, the α is split between both tails, with 2.5% on the upper tail and 2.5% on the lower tail for an α = 0.05 test.
  - This makes it more difficult to reject the null hypothesis, as you need a larger effect size for the test statistic to reach the critical value in either tail.

- two-tailed test

In [35…
```
# find absolute values greater than our observed_metric
ad_gt = np.abs(rand_mean_ad_diffs) >= np.abs(observed_ad_metric)

# how many are greater than or equal to?
num_ad_gt = ad_gt.sum()

# proportion of total that are as or more extreme
p = num_ad_gt / len(rand_mean_ad_diffs)
print(f'{p = :}')
```

```
p = 0.0078
```

- one-tailed test

In [35…
```
# one-tailed test
sum(np.array(rand_mean_ad_diffs) <= observed_ad_metric) / len(rand_mean_
```

Out[356…
```
0.0037
```

- one-tailed shows much lower p-value than 2 tailed.

Choosing the significance level (α)

What is significance level

- The threshold used to compare with the p-value.
- It represents the probability of rejecting the null hypothesis (H_0) when it is actually true, leading to a Type I error.

Common values

- 0.1 (10%): There is a 10% chance of making a Type I error. This level is sometimes used in exploratory studies where some risk of error is acceptable to detect potential effects.
- 0.05 (5%): The most commonly used value. There is a 5% chance of making a Type I error, which is considered a reasonable balance between sensitivity and specificity. It's standard in many scientific fields.
- 0.01 (1%): A stricter threshold, used when the consequences of a Type I error are severe (e.g., medical research, high-stakes financial decisions). There is a 1% chance of making a Type I error.

- Balancing Type I and Type II Errors:

1. **Reducing α lowers the risk of a Type I error** but **increases the risk of a Type II error** (failing to reject a false null hypothesis)
2. **Power of the Test:** Choosing a **lower α means the test requires a stronger effect to reject the null hypothesis**, which could reduce the test's power to detect smaller but true effects.

- Question: Does price A lead to higher conversions than price B
- **Conversion:** Turning a visit into a sale
- H_0: conversions for Price A ≤ conversions for Price B
- Price A does not lead to more conversions
- H_1: conversions for Price A > conversions for Price B

- Price A leads to more conversions

In [35…
```
# Counts of observations
df = pd.DataFrame({'Price A':[200,23539],
                   'Price B':[182,22406]},
                   index=['Conversion','No Conversion'])
df
```

Out[357…

	Price A	Price B
Conversion	200	182
No Conversion	23539	22406

- Metric: difference in percent conversion

In [35…
```
pct_conv = df.loc['Conversion'] / df.sum(axis=0) * 100
pct_conv.round(2)
```

Out[358…

	0
Price A	0.84
Price B	0.81

dtype: float64

In [35…
```
diff_pct_conv = pct_conv['Price A'] - pct_conv['Price B']
print(f'{diff_pct_conv.round(3)}%')
```

```
0.037%
```

In [36…
```
# get the total number of samples
n = df.sum().sum()
n
```

Out[360… 46327

In [36…
```
# get total conversion sample numbers
n_conversion = df.loc['Conversion'].sum()
n_conversion
```

Out[361… 382

In [36…
```
conv_samples = np.zeros(n)
conv_samples[:n_conversion] = 1

assert sum(conv_samples) == n_conversion
```

In [36…
```
conv_samples
```

Out[363… array([1., 1., 1., ..., 0., 0., 0.])

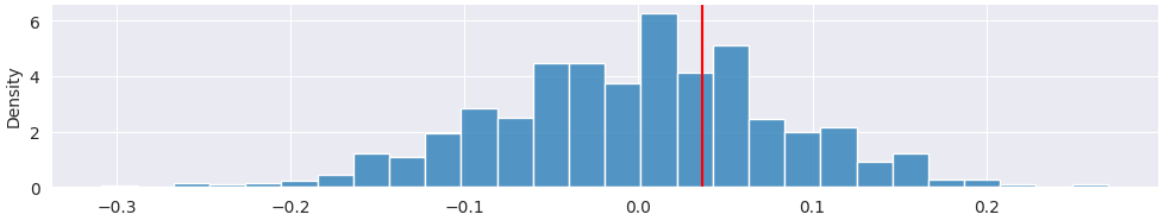
In [36…
```
n_pricea, n_priceb = df.sum(axis=0)
print(f'{n_pricea=} {n_priceb=} {n=}')

assert n_pricea + n_priceb == n
```

```
n_pricea=23739 n_priceb=22588 n=46327
```

In [36…
```
np.random.seed(123)
rand_conv_diffs = []
for i in tqdm(range(1000)):
    conv_permuted = np.random.permutation(conv_samples)
    rand_conv_a = sum(conv_permuted[:n_pricea]) / n_pricea
    rand_conv_b = sum(conv_permuted[n_pricea:]) / n_priceb
    rand_conv_diffs.append(100 * (rand_conv_a - rand_conv_b))
```

```
0%|          | 0/1000 [00:00<?, ?it/s]
```

In [36…
```
fig,ax = plt.subplots(1,1,figsize=(12,2))
ax = sns.histplot(x=rand_conv_diffs, stat='density')
ax.axvline(diff_pct_conv,color='r');
```



In [36…
```
# calculate a two-tailed p-value
sum(np.array(rand_conv_diffs) >= diff_pct_conv) / len(rand_conv_diffs)
```

Out[368… 0.35

In [36…
```
# Equation based proportion test
from statsmodels.stats.proportion import proportions_ztest

z,p = proportions_ztest(df.loc['Conversion'].values,
                        df.sum(),
                        alternative='two-sided')
print(f'{p = :0.3f}')
```

```
p = 0.662
```

Multi-Armed Bandit

- It involves choosing between several options (or “arms”) to maximize a cumulative reward.

1. **Arms and Rewards:**
    - with multiple arms, each representing a different choice or action.
    - Each arm provides a reward, but the probability distribution of these rewards is unknown. The goal is to maximize the cumulative reward by choosing the best arms over time.
2. **Exploration vs. Exploitation Trade-off:**
    - **Exploration** involves trying different arms to gather information about their reward distributions. This helps in identifying which arms are more promising.
    - **Exploitation** involves choosing the arm with the highest known average reward to maximize the immediate gain.
3. The challenge in a multi-armed bandit problem is to **balance exploration and exploitation effectively**.

- If you exploit too early, you may miss out on better rewards from other arms
- If you explore too much, you might not take advantage of the best-known option.

Exploration Vs Exploitation

- **Exploration:** There might be a better arm
- keep choosing different arms randomly

- **Exploitation:** We want to make use of the best
- keep pulling the best arm

Greedy Algorithm for MAB

1. **Choose a Small Epsilon (ϵ):**
    - Select a small value for ϵ, such as 0.1 or 0.05.
- This parameter determines the frequency of exploration.
- A higher value means more exploration
- A lower value means more exploitation.

2. Generate a **Random Number** Between 0 and 1:
    - It determines whether the algorithm explores or exploits.
3. Decision Based on ϵ:

- **random number < ϵ:**
- The algorithm **explores** by selecting an arm at random.
- This step helps gather information about the potential rewards of different arms, even if some arms have not performed well in the past.
- **random number >= ϵ:**
- The algorithm **exploits** by choosing the arm with the highest estimated average reward.
- This maximizes the immediate reward based on the current information.

4. Repeat the Process:

- The algorithm continually repeats this process, gradually improving its estimate of each arm's reward.

Ground Truth in MAB

- **ground truth**: the true, unknown probability distribution of rewards for each arm. Each arm has a fixed probability of giving a reward, but the algorithm does not know these probabilities in advance.

How the ϵ-Greedy Algorithm Approximates the Ground Truth

1. Exploration Phase (randomly choosing arms): - when the random number < ϵ, the algorithm selects arms at random. - This exploration is essential for gathering information about each arm's actual reward distribution, thus helping the algorithm form better estimates of the ground truth probabilities. - Over time, as different arms are tried, the algorithm collects data that reflects the true reward rates of each arm, which is then used to update the estimated rewards.

2. Exploitation Phase (choosing the best-known arm):

- When random number ≥ ϵ, the algorithm selects the arm with the highest known average reward based on the current estimates.
- If the estimates are close to the ground truth, then the algorithm will consistently pick the optimal arm (the one with the highest true reward probability). If the estimates deviate from the ground truth, then the algorithm may pick suboptimal arms more often.

3. Convergence to Ground Truth:

- Over many iterations, if ϵ is well-chosen, the algorithm will gather enough information to make its estimates of each arm's reward distribution converge closer to the ground truth.
- However, since the ϵ-greedy algorithm uses a fixed probability for exploration, there is no guarantee that the estimates will always perfectly match the ground truth, especially if ϵ is too high (leading to too much

exploration) or too low (leading to insufficient exploration).

Example MAB

- Imagine you have three slot machines (arms) with unknown probabilities of paying out. Initially, the algorithm has no information about which machine is better. Using the ϵ-greedy algorithm with $\epsilon = 0.1$, the following could happen:

  1. Iteration 1: A random number is generated (e.g., 0.05), which is less than ϵ (0.1). The algorithm explores by choosing an arm at random.
  2. Iteration 2: A new random number (e.g., 0.15) is greater than ϵ, so the algorithm chooses the arm with the highest known average reward so far.
  3. Iteration 3: Again, a random number is generated, and the process repeats.

Machine Learning Model

- it use models to training input data for output prediction or interpretation

In [37…

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('darkgrid')

%matplotlib inline
```

In [37…

```
df_wine = pd.read_csv('/content/wine_dataset.csv',usecols=['alcohol','as
df_wine.sample(4,random_state=1)
```

Out[378…

	alcohol	ash	hue	proline	class
161	13.69	2.54	0.96	680.0	2
117	12.42	2.19	1.06	345.0	1
19	13.64	2.56	0.96	845.0	0
69	12.21	1.75	1.28	718.0	1

- we can use data to performance various tasks

1. **Classification:** Can we predict *categorical target/label* "class" from the other columns?
2. **Regression**: Can we predict *numeric target/label* "hue" from the other columns?
3. **Feature Selection**: What are the *important features* when predicting "hue"?
4. **Interpretation**: Can a model tell us about how the features and target/label

interact?

5. **Clustering**: Do the observations group together in feature space?

Data vocal for ML

1. X, features, attributes, independent/exogenous/explanatory variables

- Ex: alcohol, trip_distance, company_industry

2. y, target, label, outcome, dependent/endogenous/response variables

- Ex: class, hue, tip_amount, stock_price

4. $f(X) \rightarrow y$, Model that maps features X to target y

Variations of ML tasks

Supervised Learning Framework: **Model is trained on labeled data.**

- The goal is to learn a mapping from input features to a target output (label), using a dataset consisting of input-output pairs.
- The model makes predictions based on the training data, and its performance is evaluated using a known set of outcomes (labels).

    1. **Classification**: Where the model learns to predict a *categorical label* (e.g., whether an email is spam or not).
    2. **Regression**: Where the model predicts a *continuous value* (e.g., predicting house prices based on various features).
    3. Common Algorithms:
        - Linear Regression
        - Logistic Regression
        - Support Vector Machines (SVM)
        - Random Forests
        - Neural Networks
    4. Applications:
        - Spam detection
        - Image recognition
        - Sentiment analysis

Unsupervised Learning: **model is trained on unlabeled data**

- The goal is to discover hidden patterns, relationships, or groupings within the data, without any prior knowledge of the outcomes.
- The model uses unlabeled data to find patterns or groupings.
    1. **Clustering**: The model groups data into *clusters based on similarities* (e.g., segmenting customers into different groups based on purchasing behavior).
    2. **Dimensionality Reduction**: Reducing the number of features in the data while preserving as much information as possible (e.g., using Principal Component Analysis (PCA)).
    3. Common Algorithms:
        - k-Means Clustering
        - Hierarchical Clustering
        - PCA (Principal Component Analysis)
    4. Applications:
        - Market segmentation
        - Anomaly detection
        - Data compression

Prediction VS Interpretation

1. **Prediction**

- Ability of a model to make accurate prediction based on the training data.
- The focus is on using patterns in the data to predict an outcome.
- The goal is to minimize errors or maximize accuracy when making predictions on new, unseen data.
- **Algorithms with Strong Predictive Power:**
    1. Neural Networks
    2. Random Forests
    3. Gradient Boosting Machines

2. **Interpretation**

- It emphasizes making the model's decision-making process transparent and explaining the **relationships between the input features and the predictions**.
- This is particularly important when the model's results need to be trusted.
- The goal is to use model to **explain** impact of each feature on the outcome.
- **Algorithms with Strong Interpretability:**

- **Algorithms with Strong Interpretability:**

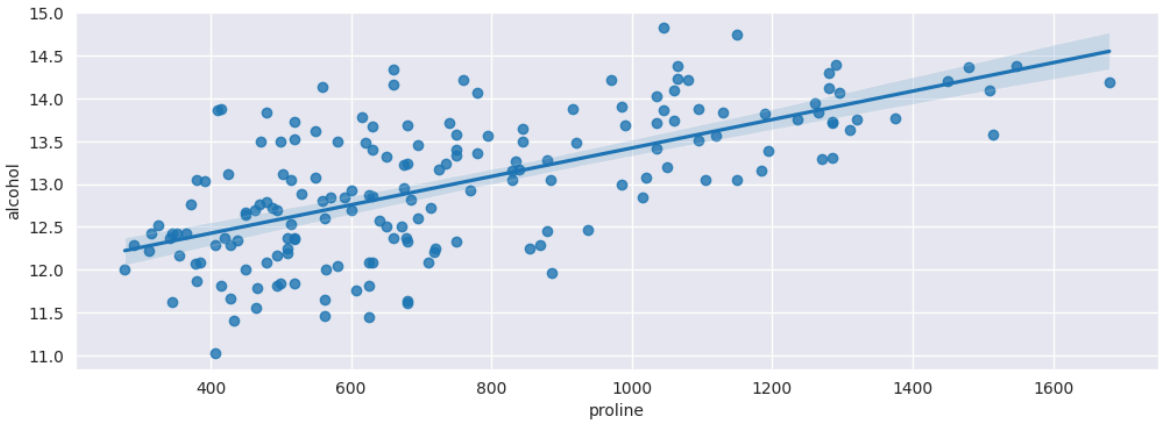
  1. Linear Regression
  2. Logistic Regression
  3. Decision Trees

Linear Regression Model

- What is the relationship between 'proline' (an amino-acid) and 'alcohol' in wine?

In [38…

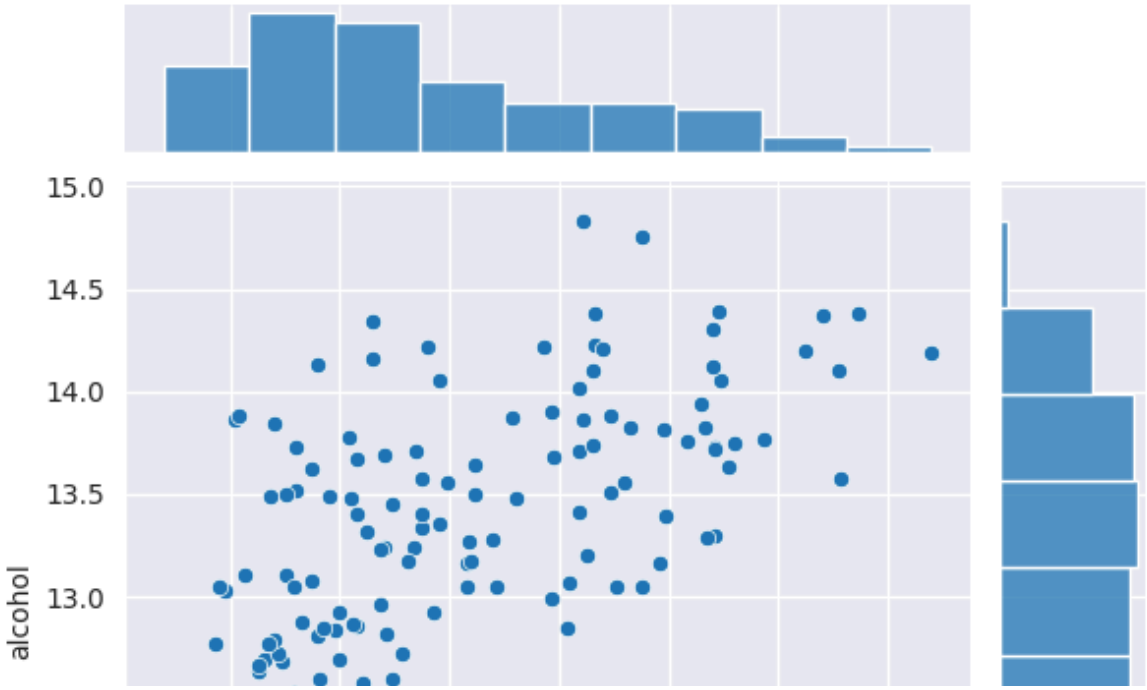
```
fig,ax = plt.subplots(1,1,figsize= (12, 4))
sns.regplot(x='proline', y='alcohol', data=df_wine);
```

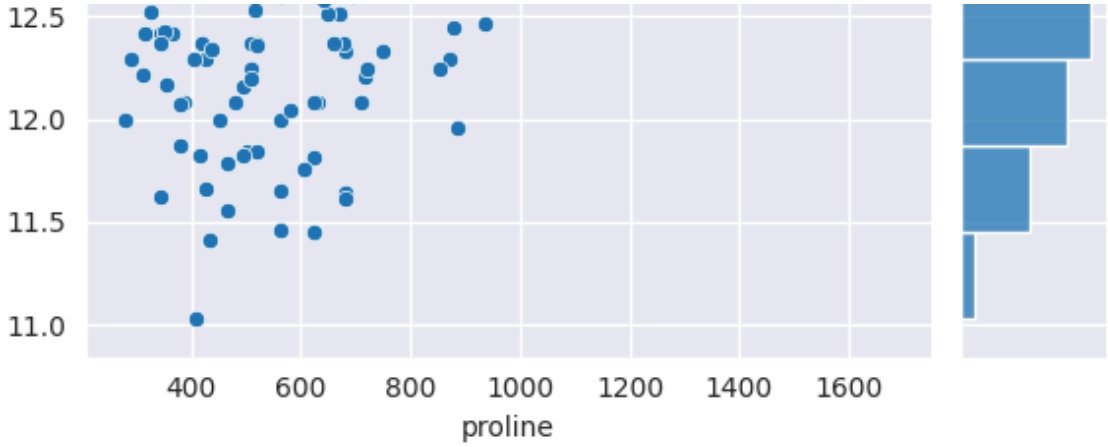


- are proline and alcohol correlated?

In [38…

```
sns.jointplot(x='proline',y='alcohol',data=df_wine);
```





Correlation VS Causation

Correlation

- The coefficients in a linear regression model reflect the correlation between each independent variable and the dependent variable. However, these coefficients should not be interpreted as evidence of causality.
- High correlation between variables can improve the predictive power of the model, but this does not mean one variable causes the other.
- Key Characteristics:
    1. Correlation measures the strength and direction of a linear relationship between two variables.
    2. A positive correlation indicates that as one variable increases, the other also tends to increase.
    3. A negative correlation means that as one variable increases, the other tends to decrease.
    4. Correlation does not imply that changes in one variable cause changes in another.

Causation

- Change in one variable directly results in a change in another.
- Linear regression models show relationships between variables, but they do not inherently demonstrate that one variable causes another.
- Regression analysis typically reveals correlation, but not causation.
- Key Characteristics:
    1. Causality indicates that one variable directly influences the other.
    2. Establishing causality often requires **controlled experiments or careful**

consideration of confounding factors.

3. Causality cannot be concluded from correlation alone, and **linear regression on its own cannot prove causality.**

In [38…

```
# Correlation

from scipy.stats import pearsonr
r,p = pearsonr(df_wine.proline,df_wine.alcohol)
print(f'r: {r:.2f}, p: {p:.2f}')
```

```
r: 0.64, p: 0.00
```

Simple Linear Regression

$$y_i = w_1 x_i + w_0 + \varepsilon_i$$

- y_i : dependent/endogenous/response target, label (Ex: `alcohol`)
- x_i : independent/exogenous/explanatory feature, attribute (Ex: `proline`)
- w_1 : coefficient, slope
- w_0 : coefficient, bias term, intercept
- ε_i : error, hopefully small, often assumed $\sim \mathcal{N}(0, 1)$
- Want to find values for w_1 and w_0 that best fit the data.
- Find a line as close to our observations as possible

Ordinary Least Squares (OLS)

- The goal is to minimize the SSE between the observed data points and the predicted values from the regression line.
- The OLS method **fits a line that minimizes the total squared difference between the observed values y and the predicted values $\hat{y}$**.
  - It is a straightforward, closed-form solution, meaning we can calculate w_0 and w_1 directly **without iterative methods**.
- **prediction**: $\hat{y}_i = f(x_i) = w_1 x_i + w_0$
- **error**: $error(y_i, \hat{y}_i) = y_i - \hat{y}_i$
- **sum of squared errors**: $\sum_{i=1:n} \left(y_i - \hat{y}_i\right)^2$
- **least squares**: make the sum of squared errors as small as possible

Gradient Descent

- An **iterative method** used to to minimize the squared error in regression by adjusting the model parameters (w_0 , w_1) to reduce the error.
- Gradient: **A vector of partial derivatives** that points in the **direction of the steepest increase or decrease**
- Process:
    1. The goal is to **minimize the error by taking steps in the opposite direction of the gradient**.
    2. **Maximization problems follow the gradient's direction** to increase the objective function.
    3. **Minimization problems follow the opposite direction** of the gradient to decrease the objective function (e.g., minimize squared error).
- Maximums and Minimums:
    1. **Global Maximum/Minimum**: The absolute best solution over the entire dataset.
    2. **Local Maximum/Minimum**: The best solution within a specific neighborhood but not necessarily the global best.

- Simple Regression by scikit-learn

In [38…
```
# import the model from sklearn
from sklearn.linear_model import LinearRegression
```

In [38…
```
# instantiate the model and set hyperparameters
lr = LinearRegression(fit_intercept=True)    # by default
```

In [38…
```
# fit the model
lr.fit(X=df_wine.proline.values.reshape(-1, 1), y=df_wine.alcohol);
```

In [38…
```
# display learned coefficients (trailing underscore indicates learned va
print(lr.coef_.round(4))
print(lr.intercept_.round(2))
```

```
[0.0017]
11.76
```

In [38…
```
# predict given new values for proline
X = np.array([1000,2000]).reshape(-1,1)
lr.predict(X).round(2)
```

Out[389… array([13.42, 15.08])

Purpose of reshape(-1, 1)

- scikit-learn's linear regression model **expects the input feature matrix X to be in a 2D format**.
- In this case, X needs to be shaped as a **column vector** with each row representing an observation, and each column representing a feature.
- array [1000, 2000] is a 1D array
- By .reshape(-1, 1):
    1. -1 tells NumPy to automatically infer from the number of rows based on the length of the array (in this case, 2 rows).
    2. 1 specifies that there should be exactly 1 column, making it a column vector.

In [39…
```
X = np.array([1000, 2000])
X
```

Out[392… array([1000, 2000])

In [39…
```
X = np.array([1000, 2000]).reshape(-1, 1)
X
```

Out[391…
```
array([[1000],
       [2000]])
```

In [39…
```
df_wine.proline.values[:5]
```

Out[393… array([1065., 1050., 1185., 1480., 735.])

In [39…
```
df_wine.proline.values.shape
```

Out[394… (178,)

In [39…
```
df_wine.proline.values.reshape(-1,1).shape # -1 means "infer from the da
```

Out[395… (178, 1)

In [39…
```
# Alternatively, use loc to make a 2-D dataframe

df_wine.loc[:,['proline']].shape
```

Out[396… (178, 1)

In [39…
```
# Alternatively, use [[]] to convert to 2-D dataframe
df_wine[['proline']].shape
```

Out[397… (178, 1)

Coefficients interpretation

- On average, the amount of change in the dependent variable Y for a one-unit change in the independent variable X, while holding all other variables constant.

In [39…
```
print(f'w_1 = {lr.coef_[0]:0.3f}, w_0 = {lr.intercept_:0.3f}')
```

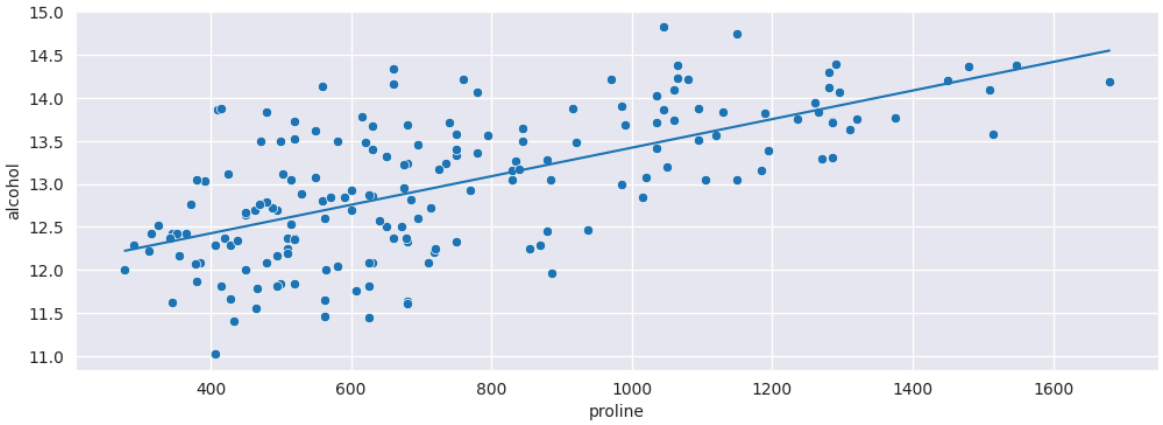
```
w_1 = 0.002, w_0 = 11.761
```

In [39…
```
print(f'alcohol = {lr.coef_[0]:0.3f}*proline + {lr.intercept_:0.3f}')
```

```
alcohol = 0.002*proline + 11.761
```

- On average, for each additional proline added, the level of alcohol increases by 0.002, assuming all other variables are held constant.
- When proline is 0, alcohol is 11.761

In [40…
```
# Plot

x_predict = [df_wine.proline.min(),df_wine.proline.max()]
y_hat = lr.predict(np.array(x_predict).reshape(-1,1))

fig,ax = plt.subplots(1,1,figsize=(12,4))
ax = sns.scatterplot(x=df_wine.proline,y=df_wine.alcohol);
ax.plot(x_predict,y_hat);
```



Multiple Linear Regression

- Multiple linear regression extends simple linear regression by modeling the

relationship between a dependent variable and two or more independent variables.

$$y_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \ldots + w_m x_{im} + \varepsilon_i$$

Ex:

$$\text{alcohol}_i = w_0 + w_{\text{proline}} \text{proline}_i + w_{\text{hue}} \text{hue}_i$$

- Objective: Find a *plane* that falls as close to our points as possible

In [40…

```
mlr = LinearRegression()
mlr.fit(df_wine[['proline','hue']],y=df_wine.alcohol);

print(f'{"intercept":10s} : {mlr.intercept_:0.3f}')
for (name,coef) in zip(['proline','hue'],mlr.coef_):
    print(f'{name:10s} : {coef: 0.3f}')
```

```
intercept  : 12.459
proline    :  0.002
hue        : -0.842
```

Multiple linear regression interpretation

- **One average**, each coefficient represents the change in the dependent variable for a one-unit change in the corresponding independent variable, **holding all other variables constant.**
- The intercept (w_0) represents the predicted value of the dependent variable when all independent variables are equal to zero.

In [40…

```
import statsmodels.api as sm

X = df_wine[['proline','hue']].copy()
X = sm.add_constant(X) # or X['const'] = 1
y = df_wine.alcohol
sm_mlr = sm.OLS(y,X).fit() # Note: X,y passed as parameters to object, n
print(sm_mlr.summary())
```

```
                            OLS Regression Results
==============================================================================
====
Dep. Variable:                alcohol   R-squared:
0.467
Model:                            OLS   Adj. R-squared:
0.461
Method:                 Least Squares   F-statistic:                     7
6.79
Date:                Thu, 17 Oct 2024   Prob (F-statistic):           1.15
e-24
Time:                        16:50:54   Log-Likelihood:                -15
8.89
No. Observations:                 178   AIC:                             3
23.8
```

```
Df Residuals:                      175   BIC:                             3
33.3
Df Model:                            2
Covariance Type:            nonrobust
==============================================================================
====
                 coef    std err          t      P>|t|      [0.025      0.
975]
------------------------------------------------------------------------------
----
const         12.4593      0.203     61.347      0.000      12.058      1
2.860
proline        0.0018      0.000     12.325      0.000       0.002
0.002
hue           -0.8418      0.202     -4.175      0.000      -1.240      -
0.444
==============================================================================
====
Omnibus:                        0.751   Durbin-Watson:
1.734
Prob(Omnibus):                  0.687   Jarque-Bera (JB):
0.606
Skew:                           0.142   Prob(JB):
0.739
Kurtosis:                       3.028   Cond. No.                     4.96
e+03
==============================================================================
====

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is cor
rectly specified.
[2] The condition number is large, 4.96e+03. This might indicate that ther
e are
strong multicollinearity or other numerical problems.
```

Adding a constant (intercept)

- Adding a constant (intercept term) works by including a term in the regression equation that accounts for the **baseline level of the dependent variable** when all the independent variables are set to zero
- **Improve Model Fit:**
  - Including a constant term allows the model to better fit the data by adjusting for the baseline level of the dependent variable.
- **Interpretation of Coefficients:**
- When a constant is included, each regression coefficient indicates the change in the dependent variable for a one-unit change in the corresponding independent variable, while holding other variables constant.
- **Avoiding Bias:**
- Without a constant term, the model would be forced to pass through the origin

- Two ways of keeping track of the bias term

1. Keep it as a separate parameter (sklearn) :

- $y = w_0 + w_1x_1 + w_2x_2 + \ldots + w_mx_m$
- $y = w_0 + \sum_{i=1}^{m} w_ix_i$

2. Append a constant of $x_0 = 1$ so x and w are the same length (statsmodels) :

- $y = w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_mx_m$
- $y = \sum_{i=0}^{m} w_ix_i$

In [40…

```
X.head(3)
```

Out[407…

	const	proline	hue
0	1.0	1065.0	1.04
1	1.0	1050.0	1.05
2	1.0	1185.0	1.03

Adjusted R-Squared

- R-Squared:
- R-squared measures the proportion of variance in the DV explained by IV.
- It ranges from 0 to 1, where a higher value indicates a better fit.
- Adjusted R-Squared:
- It adjusts for the potential inflation of R-squared when **adding more variables**, whether they significantly contribute to explaining the variability or not.
- Adding a constant (intercept term) leads to a higher R-squared value because the model can now capture the variance around the mean of the dependent variable more effectively.

Normalize Multiple Linear Regression with Z-score

In [40…

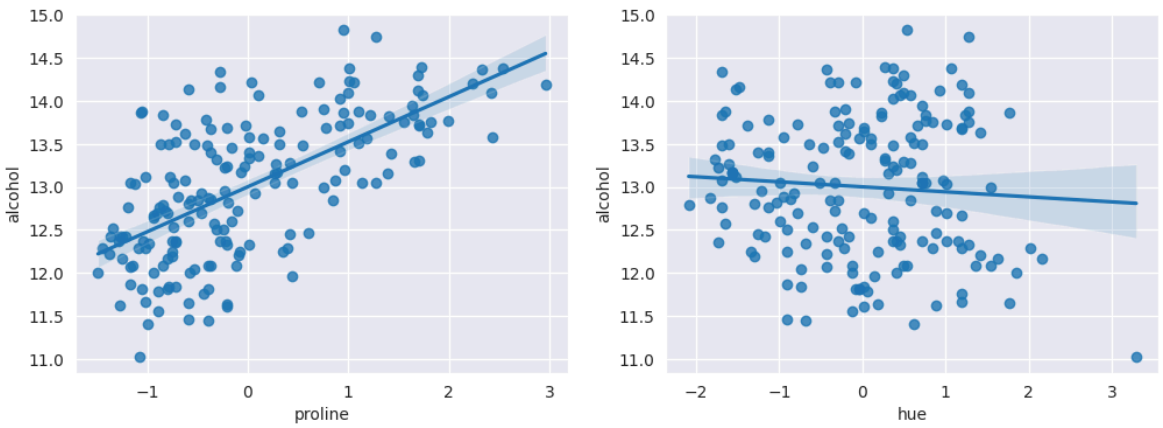
```
X_zscore = df_wine[['proline','hue']].apply(lambda x: (x-x.mean())/x.std

mlr_n = LinearRegression()
mlr_n.fit(X_zscore, df_wine.alcohol)
for (name,coef) in zip(X_zscore.columns,mlr_n.coef_):
    print(f'{name:10s} : {coef: 0.3f}')
```

```
proline    :  0.568
```

```
hue          : -0.192
```

In [40…

```
fig,ax = plt.subplots(1,2,figsize=(12,4))
sns.regplot(x=X_zscore.proline,y=df_wine.alcohol,ax=ax[0]); # axis = [0]
sns.regplot(x=X_zscore.hue,y=df_wine.alcohol,ax=ax[1]); # axis = [1] ind
```



- Purpose of normalization

1. normalizing the features in multiple linear regression using Z-score normalization to have scaling mean of 0 and a standard deviation of 1.
2. This is useful when features are on different scales, as it ensures that all features contribute equally to the model.

- Steps

1. Normalization: $(X - X.mean())/X.std()$ computes the Z-score for each feature, transforming the data so that each feature has a mean of 0 and a standard deviation of 1.
2. $z = \frac{x - \bar{x}}{s}$

Colinearity in Multiple Linear Regression

- Assumption of MLR: Linear independence
- no feature can be expressed as a weighted sum of the others.
- Colinearity Issue
- When features are linearly dependent, the model struggles to estimate the coefficients (w).
    1. The model cannot uniquely attribute the impact on the target variable to any one feature.
    2. Collinearity leads to unstable coefficient estimates, where small changes in the data can result in large changes in the estimated coefficients.

In [41…

```
df_wine.corr().style.background_gradient().format('{:.2f}')
```

Out[410…

	alcohol	ash	hue	proline	class
alcohol	1.00	0.21	-0.07	0.64	-0.33
ash	0.21	1.00	-0.07	0.22	-0.05
hue	-0.07	-0.07	1.00	0.24	-0.62
proline	0.64	0.22	0.24	1.00	-0.63
class	-0.33	-0.05	-0.62	-0.63	1.00

Classification Model