ND110 - Data Science I - Notebook

Contents

Course Info

Tags

- Author : AH Uyekita
- Dedication : 10 hours/week (suggested)
- Start : 14/12/2018
- End (Planned): 28/12/2018
- Title : Data Science I - Foundations Nanodegree Program
  - COD : ND110

Related Courses

- ND110 - Data Science I - Nanodegree Foundations

Objectives

I want to finish this course in two weeks. It includes the Optional videos and chapters.

Syllabus

- [x] Chapter 1 - Welcome
  - [x] Lesson 01 -
  - [x] Lesson 02 -
- [x] Chapter 2 - Python Introduction
  - [x] Lesson 01 -
  - [x] Lesson 02 -
  - [x] Lesson 03 -
  - [x] Lesson 04 -
  - [x] Lesson 05 -
- [x] Chapter 3 - Python for Data Analysis
  - [x] Lesson 01 -
  - [x] Lesson 02 -
  - [x] Lesson 03 -
  - [x] Lesson 04 -

Repository Structure

This is the structure of this repository, each course's chapters (or parts) will be stored in different folders.

```
ND110_data_science_foundation_01
|
+--  01-Chapter_01
|           |
|           +--  README.md                      # General information
|
+--  02-Chapter_02
|           |
|           +--  README.md                      # General information
|           +--  00-Project_01                  # Project 01
|           +--  01-Lesson_01                   # Files from Lesson 01
|           |        +--  README.md             # Notes from Lesson 01 from Chapter 02
|           +--  02-Lesson_02                   # Files from Lesson 02
|           |        +--  README.md             # Notes from Lesson 02 from Chapter 02
|           .
|
+--  03-Chapter_03
|           |
|           +--  README.md                      # General information
|           +--  00-Project_02                  # Project 02
|           +--  01-Lesson_01                   # Files from Lesson 01
|           |        +--  README.md             # Notes from Lesson 01 from Chapter 02
|           +--  02-Lesson_02                   # Files from Lesson 02
|           |        +--  README.md             # Notes from Lesson 02 from Chapter 02
|           .
```

Best practice

- Add all *deliverables* in the `GitKraken Glo`;
- Take notes using the `Markdown`.

Chapter 1

Welcome

This chapter is about the General aspects of the Udacity platform study.

Instructions

General information about the course.

- Projects Deadline
- Projects Review
- Mentoring

Tips

- Asking Help
- Keep in contact with the Slack Community
- Student Manual

Chapter 2

Python Introduction

2.1 Why Python Programming

2.1.1 Instructor

Python will be taught by Juno! As a data scientist, Juno built neural networks to analyze and categorize product images, a recommendation system to personalize shopping experiences for each user, and tools to generate insight into user behavior.

2.1.2 Welcome To Introduction To Python!

In this course, we use Python version 3 (or simply Python 3). If you'd like more details on previous versions of Python and how version 3 differs from previous versions, check out the History of Python on Wikipedia. If you're new to Python or programming in general, this article will make more sense after you've completed a lesson or two, so you may want to hold off for now. All you need to know now is that your solution code for the programming exercises in this course will be graded based on Python 3 code.

2.1.3 Programming In Python

As you learn Python throughout this course, there are a few things you should keep in mind.

- Python is case sensitive.
- Spacing is important.
- Use error messages to help you learn.

Let's get started!

2.2 Data Types and Operators

2.2.1 Conceitos Gerais

A maior parte do curso destina-se aos conceitos básicos.

- A indentação é importante para o Python porque ele define quando termina um *loop* e começa o outro ou quando termina um *statement* do *if*;
- Como muitas outras linguagens de programação o Python é *case sensitive*, isto é, maiúsculas e minúsculas definem variáveis diferentes;
- Os comentários dentro do código pode ser feito a partir do uso do `#`. ********** ### Boas práticas
- Utilizar os espaçamentos de maneira adequada para que as fórmulas fiquem fáceis de ler e consequentemente de entender;
- Comentar as linhas de código, pois o maior beneficiário desses comentários será você daqui 6 meses quando estiver revisando esse código;

2.2.2 Configuração

Instalei o Anaconda e estou rodando o Spyder 3.3.1. para simular/rodar o Python. O CodeSkulptor 3 é um alternativa.

2.2.2.1 Classes de Variáveis

Define-se as variáveis conforme as suas características e do seu uso, por exemplo, um `32` pode ser um número inteiro, até mesmo um número flutuante e um caracter. Tudo dependerá de como se define ele, usando-se as aspas ter-se-á um caracter, sem o "ponto final" é um número inteiro e com o ponto final o número de ponto flutuante (mais conhecido como *float*).

A tabela abaixo foi retirada do 39.*Summary* da *lesson* 02.

Data Structure	Ordered	Mutable	Constructor	Example
int	NA	NA	`int()`	5
float	NA	NA	`float()`	6.5
string	Yes	No	'' ou " " ou `str()`	"teste"
bool	NA	NA	NA	`True` ou `False`
list	Yes	Yes	[ ] ou `list()`	list[1,2]
tuple	Yes	No	( ) ou `tuple()`	tuple(1,2)
set	NA	Yes	{ } ou `set()`	set(1,2)
dictionary	NA	Keys: No	{ } ou `dict()`	dict("jul":1,"jun":2)

2.2.2.2 Inteiros e *float*'s

Como atribuir um `integer` e um `float` nas variáveis?

```
# Integer
teste_int = 100    # É um inteiro (não colocar o ponto final)
teste_int = 123    # É um inteiro (não colocar o ponto final)

# Float
teste_flo = 100.8  # É um float (só adicionar o ponto final)
teste_flo = 123.   # É um float (só adicionar o ponto final)
```

Note que haverá um conversão (ou *data coercion*) de `integer` para `float` caso tenha alguma operação envolvendo os dois tipos de variáveis.

```
# A nova variável será do tipo float
teste_new = teste_int + teste_flo
```

Observe que as variáveis podem ser convertidas para `integer`, `float` etc. Contudo, deve-se ressaltar que para os casos de conversão de `float` para `integer` pode-se ter uma perda de informação significativa.

```
teste = 100.987      # Variável float

teste = int(100.987) # Convertendo para integer

print(teste)         # Note que o valor impresso é 100
                     # perdeu-se tudo o que estava após a vírgula
```

2.2.2.3 Boolean `bool`

Além de `integer` e `float`, também há o `bool` que se refere aos booleanos (`True` e `False`). Observe que para o Python `True` e `False` (1 e 0) devem ser escritos exatamete como está neste texto, em `R` eles são escritos todos em caixa alta.

2.2.2.4 Strings `str`

O `strings` é uma cadeia de letras, um exemplo é o texto que estou escrevendo agora.

```
print("Hello World")  # É a impressão de uma simples string "Hello World", mas poderia
                      # ser bem mais complexa.

meu_texto = "Vamos fazer um teste!111"   # Note que esse exemplo é um pouco mais complexo
print(meu_texto)                         # já que tem espaços, números e caracteres especiais.
```

Surpreenda-se pois no Python pode-se usar os operadores (`+`, `*` etc.) para realizar algumas funções.

- `+` Une/concatena duas `strings`;
- `*` Multiplica a `string`.

2.2.2.5 Containers

A `list` é uma estrutura de dados que pode conter `integer`, `floats`, `strings` e `booleans`. Os benefícios das `lists` é que elas podem ser alteradas (*mutability*) e também podem ser reorganizadas.

```
minha_lista = [33 , "anderson" , True , 123.45] # Pode ser tudo misturado
minha_lista[0]                                  # Começa no zero - zero index based
minha_lista[-1]                                 # Tem os seus truques! Último elemento.
minha_lista[2:]               # Slicing         # Retorna os elementos cujo index é maior que 2
minha_lista[:2]               # Slicing         # Retorna os elementos cujo index é menor ou igual a 2
minha_lista[2:3]              # Slicing         # Retorna os elementos cujo index é maior que 2 e menor
```

Uma variable da classe `tuple` é um tipo de lista imutável, isto é, não tem como usar `.sorted()`. Eles possuem uso específico, por exemplo, em latitude e longitude, pois sempre estão juntos.

```
# Defining a tuple variable
my_name = "anderson","uyekita"

# Tuple unpacking
pri_nom,sob_nom = my_name

# Editing/Updating a tuple variable
my_name = "jose","silva"
```

Observe que não é necessário o uso dos parêntesis. Há a particularidade do "tuple unpacking" que é atribuir todos os valores do `tuple` de uma só vez em outras variáveis, conforme o exemplo.

Uma variável do tipo `set` possui algumas características marcantes como: é mutável, portanto, pode-se alterar os valores desse container de elementos, mas ela não é ordenada, ou seja, não se tem ideia de quem é o primeiro ou último elemento. Além dessas características, ela não admite valores duplicados.

```
# Definição de uma lista simples
minha_lista = [1,2,3,1,2,3,4,5,6] # há valores duplicados

meu_set = set(minha_lista)        # {4, 3, 2, 1, 5, 6}
                                  # Agora é um <class 'set'>
```

Como o `set` é *unordered* todas as vezes que imprimir essa variável dessa classe a sequencia dos elementos alterará. Ressalta-se que o `in` funciona nas variáveis da classe `set`. Además, há `methods` específicos para essa classe que será descrita adiante.

O *dictionary* é uma estrutura de dados peculiar que possui um *key* que pode ser usado para mapear sendo esse *key* tendo um *value*.

```
# Example dictionary
elements = {"anderson": 1, "hitoshi": 2, "uyekita": 3}

elements["key"] = "value"      # Add value mapped by key
elements["teste"] = 9          # Add a new variable in the dictionary
elements["uma_lista"] = [1,2]  # Add a list
elements["100"] = "a"          # Other example
```

Observe que o *dictionary* é bem eclético, tem de tudo e aceita tudo. Pode-se usar o operador `in` o qual terá um retorno `booleano`. O método `.get()` é usado para ter o retorno do *value* da um dado *key*.

O *nested dictionary* é o uso de um *dictionary* tendo os seus *values* um outro *dictionary*.

```
# Exemplo retirado do site da Udacity.
elements = {"hydrogen": {"number": 1,
                         "weight": 1.00794,
                         "symbol": "H"},
            "helium": {"number": 2,
                       "weight": 4.002602,
                       "symbol": "He"}}
```

Sendo assim possível organizar informações estruturadas.

2.2.3 Operadores

São os operadores matemáticos básicos:

2.2.3.1 + Adição

Adiciona dois elementos.

```
print(5 + 3) # Somando dois números inteiros

print("Hello" + " " + "World") # Somando três strings
```

2.2.3.2 - Subtração

Subtrai dois elementos.

```
print(5 - 3)
```

2.2.3.3 * Multiplicação

Multiplica dois elementos.

```
print(5 * 3)

print("Ha" * 5)  # Terá como saída HaHaHaHaHa
```

2.2.3.4 / Divisão

Divide dois elementos.

```
print(5 / 3) # Retorna um float
```

2.2.3.5 % Resto da Divisão

Divide dois elementos e retorna o resto da divisão.

```
print(5 % 3) # Retorna 2
```

2.2.3.6 ** Exponenciação

Eleva o primeiro termo ao segundo (normalmente anotamos como 5^3).

```
print(5 ** 3) # Retorna cinco elevado à terceira potência 5^3
```

2.2.3.7 // Retorna o Quociente da Divisão

Divide dois elementos e retorna o quociente da divisão.

```
print(5 // 3) # Retorna 1
```

2.2.3.8 `in` e `not it` Possui/Pertecen/Tem

O retorno desse operador é um valor booleano resultante da busca do primeiro elemento no segundo.

```
print("hitoshi" in ["anderson","hitoshi","uyekita"])       # Retorna True
print("hitoshi" not in ["anderson","hitoshi","uyekita"])   # Retorna False
```

2.2.4 *Assignment Operators*

Conforme abordado em sala de aula, essa forma de notação é para simplificar o código. É muito parecido com o `i++`(que é o mesmo que `i=i+1`) do C++.

2.2.4.1 `+=`

Atualização da variável a partir de soma.

```
teste = teste + 100   # Como eu faço
teste += 100          # Como um programador faz
```

2.2.4.2 `-=`

Atualização da variável a partir de uma subtração.

```
teste = teste - 100   # Como eu faço
teste -= 100          # Como um programador faz
```

2.2.4.3 `*=`

Atualização da variável a partir de uma multiplicação.

```
teste = teste * 100   # Como eu faço
teste *= 100          # Como um programador faz
```

2.2.4.4 `/=`

Atualização da variável a partir de uma divisão.

```
teste = teste / 100   # Como eu faço
teste /= 100          # Como um programador faz
```

2.2.4.5 `//=`

Atualização da variável pelo quociente de uma divisão.

```
teste = teste // 100   # Como eu faço
teste //= 100          # Como um programador faz
```

2.2.4.6 %=

Atualização da variável pelo resto de uma divisão.

```
teste = teste % 100   # Como eu faço
teste %= 100          # Como um programador faz
```

2.2.4.7 **=

Atualização da variável pelo resto de uma divisão.

```
teste = teste ** 100  # Como eu faço
teste %= 100          # Como um programador faz
```

Há outros mais complicados no site do Programiz.

2.2.5 *Comparison Operators*

Lógica básica de comparação entre dois argumentos, igual a qualquer outra linguagem de programação.

2.2.5.1 < Menor

Compara dois elementos e retorna `True` ou `False`.

```
100 < 90  # 100 é menor que 90
False     # Não
```

2.2.5.2 > Maior

Compara dois elementos e retorna `True` ou `False`.

```
100 > 90  # 100 é maior que 90
True      # Sim
```

2.2.5.3 <= Menor ou igual

Compara dois elementos e retorna `True` ou `False`.

```
100 <= 90  # 100 é menor ou igual a 90
False      # Não
```

2.2.5.4 >= Maior ou igual

Compara dois elementos e retorna `True` ou `False`.

```
100 > 90  # 100 é maior ou igual a 90
True      # Sim
```

2.2.5.5 `==` Igual

Compara dois elementos e retorna `True` ou `False`.

```
100 == 90  # 100 é igual (ou idêntico) a 90
False      # Não
```

2.2.5.6 `!=` Diferente (ou *não igual*)

Compara dois elementos e retorna `True` ou `False`.

```
100 != 90  # 100 é differente de 90
True       # Sim
```

2.2.6 *Logical Operators*

O Python possui 3 operadores de lógica que são: `and`, `or` e `not`. Eles devem ser escritos necessariamente em letras minúsculas.

2.2.6.1 `and`

Comporta-se conforme a tabela abaixo

a	b	**s**
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

2.2.6.2 `or`

Comporta-se conforme a tabela abaixo

a	b	**s**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

2.2.6.3 `not`

Retorna o booleano contrário.

a	**s**
0	**1**

a	**s**
1	**0**

2.2.7 Built-in functions

Esta será um lista das funções e qualquer outra coisa que foi apresentado em sala de aula.

2.2.8 `print()`

```
print("Hello World")  # Imprime Hello World
```

Imprime a variável.

2.2.9 `type()`

```
teste = type(100)    # Atribui o resultado da função em teste
print(teste)         # Imprime a classe da variável dentro do type()
                     # que no caso é 100. Logo, será <class 'int'>
```

Retorna o tipo de variável.

2.2.10 `len()`

```
print(len("Hello World"))  # Retorna 11, pois conta o espaço também.
```

Retorna o comprimento de uma `strings`, isto é, a quantidade de caracteres. E para os casos de um vetor retorna o comprimento.

2.2.11 `int()`

```
int(43.3)
```

Converte o elemento declarado na função para um `integer`.

2.2.12 `str()`

```
str(43.3)
```

Converte o elemento declarado na função para um `string`.

2.2.13 `float()`

```
str(43)
```

Converte o elemento declarado na função para um `float`.

2.2.14 max()

```
print(max([100,40,50,30,20]))                                          # Retorna 100
print(max(["anderson","hitoshi","uyekita","mogi","das","cruzes"]))  # Retorna uyekita
```

Retorna o maior número dentro de uma lista. Note que só funcionará quando a lista for só de um tipo de variável.

2.2.15 min()

```
print(min([100,40,50,30,20]))                                          # Retorna 20
print(min(["anderson","hitoshi","uyekita","mogi","das","cruzes"]))  # Retorna anderson
```

Retorna o maior número dentro de uma lista. Note que só funcionará quando a lista for só de um tipo de variável.

2.2.16 sorted()

```
print(sorted(["anderson","hitoshi","uyekita","mogi","das","cruzes"]))                  # Retorna uma li
print(sorted(["anderson","hitoshi","uyekita","mogi","das","cruzes"], reverse = True))  # Retorna uma li
```

Conforme a sua tradução, ordenará a lista de forma alfabética ou do menor para o maior. Ele pode reverter a forma de ordenar os retornos "setando" o argumento `reverse` para `True`.

2.2.17 Methods

Como o Python é uma linguagem orientada a objeto, há alguns `methods` relacionado a alguma `classe`. O funcionamento de um método é similar ao de uma função, contudo o método está ligado a alguma classe e só será útil para essa classe. Isto quer dizer que não há possibilidade de usar o `.title()` num número `integer` ou `float`.

2.2.18 .title()

```
meu_nome = "anderson uyekita"
print(meu_nome.title())
```

O retorno da aplicação do método `.title()` é a substituição das primeiras letras minúsculas do meu nome para maiúsculas, ficando assim " **A**nderson **U**yekita "

2.2.19 .islower()

```
meu_nome = "anderson uyekita"
print(meu_nome.lower())
```

O retorno da aplicação deste método retorna um `booleano` e significa se há ou não alguma letra maiúscula, se sim `False` senão `True`.

2.2.20 .format()

```
# Exemplo 1
print("Eu sou o Hitoshi e tenho {} anos".format(33))

# Exemplo 2 (retirado das notas de aula)
animal = "dog"
action = "bite"
print("Does your {} {}?".format(animal, action))
```

O retorno desse método é a substituição dos `{}` pelo `33`.

2.2.21 .split()

```
meu_nome = "anderson hitoshi uyekita"

print(meu_nome.split())  # Como resultado tem-se uma lista
                         # ['anderson', 'hitoshi', 'uyekita']
```

Conforme o nome diz, divide uma `string` baseado em algum separador que pode ser o espaço (*default*), tabulação, traços, pontos, etc.

2.2.22 .join()

```
print(" ".join(["anderson","hitoshi","uyekita","mogi","das","cruzes"])) # Retorna uma string com espaços
```

É o inverso do `.split()`. Atente-se que o "separador" é declarado antes do método `.join()`, neste exemplo é o `" "` é o espaço.

2.2.23 .append()

```
meu_nome = ["anderson","hitoshi"] # Minha lista
meu_nome.append("uyekita")        # Agregando na minha lista o meu sobrenome
```

Esse `method` altera a lista original `meu_nome`.

2.2.24 .add()

```
meu_nome = {"anderson","uyekita"}  # Para criar um set basta usar os {}

meu_nome.add("hitoshi")            # {"anderson","uyekita","hitoshi"}
```

A função adiciona um novo elemento (ou novos elementos) na variável set designada.

2.2.25 .pop()

```
meu_nome = {"anderson","uyekita","hitoshi"}  # Para criar um set basta usar os {}

meu_nome.pop("hitoshi")                      # Retira um elemento randomicamente
```

Isso é um pouco bizarro, mas o método `.pop()` remove um elemento aleatório do set designado.

2.2.26 `.get()`

```
meu_nome = {"anderson":1,"uyekita":2,"hitoshi":3}  # Para criar um set basta usar os {} e adicionar os v

print(meu_nome.get("hitoshi"))                     # Retorna 3
```

Isso é um pouco bizarro, mas o método `.pop()` remove um elemento aleatório do set designado. Deve-se ressaltar também que caso o `.get()` nenhum valor que procurar ele poderá retornar um valor *default*, sendo assim `.get("hitoshi",0)` ao invés de retornar "None", retornará 0.

2.2.27 Referências

- String methods

2.3 Control Flow

2.3.1 *Control Flow*

Nesta aula serão abordados temas como:

- *Conditional statements* (Famoso *if* e *else*)
- Expressões booleanas
- Loops usando *for* e *while*
- Como uma estratégia de parada dos loops *break* e *continue*
- Zip e enumerate (Isso eu num sei o que é)
- List comprehensions (idem, *no idea*)

2.3.1.1 Revisão da `Lesson02`

Data Structure	Ordered	Mutable	Constructor	Example
int	NA	NA	`int()`	5
float	NA	NA	`float()`	6.5
string	Yes	No	'' ou " " ou `str()`	"teste"
bool	NA	NA	NA	`True` ou `False`
list	Yes	Yes	[ ] ou `list()`	list[1,2]
tuple	Yes	No	( ) ou `tuple()`	tuple(1,2)
set	NA	Yes	{ } ou `set()`	set(1,2)
dictionary	NA	Keys: No	{ } ou `dict()`	dict("jul":1,"jun":2)

Maiores informações acerca das `built-in` *functions* e `methods` somente acessando as notes_lesson02.md.

2.3.2 Expressões Booleanas

Algumas expressões booleanas dignas de nota:

2.3.2.1 a < b < c ou a > b > c

Essa expressão pode ser uma condição de avaliação de uma *conditinal statement*, por exemplo.

```
# Exemplo do video
if 18.5 <= weight / height**2 < 25:       # A relação weight / height**2 deve estar
    print("BMI is considered 'normal'")   # entre 18.5 e 25 para que a condição seja aceita
```

2.3.3 *Conditional statements*

Note que no Python diferentemente de outras linguagem de programação não possui o maldito abre e fecha de `{}` para definir quando um *conditinal statement* começa e termina, por este motivo a indentação é fundamental para que a linguagem seja interpretada corretamente pelo compilador. **Não se esquecer do** `:`.

O exemplo abaixo possui dois *if*'s *nested*.

```
if condição:
+---+------------------+      # 1: Espaços necessários para indentar corretamente;
| 1 |        2         |      # 2: área do código que deverá ser executada caso
+---+------------------+      #    a condição seja verdadeira

else:
    if condição2:
    +---+----------------------+  # 3: Espaços necessários para indentar corretamente;
    | 3 |         4            |  # 4: área de código
    +---+----------------------+
    else:
    +---+----------------------+
    | 3 |         4            |
    +---+----------------------+
```

2.3.3.1 `if()`

```
if dinheiro < 200:      # Condição
    dinheiro += 200     # Ações caso a condição
    banco -= 200        # seja satisfeita
```

É o principal *conditional statement* e serve para filtrar, separar, eleger etc. algum elemento para sofrer uma determinada sequencia de comandos.

2.3.3.2 `else`

```
if (numero % 2) == 0:                # Se o resto for igual a zero, é um número par
    print(str(numero) + " é par")    # Imprime texto se a condição for verdadeira
else:
    print(str(numero) + " é ímpar")  # Imprime texto se a condição for falsa.
```

É o complemento do *if* e não requer condição (já que é o resto da condição do *if*), possui uma dualidade entre `0` e `1`, mas em alguns casos há mais de dois estados, nestas situações usa-se o `elif()`.

2.3.3.3 `elif()`

```
if numero > 100:
    print("maior que 100")

elif numero > 50:
    print("maior que 50 e menor e igual a 100")

else:
    print("número menor ou igual a 50")
```

Note que o o `elif()` é um intermediário entre o `if()` e o else, onde ele desempenha um papel de diminuir a quantidade de *nested if*.

2.3.4 *Loops*

As estruturas (que seriam os bloquinhos de código) do *f0r* e do *while* são parecidos com aqueles do *conditional statements*, pois os *loops* baseiam-se também pela indentação do código.

```
for condição:
+---+------------------+      # 1: Espaços necessários para indentar corretamente;
| 1 |        2         |      # 2: área do código que deverá ser executada caso
+---+------------------+      #    a condição seja verdadeira
```

2.3.4.1 `for()`

```
for i in n:      # i é a variável e n é uma lista de elementos
    print(i)     # imprime todos os elementos da lista n
```

O laço for será executado para cada elemento da lista *n*. Após percorrer a lista o laço termina.

2.3.4.2 `while()`

```
money = 0
while money < 1000:      # o laço será executo indefinidamente até que a condição seja False
    money += 100         # imprime money
```

A diferença entre *for* e *while* é que o segundo não possui limites, ele poderá ser executado indefinidamente até que a condição de parada seja atingida.

2.3.5 *List Comprehensions*

Isso é algo único da linguagem Python, segundo a **Juno Lee** só em Python tem essa coisa aí.

```
minha_lista = ["anderson", "hitoshi", "uyekita"] # Lista qualquer
nova_lista =[] # Uma nova lista

# Um loop de exemplo.
for nom in minha_lista:
    nova_lista.append(nom.title())

print(nova_lista) # imprime o resultado para comparar

# Em uma linha faz-se tudo.
nova_lista = [nom.title() for nom in minha_lista]

print(nova_lista) # para conferir.
```

Note que o mesmo código que foi feito utilizando algumas linhas, pode ser realizada usando uma única linha.

2.3.6 Built-in functions

Esta será um lista das funções e qualquer outra coisa que foi apresentado em sala de aula.

2.3.6.1 `range()`

```
start,end,step = 1,6,2 # inicializa os dados
                           # start possui valor default de zero
                           # step possui valor default de 1
lista = range(1,6,2)   # cria um objeto classe range
                           # print(lista) é um <class 'range'>
                            # A lista implícita é: [1, 3, 5]
```

A variável criada *lista* é um objeto de classe *range* e só será útil se caso a utilize em associação com o *for* (ainda não encontrei outras casos de uso do `range()`).

2.3.6.2 `sum()`

```
minha_lista = [10, 20, 30, 40] # Um exemplo de lista
sum(minha_lista)               # Retorna a soma dos elementos da lista
```

Simplesmente retorna a soma de todos os elementos de dada lista.

2.3.6.3 `Break`

Interrompe a instância de *for* ou *while* saindo totalmente do laço.

2.3.6.4 `Continue`

Pula a iteração para a próxima. É melhor porque não interrompe repentinamente a execução do *script*.

2.3.6.5 `zip()`

```
nome = ["anderson", "hitoshi", "uyekita"] # Uma lista qualquer
index = [1, 2, 3]                         # Outra lista qualquer
print(zip(nome,index))                    # Unindo as duas listas para criar uma lista de tuples
                                          # Note que se pode usar o unpacking também
```

Observe que é possível criar uma lista de *tuple*'s, conforme o exemplo acima, mas para realizar o *unpacking* deve-se utilizar um "operador" que ainda não foi abordado, mas de uso específico que é o "*".

```
lista_tuples = [('anderson', 1), # lista qualquer de tuples
                ('hitoshi', 2),  # é só um exemplo
                ('uyekita', 3)]

nome, index = zip(*lista_tuples) # Unpacking usando o *
```

Isso é bem prático quando tempos uma lista de *tuple*'s.

2.3.6.6 `enumerate`

```
letters = ['anderson', 'hitoshi', 'uyekita']  # Uma lista qualquer.

for i in enumerate(letters):                  # Cria uma lista de tuple's.
    print(i)                                  # Imprime cada tuple da lista criada.
```

A única maneira de entender essa função é aplicando a num *for*. O resultado é a criação de *tuple*'s.

2.3.7 Methods

2.3.7.1 `.items()`

```
apelidos = {"Chico": 'Francisco',    # Declarando uma biblioteca
            "Tião": 'Sebastião',
            "Zé": 'José'}

for key,value in apelidos.items():   # Emprego do .items()
    print("Apelido: {} e Nome:{}".format(key,value))
```

Nos casos onde é necessário as informações dos *values* de uma biblioteca, usa-se o `.items()`.

2.4 Functions

A função nada mais é que a abstração de uma rotina (ou *script*) encapsulado num simples *call* de uma função *built-in* (parece mas não é *built-in* porque nós que a criamos).

2.4.1 Conteúdo da aula

Essa lição abordou:

- Defining Functions
- Variable Scope
- Documentation
- Lambda Expressions
- Iterators and Generators

2.4.2 Estrutura de uma função

A função possui uma estrutura que deve ser seguida para que o devido funcionamento.

- Sempre começa com `def`;
- Deve ser indentada;
- Possui argumento;
- Pode ou não retornar (*return*) algum valor (*None* se não for retornar nada).

```
def my_function(arg1, arg2)
+---+----------------------------+
|   |                            |
| 1 |           2                |
|   |return                      |  # Note que não é uma exigência que tenha o return
+---+----------------------------+

1: área de indentação
2: área de código
```

Ressalta-se que as funções podem ter variáveis locais (*local variable*), isto é, as variáveis internas à função não afetam o *environment* externo dela (aqui eles chamam de *variable scope*). Contudo, as variáveis definidas no *global scope* podem ser usadas dentro da função e podem ser até modificadas caso essa variável for um dos argumentos da função.

Além disso, pode-se declarar variáveis *default* para cada argumento (igual o R). Outra similaridade com o R é com relação ao uso da função.

- Declarar argumentos por posição: my_function(10,5)
- Declarar argumentos por nomes: my_function(arg2 = 5, arg1 = 10)

2.4.3 *Lambda Expressions*

Não deixa de ser uma função, mas com um escopo menor, não possui nome (por isso que é uma função anônima) e provelmente será empregada para fins mais simples. Abaixo uma comparação entre uma função dita normal/convencional e a lambda.

```
# Função convencional
def multiply(x, y):
    return x * y

# Lambda Expression
multiply = lambda x, y: x * y
```

Note que é possível reduzir a quantidade de linhas, mas o principal é não ter que definir um novo nome para uma função que provavelmente não terá muito uso ao longo do corpo do código.

Conforme a **Juno Lee** as *lambdas expressions* tornarão importantes no futuro, quando o a quantidade de funções definidas é grande, isto é, fazendo-se pequenas alterações ou combinações entre as funções já definidas o poder da *lambda expression* pode ser muito grande. Só evoluindo para confirmar isso.

Um exemplo de como podemos associar funções regulares com o *lambda expressions*.

```
def teste(arg1):    # Eleva ao quadrado
    return arg1**2

def teste2(arg2):   # Divide por 100
    return arg2/100

teste3 = lambda x,y: teste(x) * teste2(y) # Lambda Expression que usa as duas funções

print(teste3(3,1)) # Espera-se que imprima o resultado de 9/100
```

2.4.4 *Documentation*

Asssim já como comentado na `Lesson02`, documentação nada mais que as boas práticas da programação.

- Explicar de maneira clara e concisa o que essa função faz;
- Quais são as entradas e que tipo de variáveis são (`int`, `float`, `list` etc.);
- Qual é a saída (`None`, `list`, `dict` etc.).

Conforme a convenção do PEP (eu acho que é isso) tem uma forma de anotar a documentação que é:

- Usar três " para abrir a documentação e outros três para fechar;
- Primeira linha: Discorrer sobre o uso geral da função;
- INPUT: quais são;
- OUTPUT: qual é.

Agora um exemplo de como comentar:

```
def my_function(arg1,arg2):
    """
    Função que não faz nada só serve como exemplo

    INPUT: arg1 é um int - É o tamanho da minha paciência em escrever essas anotações
    OUTPUT: arg2 é um flota - É a expectativa de dias melhores
    """
```

2.4.5 Built-in functions

Esta será um lista das funções e qualquer outra coisa que foi apresentado em sala de aula.

2.4.5.1 `map()`

```
my_df = [[10,20,30],[5,5,10],[1,50,1]] # Criei uma lista de lista

print(map(sum,my_df)) # Para cada lista irá aplicar a soma
                      # [60, 20, 52]
```

É como se estivesse aplicando as funções da família `apply`.

2.4.5.2 `filter()`

```
my_df = [10,20,30] # lista qualquer

teste = lambda x : x > 20 # lambda expression para avaliar

print(filter(teste,my_df)) # filter
                           # retornará os valores que forem avaliados como True da função lambda teste
```

Filtra os elementos com base numa função que avalia.

2.4.6 Referências para Iteratos e Generators

https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/

2.5 Scripting

Escrever e editar *scripts*, tendo até entradas de dadas externas (*inputs*) do usuário.

2.5.1 Conteúdo da aula

Essa lição abordou:

- Python Installation and Environment Setup
- Running and Editing Python Scripts
- Interacting with User Input
- Handling Exceptions
- Reading and Writing Files
- Importing Local, Standard, and Third-Party Modules
- Experimenting with an Interpreter

2.5.2 Erros e Exceções

Os *Syntax Erros* são quando o "interpretador do código" não consegue interpretar o código, isto pode ser ocasionado por uma má digitação (famoso *typo*). Seguramente, esses erros podem ser corrigidos antes de executar o código.

Ao passo que as *Exceptions* são mais complexos, pois o "interpretador do código" conseguiu ler, mas há alguma inconsistência. Note que esse erro ocorre durante a execução do programa. Há diferentes tipos de *exceptions*:

2.5.2.1 ValueError

```
x = int(input("Digite um número: "))
```

Caso a entrada seja um texto, por exemplo, "ten". Ocorrerá o problema de *ValueError*.

Tradução: Quando o objeto está correto, mas o valor declarado/passado como input é inapropriado para aquela função (*built-in* ou auela que criamos mesmo).

2.5.2.2 AssertionError:

Tradução: Quando uma condição assetiva falha...

2.5.2.3 NameError

```
# ValueError: Quando a variável a não foi declarada ainda
sum(a + 1)
```

2.5.2.4 IndexError:

```
teste = ["a","b"]

print(teste[100])
```

Ultrapassagem dos limites de uma lista.

2.5.2.5 TypeError

```
# TypeError: Tipo erro de quando soma _str_ com _int_
sum("a" + 1)
```

Tentar somar um *str* com um *int*. ********** ###
Importing scripts

Geralmente todo *script* importado deve ser declarado, por convenção, no início do arquivo.

```
import my_script
```

O local desse *script* tem que ser no mesmo diretório.

```
meu_recente_trabalho.py
|
+-my_script.py
```

2.5.3 `Built-in` *functions*

Esta será um lista das funções e qualquer outra coisa que foi apresentado em sala de aula.

2.5.3.1 `input()`

```
qualquer_coisa = input("Escreva-me algo interessante: ")
print(str("Isso é interessante hein!") + qualquer_coisa)
```

Pode-se obter informações a partir de entradas usando o *input*. Esse dado de entrada pode ser transformado em *int*, *float* etc..

2.5.3.2 `try()`

```
try:     # Opção a prova de tonterías
    int(input("Digite um número legal: "))  # Digita um número aí... 51
except:  # Caso o operador faça uma burrada o script não vai interromper bruscamente
    print("Tá de sacanagem né!? Eu pedi um número e não uma string!")  # Vai chamar a atenção dele.
```

O *try statement* juntamente com o `except()` possibilitará que o programador utilize diferentes estratégias para contar possíveis problemas de *inputs*, neste caso exemplo, aparecerá apenas um aviso e nada mais, há a possibilidade de retornar a solicitar um número, mas esse deve ser feito com um *loop* `while()`.

```
while True:  # Com esse loop o script fica teimoso, só sai quando o operador inserir um número.
    try:
        x = int(input("Digite um número legal: "))
        break # Depois de muita teimosia, por fim, digitou um número. Daí sai do While infinto.
    except:
        print("tá de sacanagem né!?")

print("Se você ler isso é que saiu do loop") # Só para enviar uma mensagem de aviso
```

Observe que se não fosse o `try()` este *script* teria sido interrompido.

2.5.3.3 `except()`

```
while True:
    try:
        x = int(input("Digite um número legal: "))
        break
    except ValueError:
        print("tá de sacanagem né!?")
      except KeyboardInterrupt:
        print("Apertou Ctrl+C")
        break
    finally:
        print("Digite um valor válido")
```

Note que o `except()` pode ter formas específicas de atuação:

- ValueError: Quando um valor não esperado é atribuído, neste caso quando se digita um texto ao invés de um número;
- KeyboardInterrupt: Quando se interrompe o *script* ao apertar o Ctrl+C.

2.5.3.4 `finally()`

```
while True:
    try:
        x = int(input("Digite um número legal: "))
        break
    except:
        print("tá de sacanagem né!?")
    finally:
        print("Digite um valor válido")
```

O `finally()` será executado sempre caindo no `try()` ou no `except()`. Não vi muita utilidade **agora**, mas em breve entenderei mais sobre o uso dele. A *Juno Lee* disse que há um uso muito interessante para os casos de usar o `try()` para carregar (*load*) arquivos, pois quando isso é feito (não sei o porquê) abre-se alguma coisa (ótimo hein) e o `finally()` serviria para fechar essa "abertura".

> Lendo o link do Stack, pode ser útil quando há um `return` no `except`, neste caso o `finally()` será executado **antes** do return.

2.5.3.5 `open()`

```
data = open("path/file.txt","r")  # Carregando um arquivo txt localizado pelo path.
                                  # O argumento "r" significa read only.
dataset = data.read()
data.close()
```

O resultado do `open()` será um *file object*, caso seja necessário alguma alteração neste arquivo que é apenas leitura, pode-se usar o *method* `.read()` para acessar o conteúdo desse arquivo.

Ressalta-se também que o argumento pode ser alterado para `w`, o que significa que o arquivo estará sendo aberto para ser gravado. Tenha cuidado porque o conteúdo desse arquivo será deletado para ser sobreescrito. O `method` usado para a gravação é o `.write()`.

2.5.4 Methods

2.5.4.1 .read()

```
data = open("path/file.txt","r")  # Carregando um arquivo txt localizado pelo path.
                                  # O argumento "r" significa read only.
dataset = data.read()
data.close()
```

Este *method* transforma o *file object* numa string, sendo assim possível a edição e análise.

2.5.4.2 .close()

```
data = open("path/file.txt","r")  # Carregando um arquivo txt localizado pelo path.
                                  # O argumento "r" significa read only.
dataset = data.read()
data.close()
```

Não me pergunte o por quê, mas se deve fechar o `open()` e por isso que tem essa coisa aqui. Caso isso fique aberto, haverá um uso desnecessário de memória para mantê-lo aberto.

2.5.4.3 .write()

Serve para gravar um arquivo tipo `txt`.

2.6 Project Overview (Instructions)

2.6.1 Overview

In this project, you will make use of Python to explore data related to bike share systems for Chicago. You will write code to import the data and answer interesting questions about it by computing descriptive statistics. You will also create some important functions and plot charts.

2.6.2 What Software Do I Need?

To complete this project, the following software requirements apply:

- Python 3. The following packages in the Python Standard Library will likely be useful: csv and matplotlib.
- A text editor, like Sublime or Atom.
- A terminal application (Terminal on Mac and Linux or Cygwin on Windows).

2.6.3 2. Project Details

2.6.3.1 Bike Share Data

Over the past decade, bicycle-sharing systems have been growing in number and popularity in cities across the world. Bicycle-sharing systems allow users to rent bicycles on a very short-term basis for a price. This allows people to borrow a bike from point A and return it at point B, though they can also return it to the same location if they'd like to just go for a ride. Regardless, each bike can serve several users per day.

Thanks to the rise in information technologies, it is easy for a user of the system to access a dock within the system to unlock or return bicycles. These technologies also provide a wealth of data that can be used to explore how these bike-sharing systems are used.

In this project, you will use data provided by Motivate, a bike share system provider for many major cities in the United States, to uncover bike share usage patterns. You will use the data from one of the largest cities of United States: Chicago.

2.6.3.2 The Datasets

Data for the first six months of 2017 are provided. The data file contain six (6) columns:

- Start Time (e.g. 2017-01-01 00:07:57)
- End Time (e.g. 2017-01-01 00:20:53)
- Trip Duration (in seconds, e.g., 776)
- Start Station (e.g. Broadway & Barry Ave)
- End Station (e.g. Sedgwick St & North Ave)
- User Type (Subscriber or Customer)
- Gender (Male or Female)
- Birth Year (e.g., 1980)

The original files, which can be accessed here (Chicago, New York City, Washington), had more columns and they differed in format in many cases. Some data wrangling has been performed to condense these files to the above core six columns to make your analysis and the evaluation of your Python skills more straightforward. In the Data Wrangling course that follows this course in the Data Analyst Nanodegree program, students learn how to wrangle the dirtiest, messiest datasets so don't fret if you worried about missing out.

2.6.3.3 The Questions

You will write code to complete the following tasks:

- Task 1: Print the first 20 samples(rows) from the database
- Task 2: Print the gender(column) of the first 20 samples
- Task 3: Create a function to get the columns as a list
- Task 4: Count how many of each gender do we have
- Task 5: Create a function to count the genders
- Task 6: Show the most popular gender
- Task 7: Plot a a chart using the previous data
- Task 8: Answer why summing the number of Males and Females doesn't match the number of samples
- Task 9: Find the minimum, maximum, mean and median duration of the trips
- Task 10: Get all the start stations of the dataset
- Task 11: Create a function count the occurrence of any given column (optional)

2.6.3.4 The Files

To answer these questions using Python, you will need to write a Python script. To help guide your work in this project, a template with helper code and comments is provided as a downloadable .py file. You will also need the dataset file. All of the following files are available for download.

chicago_bikeshare_pt.zip

- bikeshare.py
- chicago.csv
- Once you have downloaded this zip file, move to the next page for more details on the code you will be writing.

2.6.4 3. Code Walkthrough

2.6.4.1 *TODOs*

All of the code you must fill out in chicago_bikeshare_en.py is marked in comments that start with "TODO". Take a detailed read through that file to get a gauge for how the script flows and the additions you will have to make to complete this project.

2.6.4.2 *ASSERTs*

We are using the assert to make sure your code is returning an expected value or an output in the right format. DO NOT CHANGE IT. If you can't pass through an assert, ask for help.

2.6.4.3 *The `csv` Module*

The csv module is core to completing this project. One thing to be careful about—these bikeshare CSV files are quite large so iterating through them will be costly in terms of compute time.

- Be sure to bite off bits of code that you can chew and regularly test your code as you develop it. Print statements are your friend.
- Do not try to open the CSV with a text editor. It may crash your computer.
- Load each CSV file into a data structure once at the beginning of the script rather than at the beginning of every function. Hint: You may use the code as it was proposet (with a list of lists), but converting the DictReader iterator into a list of dictionaries (as described in this Stack Overflow post) could be handy! You only need to do the properly changes.

If you are familiar with NumPy and/or pandas, you may realize that using the csv module is much less efficient than these libraries tailored for data analysis. The csv module is used in this project so foundational programming skills can be tested, as well as to gain an appreciation for the speed at which NumPy and pandas (which are taught later) can do their calculations on large files. Do not use those libraries.

2.6.5 4. Project Submission `Due Jan 04/2019`

In this project, you will write Python code deal with the Chicago bike share data and answer interesting questions about it by computing descriptive statistics. You will also write practical functions to show your skills in Python. The initial code and dataset is available here.

2.6.6 Before You Submit

2.6.6.1 Check the Rubric

Your project will be evaluated by a Udacity reviewer according to this Project Rubric. Be sure to review it thoroughly before you submit. Your project "meets specifications" only if it meets specifications in all the criteria. If you see room for improvement in any category in which you do not meet specifications, be sure to take some time to revise your work until you feel it is up to expectations. In particular, there is one section of the rubric that cares about the quality of your code. It is important that you not only obtain the correct answers with your code, but that you have followed good coding practices to obtain your solutions.

2.6.6.2 Gather Submission Materials

All you need to submit for this project is:

- chicago_bikeshare_en.py: Your code

There is no need for you to include any data files with your submission.

2.6.7 Submitting the Project

When you're ready, click on the "Submit Project" button to go to the project submission page. You can submit your files as a .zip archive or you can link to a GitHub repository containing your project files. If you go with GitHub, note that your submission will be a snapshot of the linked repository at the time of submission. It is recommended that you keep each of your projects in a separate repository to avoid any potential confusion: if a reviewer gets multiple folders representing multiple projects, there might be confusion regarding what project is to be evaluated.

It can take us up to a week to grade the project, but in most cases it is much faster. You will get an email once your submission has been reviewed. If you are having any problems submitting your project or wish to check on the status of your submission, please email us at suporte@udacity.com. In the meantime, you should feel free to proceed with your learning journey by continuing on to the next module in the program.

Chapter 3

Python for Data Analysis

3.1 The Data Analysis Process

The Data Analysis Process is divided into 5 steps:

1. Question
2. Wrangle
3. Explore
4. Draw Conclusions
5. Communicate

This view it is the same presented by Roger Peng in Managing Data Analysis (a course of Executive Data Science Specialization), the next table shows a comparison between both:

Table 1 - Comparison between methods

Step	Juno Lee	Roger Peng**
1	Question	Stating and refining the question
2	Wrangle	Exploring the data
3	Explore	Building formal statistical models
4	Draw Conclusions	Interpreting the results
5	Communicate	Communicating the results

(**): Part of this table was extracted from the Coursera website.

The Figure 1 shows a summary of each step (this picture was picked from the book of Roger Peng called Art of Data Science).

Although the process has divided into 5 step, these step are not statics and usually this is not a linear from 1 to 5. There are a lot of iterations until the last step.

3.1.1 Step 1 - Ask a Question

Usually all the analysis starts based on a question (good one), which we would like to answer using data. Sometimes we already have these data and we need to "think" what is a good question to this data (probabily later you'll need more data). Generally, you do not have data but you have a question and you'll need to find a good data set. This question must have 5 features:

	Set Expectations	Collect Information	Revise Expectations
Question	Question is of interest to audience	Literature Search/Experts	Sharpen question
EDA	Data are appropriate for question	Make exploratory plots of data	Refine question or collect more data
Formal Modeling	Primary model answers question	Fit secondary models, sensitivity analysis	Revise formal model to include more predictors
Interpretation	Interpretation of analyses provides a specific & meaningful answer to the question	Interpret totality of analyses with focus on effect sizes & uncertainty	Revise EDA and/or models to provide specific & interpretable answer
Communication	Process & results of analysis are understood, complete & meaningful to audience	Seek feedback	Revise analyses or approach to presentation

Figure 3.1: Figure 1

- Interest
- Answerable
- Specific
- Pausible
- Not already answered

Keep in mind that the question must be specific. If the question is not specific, we must refine it.

3.1.2 Step 2 - Wrangle

This step is quite different from the Roger one, because here we deal with:

- Gathering: If you do not have data you need to find it;
  - Download from the database stored in the webs;
  - API
  - Web Scraping
- Assess: Assess the quality of the data and the structure.
  - Structural problems: Different files with same information, but distincts column names
  - Missing data
  - incorrect data type
  - duplicates
- Clean: Modifying the data to ensure the quality.

All this steps were to prepare the data to an analysis.

Sometimes Wrangling and EDA are binded into one step, but here are splited.

3.1.3 Step 3 - EDA

Here we perform the EDA (Exploratory Data Analysis), discover some patterns, relationships, descriptive analysis, maximize the potential of the analysis, visualizations, and models. Also, removing outliers, and creating new descriptive features from existind data.

- Exploring
- Augment

In this step usually we need to revisit the question and refine with the knowledge gathered (change the question or need more data).

3.1.4 Step 4 - Draw Conclusions

This is step was to predict something (machine learning or inferencial statistics).

3.1.5 Step 5 - Communicate

Communicate the results.

3.1.6 Packages

In this course, three packages will be used massively.

- Numpy

```
import numpy as np
```

Convention: Abbreviate numpy as np.

- Pandas

```
import pandas as pd
```

Convention: Pandas numpy as pd.

- matplotlib

```
import pandas as pd
% matplotlib inline
```

3.1.7 Methods

Methods used during the course videos.

3.1.7.1 read_csv()

```
df = pd.read_csv('student_scores.csv')
df = pd.read_csv('student_scores.csv', sep=':')
```

Import the dataframe.

3.1.7.2 .head()

```
df.head()
```

Show the first 5 rows

3.1.7.3 .shape

```
df.shape()
```

Prints the dimenions.

3.1.7.4 .dtypes

Prints the types of each variable.

3.1.7.5 .info()

Display a summary of each variable.

It is good to find missing values.

3.1.7.6 .nunique()

Return the number the unique values.

3.1.7.7 .describe()

This is a real summary.

3.1.7.8 .tail()

The last 5 rows.

3.1.7.9 loc and iloc

Selecting the columns using **names**.

```
df_means = df.loc[:,'id':'fractal_dimension_mean']
```

Subsetting columns from “id” to “fractal_dimension_mean”.

Same range of columns but using **index**.

```
# repeat the step above using index numbers
df_means = df.iloc[:,:11]
```

3.1.7.10 .duplicated()

Return a boolean vector., which could be useful to count the number of duplicated.

3.1.7.11 .drop_duplicated()

Show the data set cleaned without duplicated, but do not update the original dataframe to do it so you need to set inplace as True.

- inplace = True

3.1.7.12 .mean()

```
df["desired_column"].mean()
```

Mean function.

3.1.7.13 .fillna(X)

Fill the NA with X.

- Alternativaly you can add inplace to update the current dataframe.

3.1.7.14 pd.to_daytime(df['time'])

Update the object of time, but you need to assign to the dataframe columns to change it.

3.1.7.15 .hist()

```
data.hist()

data.hist(figsize = (8,8)) # Biger figures.

data['age'].hist() # For a specific variable/featues.
```

Plot a simple histogram, beware because if you have many feactures, the histogram going to be crowded.

3.1.7.16 .plot()

```
data['age'].plot(kind='hist'); # Different way to plot a hist()
data['age'].plot(kind='bar'); # Different way to plot a hist()
data['age'].plot(kind='pie',figsize= (8,8)); # Different way to plot a hist()

# Matrix
pd.plotting.scatter_matrix(data,figsize=(15,15))
```

```
# Scatter regular
data.plot(x = "compactness", y = "concavity" , kind = "scatter")

# Boxplot
data['concave_points'].plot(kind = "box")
```

3.1.7.17 value_counts()

Aggregates counts for each new unique value in a column. Shows a vector with this values.

```
data['something'].value_counts().plot(kind = 'bar'); # Creates a bar ploat based on the value_counts cr
```

3.1.8 Subsetting

```
data[data['anything'] == "M"] # data['anything'] == "M" -> will select each row with True.
```

3.1.9 Indexing

```
# Example 1
ind = data['something'].value_counts().index
data['something'].value_counts()[index].plot(kind='bar') # Indexing!!

# Example 2
ind = data['anything'].value_counts().index
df['anything'].value_counts()[index].plot(kind='pie',figsize = (8,8))
```

3.2 Data Analysis Process - Case Study I

3.3 Data Analysis Process - Case Study II

3.4 Programming Workflow for Data Analysis

3.5 Project Overview (Instructions)

3.5.1 Overview

In this project, you will analyze a dataset and then communicate your findings about it. You will use the Python libraries NumPy, pandas, and Matplotlib to make your analysis easier.

Preparation for this project with: Intro to Data Analysis

3.5.2 What do I need to install?

You will need an installation of Python, plus the following libraries:

```
* pandas
* NumPy
* Matplotlib
* csv
```

We recommend installing Anaconda, which comes with all of the necessary packages, as well as iPython notebook. You can find installation instructions here.

3.5.3 Why this Project?

In this project, you'll go through the data analysis process and see how everything fits together. Later Nanodegree projects will focus on individual pieces of the data analysis process.

You'll use the Python libraries NumPy, pandas, and Matplotlib, which make writing data analysis code in Python a lot easier! Not only that, these are sought-after skills by employers!

3.5.4 What will I learn?

After completing the project, you will:

- Know all the steps involved in a typical data analysis process
- Be comfortable posing questions that can be answered with a given dataset and then answering those questions
- Know how to investigate problems in a dataset and wrangle the data into a format you can use
- Have practice communicating the results of your analysis
- Be able to use vectorized operations in NumPy and pandas to speed up your data analysis code
- Be familiar with pandas' Series and DataFrame objects, which let you access your data more conveniently
- Know how to use Matplotlib to produce plots showing your findings

3.5.5 2. Project Details

3.5.5.1 How do I Complete this Project?

This project is connected with the Introduction to Data Analysis course, but depending on your background knowledge, you may not need to take the whole class to complete this project.

3.5.5.2 Introduction

For the final project, you will conduct your own data analysis and create a file to share that documents your findings. You should start by taking a look at your dataset and brainstorming what questions you could answer using it. Then you should use Pandas and NumPy to answer the questions you are most interested in, and create a report sharing the answers. You will not be required to use inferential statistics or machine learning to complete this project, but you should make it clear in your communications that your findings are tentative. This project is open-ended in that we are not looking for one right answer.

3.5.5.3 Step One - Choose Your Data Set

Click this link to open a document with links and information about data sets that you can investigate for this project. You must choose one of these datasets to complete the project.

3.5.5.4 Step Two - Get Organized

Eventually you'll want to submit your project (and share it with friends, family, and employers). Get organized before you begin. We recommend creating a single folder that will eventually contain:

- The report communicating your findings
- Any Python code you wrote as part of your analysis
- The data set you used (which you will not need to submit)

You may wish to use Jupyter notebook, in which case you can submit both the code you wrote and the report of your findings in the same document. Otherwise, you will need to submit your report and code separately. If you would like a notebook template to help organize your investigation, you can find a link in the resources at the bottom of the page or you can click here. You can also complete and submit the project in the classroom by going to the Project Notebook part of this lesson.

3.5.5.5 Step Three - Analyze Your Data

Brainstorm some questions you could answer using the data set you chose, then start answering those questions. You can find some questions in the data set options to help you get started.

Try and suggest questions that promote looking at relationships between multiple variables. You should aim to analyze at least one dependent variable and three independent variables in your investigation. Make sure you use NumPy and Pandas where they are appropriate!

3.5.5.6 Step Four - Share Your Findings

Once you have finished analyzing the data, create a report that shares the findings you found most interesting. If you use a Jupyter notebook, share your findings alongside the code you used to perform the analysis. make sure that your report text is contained in Markdown cells to clearly distinguish your comments and findings from your code work. You should also feel free to use other tools and software to craft your final report, but make sure that you can submit your report as an HTML or PDF file so that it can be opened easily.

3.5.5.7 Step Five - Review

Use the Project Rubric to review your project. If you are happy with your submission, then you're ready to submit your project. If you see room for improvement, keep working to improve your project!

3.5.6 3. Video

3.5.7 4. Investigate a Dataset

3.5.8 Project Submission

Choose one of Udacity's curated datasets and investigate it using NumPy and pandas. Go through the entire data analysis process, starting by posing a question and finishing by sharing your findings.

3.5.9 Evaluation

Use the Project Rubric to review your project. If you are happy with your submission, then you are ready to submit! If you see room for improvement in any category in which you do not meet specifications, keep working!

Your project will be evaluated by a Udacity reviewer according to the same Project Rubric. Your project must "meet specifications" or "exceed specifications" in each category in order for your submission to pass.

3.5.10 Submission

3.5.10.1 What to include in your submission

1. A PDF or HTML file containing your analysis. This file should include:

- A note specifying which dataset you analyzed
- A statement of the question(s) you posed
- A description of what you did to investigate those questions
- Documentation of any data wrangling you did
- Summary statistics and plots communicating your final results

2. Code you used to perform your analysis. If you used a Jupyter notebook, you can submit your .ipynb. Otherwise, you should submit the code separately in .py file(s).
3. A list of Web sites, books, forums, blog posts, github repositories, etc. that you referred to or used in creating your submission (add N/A if you did not use any such resources).

3.5.10.2 Jupyter notebook instructions

If you used a Jupyter notebook on your computer to create your project, you can include all your code and analysis in the notebook and do not need to create additional files for your analysis. You will still need to export your work in a PDF or HTML format also (see point 1 above), and include this in your submission as well. To download your notebook as an HTML file, click on File -> Download.As -> HTML (.html) within the notebook. If you get an error about "No module name", then open a terminal and try installing the missing module using pip install (don't include the "<" or ">" or any words following a period in the module name).

3.5.10.3 Ready to submit your project?

Click on the "Submit Project" button and follow the instructions to submit!

It can take us up to a week to grade the project, but in most cases it is much faster. You will get an email when your submission has been reviewed.

If you are having any problems submitting your project or wish to check on the status of your submission, please email us at review-support@udacity.com.