Contents

Week 1: Review of Python Syntax

By the end of this week, you will be able to recognize and apply the following Python constructs: objects, types, operators, if statements, functions, lists, dictionaries, Series and DataFrames. You have already been exposed to these concepts during the DataCamp courses during the summer, but we will review everything to ensure that you can comfortably use these building blocks to build more complex programs in the next few weeks.

Session 1: Basic Building Blocks

1.1 Objects, Types and Operators

Object: a “piece” of data.

```
[1]: # Examples
     3
     [3,4]
     "Learning Python is fun."
     True
```

```
True
```

Type: What “kind” of data.

```
[2]: type(3)
```

```
int
```

```
[3]: type([3,4])
```

```
list
```

```
[4]: type("Learning Python")
```

```
str
```

```
[5]: type(True)
```

```
bool
```

```
[6]: type(3.0)
```

```
float
```

Operator: +, -, *, /, <, >, <=, >=, ==, !=, and, or, etc.

```
[7]: 3+5
```

```
8
```

```
[8]: 3.0+5
```

```
8.0
```

```
[9]: [3]+[5]
```

```
[3, 5]
```

```
[10]: '3'+'2'
```

```
'32'
```

```
[11]: 'I Love Python'+'!'
```

```
'I Love Python!'
```

```
[12]: '3'+2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 '3'+2

TypeError: can only concatenate str (not "int") to str
```

```
[13]: 3==3.0
```

```
True
```

```
[14]: 3==[3]
```

```
False
```

```
[15]: 3!=[3]
```

```
True
```

```
[16]: 1<2 and 4<3
```

```
False
```

```
[17]: 1<2 or 4<3
```

```
True
```

Order of Operations

(Higher Rows have Precedence)

Operator	Examples
Parenthesized expressions	`( )`
Exponentiation	`**`
Positive/Negative	`+3 -2`
Multiplication/Division	`* /`
Addition/Subtraction	`+ -`
Bitwise operations	`& |`
Comparisons	`< <= > >= != ==`
Boolean NOT	`not`
Boolean AND	`and`
Boolean OR	`or`
Assignment	`=`

```
[18]: (3>2)*2**3+(4=='4')
```

```
8
```

```
[19]: 5.0*2>4 and 3<2
```

```
False
```

Discussion questions

a) Why do you think 5==5.0 yields True, but 5==‘5’ yields False? Any explanation that you find reasonable is fine.

b) Explain why 5+‘5’ yields an error message.

c) Without using a computer, predict the result of the following expressions. Then check your answers using the computer.

i) `5+3**2` **ii)** `5+(6>3)*2` **iii)** `4*(5-(2-1))+(6==1)` **iv)** `2*(2+1)+6/3+2**3` **v)** `'3.1'+'4'` **vi)** `2*4<2**3 or -3>2` **vii)** `not 3*1<2`

1.2 Assignment to Variables and Input/Output

The assignment `s=` should not be read as “s is equal to” but as “associate the name ‘s’ with the object.”

```
[20]: s='Hello world'
      s=s+'!'
      s
```

```
'Hello world!'
```

```
[21]: num=float(input('Enter a number: '))
      ans=num*2
      print(num,"* 2 =",ans)
```

```
Enter a number:  5

5.0 * 2 = 10.0
```

```
[22]: input()
```

```
 Hi

'Hi'
```

```
[23]: input('Enter a number: ')
```

```
Enter a number:  5

'5'
```

```
[24]: float('5')
```

```
5.0
```

```
[25]: print('You entered',5.0)
```

```
You entered 5.0
```

```
[26]: print('You entered',float(input('Enter a number: ')))
```

```
Enter a number:  10

You entered 10.0
```

```
[27]: round(3.1415926)
```

```
3
```

```
[28]: round(3.1415926,3)
```

```
3.142
```

```
[29]: help(round)
```

```
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negative.
```

Without using a computer, predict the final value of x after executing the following sequence.

```
x=3-1
x=x+1
x=2**x
```

```
[30]: # Example of multiple assignment
      num,ans=5,10
      # Example of f-string formatting
      print(f'{num} * 2 = {ans}')
```

```
5 * 2 = 10
```

```
[31]: first=input('Enter first name: ')
      last=input('Enter last name: ')
      print(f'Your full name is {first} {last}.')
```

```
Enter first name:  Peng
Enter last name:  Shi

Your full name is Peng Shi.
```

Grading scheme for the weekly exercises:

Download the Jupyter notebook containing all the exercises from Blackboard and submit it there before the due date. The grading scheme on all the weekly exercises are provided below.

- 5: Every question is completed and the code returns the correct outputs on all of the sample inputs.
- 4: Almost complete, but certain questions are blank, returns clearly incorrect outputs, or the code there does not run at all.
- 3: This score will not be assigned, as everyone should strive to get 4 or 5.
- 2: Not close to complete, but at least 50% complete.
- 1: At least 10% complete, but less than 50% complete
- 0: Less than 10% complete, or response is identical to someone else's, indicating plagiarism.

A perfect score is 5. Note that your code does not need to be absolutely perfect to receive a 5, but you need to complete every question and ensure that the outputs are correct on all of the sample runs included here. **To ensure that you get 5 out of 5, before you submit, restart the Kernel and run all, and check that all of the outputs are as intended.**

The weekly exercises are intended to be completed in 5-6 hours, excluding class time. You should budget at least this much time per week for these exercises.

Exercise 1.1: Automating a Calculation

a) Write a program that asks the user for the quantity sold and the price per unit and multiply them to calculate the total revenue, as in the sample run below. (Hint: you need to convert from string to float before multiplying.)

[33]:

```
Input quantity sold:  4
Input the price per unit:  3.5
```

```
Total revenue is $14.0.
```

b) Write a program that calculates the present value of a certain investment, which will pay off a cash value of C in n years. The program should ask the user to input the final cash value C, the annual interest rate r (in percentage points), and the number of years n. The formula for present value V is

$$V = \frac{C}{(1 + \frac{r}{100})^n}$$

You should round the final answer to two decimal places, as in the sample run below.

[34]:

```
Input final cash value:  300000
Input annual interest rate in percent:  5.5
Input number of years:  30
```

```
The present value is $60193.2.
```

1.3 If Statements

```
[35]: phrase=input('What school do you go to: ')
      if phrase=='USC':
          print('Fight on!')
```

```
What school do you go to:  USC
```

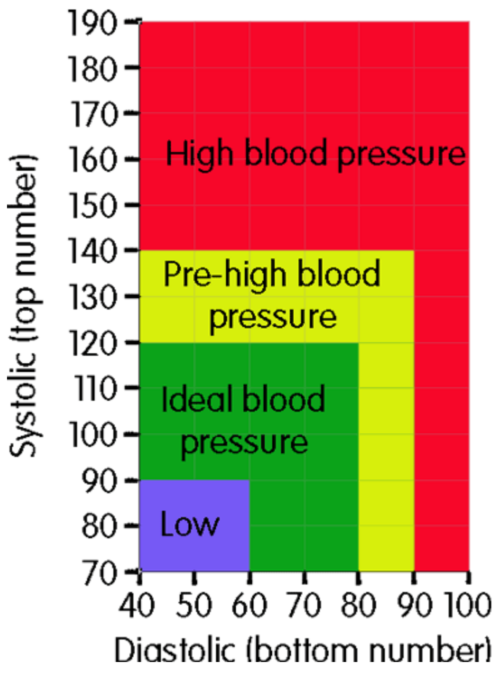
```
Fight on!
```

```
[36]: phrase=input('What school do you go to: ')
      if phrase=='USC':
          print('Fight on!')
```

```
What school do you go to:  UCLA
```

Example: Blood Pressure Triage

The following program asks for the user's systolic and diastolic blood pressure, and output one of "LOW", "IDEAL", "PRE-HIGH" or "HIGH" according to this chart:



```
[37]: high=float(input('Systolic blood pressure: '))
      low=float(input('Diastolic blood pressure: '))
      if low<=60 and high<=90:
          answer='LOW'
      elif low<=80 and high<=120:
          answer='IDEAL'
      elif low<=90 and high<=140:
          answer='PRE-HIGH'
      else:
          answer='HIGH'
      print(f'Your blood pressure is {answer}.')
```

```
Systolic blood pressure:  100
Diastolic blood pressure:  70

Your blood pressure is IDEAL.
```

Example: Quarantine Protocol

The following program implements a primitive protocol for triaging asymptomatic individuals during a pandemic.

```
[38]: Q1=input('Having been in one of the high risk regions in the past 14 days? (yes/no): ')
      if Q1=='yes':
          print('You should be quarantined for 14 days.')
      else:
          Q2=input('Have you been in contact with anyone who recently visited those regions? (yes/no): ')
          if Q2=='yes':
              print('You should consider self-quarantine for 14 days.')
          else:
              print("You don't need to quarantine but should practice the regular precautions.")
```

```
Having been in one of the high risk regions in the past 14 days? (yes/no):  no
Have you been in contact with anyone who recently visited those regions? (yes/no):  no

You don't need to quarantine but should practice the regular precautions.
```

Exercise 1.2: Automating Simple Logic

a) Basestock Policy in Inventory Management

Write a program that asks the user for the current inventory level.

- If the inventory is at least equal to the basestock level of 100, then output "Sufficient inventory. No need to order."
- Otherwise, output "Order x units", where x is the difference between the basestock level and the current inventory.

[39]:

```
Current inventory:  75
```

```
Order 25 units.
```

b) Payroll Calculator

Write a program that asks the user to input the number of hours worked this week, and calculates the total pay based on the following contract: for the first 40 hours, the hourly pay is 10. Each hour worked over 40 is compensated at an increased rate of 15 per hour.

[40]:

```
Hours worked this week:  42.5
```

```
Total pay is $437.5.
```

c): Grade Conversion

Write a program that asks the user to input a numerical score, and convert it into a letter grade based on the following table:

Score	Grade
At least 90	A
At least 80 but less than 90	B
At least 70 but less than 80	C
At least 60 but less than 70	D
Less than 60	F

[41]:

```
Enter numerical score:  95
```

```
Letter grade: A
```

Exercise 1.3: Temperature Converter

Write a program that asks for the user to input a temperature in Fahrenheit and convert it to Celcius, using the formula

$$T_{celcius} = (T_{fahrenheit} - 32) \times \frac{5}{9}.$$

The program should elicit the user to input a number using the prompt "Enter temperature in Fahrenheit:". Afterward, the program should print a line giving the temperature in celcius, rounded to two decimal places.

Sample run 1:

```
Enter temperature in Fahrenheit: 77
Equivalent temperature in celcius: 25.0
```

Sample run 2:

```
Enter temperature in Fahrenheit: 77.5
Equivalent temperature in celcius: 25.28
```

Exercise 1.4: BMI Calculator

Write a program which asks users to input their weight (in lbs) and height (in inches), and compute their body mass index (BMI), which is defind as $BMI = \frac{W}{H^2}$, where W is their weight converted to kg, and H is their height converted to m. (Assume 1 kg=2.205 lbs, and 1 m = 39.37 inches.) The program can assume that the weight and height the user inputs are positive numbers.

Classification	BMI (kg/m^2)
Underweight:	<18.5
Normal:	18.5 to 25 (>=18.5 but <25)
Overweight:	25 to 30 (>=25 but <30)
Obese:	>=30

The program should print a message classifying the BMI according to the above chart. The message must round the outputed BMI to 2 decimal places, and be formated exactly as the sample outputs below.

Sample run 1:

```
Please input your weight (in lbs): 145
Please input your height (in inches): 71
Your BMI is 20.22, which is classified as Normal.
```

Sample run 2:

```
Please input your weight (in lbs): 180.5
Please input your height (in inches): 60.5
Your BMI is 34.66, which is classified as Obese.
```

Session 2: Functions and Data Structures

1.4 Writing your own function

```
[44]: def add(a,b):
          return a+b
```

```
[45]: add(5,3)
```

```
8
```

```
[46]: add(2.5,9)
```

```
11.5
```

```
[47]: help(add)
```

```
Help on function add in module __main__:

add(a, b)
```

```
[48]: def add(a,b):
          'Function that returns the sum of the two input arguments. '
          return a+b
```

```
[49]: help(add)
```

```
Help on function add in module __main__:

add(a, b)
    Function that returns the sum of the two input arguments.
```

```
[50]: def add(a,b=1):
          '''Function with two input arguments:
        - a: the first argument.
        - b: the second (defaults to 1).
      The function returns the sum of the two arguments.
          '''
          return a+b
```

```
[51]: help(add)
```

```
Help on function add in module __main__:

add(a, b=1)
    Function with two input arguments:
     - a: the first argument.
     - b: the second (defaults to 1).
    The function returns the sum of the two arguments.
```

```
[52]: add(3)
```

```
4
```

```
[53]: # Returning multiple outputs
      def sumAndProduct(a,b):
          return a+b,a*b
```

```
[54]: s,p=sumAndProduct(5,3)
      print(f'sum: {s}  product: {p}')
```

```
sum: 8  product: 15
```

```
[55]: # Returning no outputs
      def sumAndProduct2(a,b):
          print(f'sum: {a+b}  product: {a*b}')
```

```
[56]: sumAndProduct2(5,3)
```

```
sum: 8  product: 15
```

```
[57]: def bloodPressure(high,low):
          '''Function that classifies a person's blood pressure.
          - high: systolic pressure.
          - low: diastolic pressure.
          '''
          if low<=60 and high<=90:
              answer='LOW'
          elif low<=80 and high<=120:
              answer='IDEAL'
          elif low<=90 and high<=140:
              answer='PRE-HIGH'
          else:
              answer='HIGH'
          return answer
```

```
[58]: print(f'100/70 is {bloodPressure(100,70)}.')
```

```
100/70 is IDEAL.
```

```
[59]: print(f'190/100 is {bloodPressure(190,100)}.')
```

```
190/100 is HIGH.
```

```
[60]: help(bloodPressure)
```

```
Help on function bloodPressure in module __main__:

bloodPressure(high, low)
    Function that classifies a person's blood pressure.
    - high: systolic pressure.
    - low: diastolic pressure.
```

Exercise 1.5: Making your Code for 1.2 Re-Usable

a) Write a function called `orderQuantity` with two input arguments

- **inventory** (default value 0): number of items on hand.
- **basestock** (default value 100): the target inventory level.

If inventory is at least equal to basestock, then return 0. Otherwise, return how many items you should order to meet the target. Include an appropriate docstring to explain what the function does.

```
[62]: # Code to test your function
      print('Result with no inputs:', orderQuantity())
      print(f'Order {orderQuantity(25)} when inventory is 25.')
      print(f'Order {orderQuantity(51,50)} when inventory is 51 and basestock is 50.')
```

```
Result with no inputs: 100
Order 75 when inventory is 25.
Order 0 when inventory is 51 and basestock is 50.
```

```
[63]: orderQuantity(basestock=200)
```

```
200
```

```
[64]: orderQuantity(inventory=80)
```

```
20
```

```
[65]: help(orderQuantity)
```

```
Help on function orderQuantity in module __main__:

orderQuantity(inventory=0, basestock=100)
    Calculates order quantity given current inventory and basestock level
    - inventory: number of items on hand
    - basetock: the target inventory level
```

```
[66]: # Tests using assert
      # Test case 1: inventory is greater than or equal to basestock
      assert orderQuantity(100, 100) == 0
      assert orderQuantity(200, 100) == 0

      # Test case 2: inventory is less than basestock
      assert orderQuantity(60, 100) == 40
      assert orderQuantity(0, 100) == 100
```

b) Write a function called `calculateWage` with three input arguments:

- **hours**: the number of hours worked.
- **base** (default value 10): the pay for each of the first 40 hours worked.
- **bonus** (default value 0.5): the proportional bonus for each hour worked above 40.

The function should return the total pay.

```
[68]: # Testing code
      print('Pay for 42 hours with default base and bonus:',calculateWage(42))
      print('Pay for 42 hours with base 12/hour and default bonus:',calculateWage(42,12))
      print('Pay for 42 hours with base 12/hour and bonus 60%:', calculateWage(42,12,.6))
      print('Pay for 42 hours with default base and bonus 60%:',calculateWage(42,bonus=0.6))
```

```
Pay for 42 hours with default base and bonus: 430.0
Pay for 42 hours with base 12/hour and default bonus: 516.0
Pay for 42 hours with base 12/hour and bonus 60%: 518.4
Pay for 42 hours with default base and bonus 60%: 432.0
```

c) Write a function called `letterGrade` with one input argument:

- **score**: the numerical score.

The function should return a string denoting the letter grade.

```
[70]: # Code to test
      print(f'Grade corresponding to 95 is {letterGrade(95)}.')
      print(f'Grade corresponding to 80 is {letterGrade(80)}.')
      print(f'Grade corresponding to 60 is {letterGrade(60)}.')
```

```
Grade corresponding to 95 is A.
Grade corresponding to 80 is B.
Grade corresponding to 60 is D.
```

```
[71]: # More extensive tests
      assert letterGrade(90)=='A'
      assert letterGrade(89)=='B'
      assert letterGrade(80)=='B'
      assert letterGrade(79)=='C'
      assert letterGrade(70)=='C'
      assert letterGrade(69)=='D'
      assert letterGrade(60)=='D'
      assert letterGrade(59)=='F'
      assert letterGrade(0)=='F'
```

1.5 List

A list is the simpliest data structure and it allows you to store a sequence of arbitrary objects while keeping track of their order.

```
[1]: # product sales in successive weeks
     sales=[10,25,5,15]
```

```
[2]: sales[0]
```

```
10
```

```
[3]: sales[1]
```

```
25
```

```
[4]: len(sales)
```

```
4
```

```
[5]: sales.append(20)
     sales
```

```
[10, 25, 5, 15, 20]
```

```
[6]: sales+[5,10]
```

```
[10, 25, 5, 15, 20, 5, 10]
```

```
[7]: sales
```

```
[10, 25, 5, 15, 20]
```

```
[8]: max(sales)
```

```
25
```

```
[9]: min(sales)
```

```
5
```

```
[10]: sum(sales)
```

```
75
```

```
[11]: sales[1:3]
```

```
[25, 5]
```

```
[12]: sales[:2]
```

```
[10, 25]
```

```
[13]: sales[-1]
```

```
20
```

```
[14]: 25 in sales
```

```
True
```

```
[15]: 25 not in sales
```

```
False
```

```
[16]: data=[[10,7.5],[25,5.5],[5,10],[15,7]]
```

```
[17]: data[0]
```

```
[10, 7.5]
```

```
[18]: data[0][1]
```

```
7.5
```

Exercise 1.6: Practice with Lists

a) Create an empty list called "l". **b)** Append each of the following objects to the list: `30`, `4.0`, `[3,4]`, `"Hello"`. The final value of the list should be as below.

```
[21]: # Value of l after running your code for parts a) and b) in order
      l
```

```
[30, 4.0, [3, 4], 'Hello']
```

c) Obtain the type of each element of the list.

d) Obtain a slice of the list from `30` to `4.0`.

e) Obtain the third element `[3,4]` in two different ways, using positive and negative indexing respectively.

f) Find the length of the third element (which is a list as well).

g) Check if the integer `4` is in the list. Do the same for the string `'4'`.

1.6 Dictionary

A dictionary represents a mapping between keys and values. It can be used to store labelled data in which the relative order of entries does not matter.

```
[28]: sales={'USA':3000, 'China':2000, 'India':4000}
```

```
[14]: sales['China']
```

```
2000
```

```
[15]: len(sales)
```

```
3
```

```
[16]: # Adding a new entry
      sales['UK']=1000
      sales
```

```
{'USA': 3000, 'China': 2000, 'India': 4000, 'UK': 1000}
```

```
[17]: # Updating entries
      sales['China']=2500
      sales
```

```
{'USA': 3000, 'China': 2500, 'India': 4000, 'UK': 1000}
```

```
[18]: # Checking whether a key exists
      'USA' in sales
```

```
True
```

```
[19]: 'USA' not in sales
```

```
False
```

```
[29]: max(sales)
```

```
'USA'
```

```
[33]: sales.get('USA')
```

```
3000
```

```
[32]: max(sales,key=sales.get)
```

```
'India'
```

```
[20]: sales.keys()
```

```
dict_keys(['USA', 'China', 'India', 'UK'])
```

```
[21]: sales.values()
```

```
dict_values([3000, 2500, 4000, 1000])
```

```
[22]: # Checking whether a value exists
      4000 in sales.values()
```

```
True
```

Exercise 1.7: Practice with Dictionary

a) Create an empty dictionary called "english".

b) Add the keys '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' to this dictionary, and each number should maps to its name in English in lowercase. (i.e. the key '3' should map the the value 'three'). You should do this one key-value pair at a time.

c) Ask the user to input a digit. If the input is one of the keys of the above dictionary, then print "You entered XXX." where XXX is the corresponding value. For any other input, print "I cannot read your input."

```
[26]:
```

```
Please enter a digit: 3
You entered three.
```

1.7 Series

To motivate the need for Series, consider some drawbacks of lists and dictionaries when representing the following table:

Day	Temperature
Sun	55
Mon	61
Tue	55
Wed	65

- Lists are suitable for storing sequential data, and are accessed by numerical indices.

```
[19]: day = ['Sun', 'Mon', 'Tue', 'Wed']
      temp = [55, 61, 55, 65]
      print(day[0],'->', temp[0])
```

```
Sun -> 55
```

- Dictionaries are suitable for storing a mapping between key and values. The benefit is that the keys carry more meaning. The drawback is that dictionaries do not maintain the sequential order of information, so one cannot refer to an element by its order.

```
[20]: weather = {'Sun': 55, 'Mon': 61, 'Tue': 55, 'Wed': 65}
      weather['Sun']
```

```
55
```

Series is a data structure in the `pandas` package that combines the funtionality of both lists and dictionaries: it allows both user-defined keys and maintain the sequential order. Think of it as a list in which the entries are also labelled. It is the backbone of modern data analysis in Python.

Creating a Series

One can create a Series either from two lists or from one dictionary, as below.

```
[2]: import pandas as pd
     s = pd.Series([55, 61, 55, 65], index=['Sun', 'Mon', 'Tue', 'Wed'])
     s
```

```
Sun    55
Mon    61
Tue    55
Wed    65
dtype: int64
```

```
[16]: s = pd.Series({'Sun': 55, 'Mon': 61, 'Tue': 55, 'Wed': 65})
      s
```

```
Sun    55
Mon    61
Tue    55
Wed    65
dtype: int64
```

One can also create an empty Series and add entries as needed, as below.

```
[17]: s= pd.Series(dtype=int)
      s['Sun']=55
      s['Mon']=61
      s['Tue']=55
```

```
s['Wed']=65
s
```

```
Sun    55
Mon    61
Tue    55
Wed    65
dtype: int64
```

Examining a Series

After loading a Series, you may want to first examine the first few entries or the last few entries instead of overwhelming yourself with all of the data.

```
[26]: s.head(3)
```

```
Sun    55
Mon    61
Tue    55
dtype: int64
```

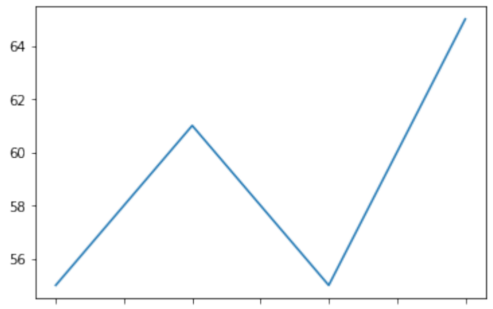
```
[27]: s.tail(2)
```

```
Tue    55
Wed    65
dtype: int64
```

Basic plotting is easy with Series.

```
[28]: s.plot()
```

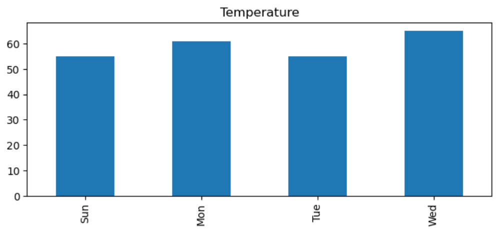
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f846c8ca2d0>
```



Note that due to a bug in Jupyter notebook, the very first figure in a notebook may not render and may need to be re-run. To get rid of the <matplotlib.axes...>, you can do

```
[34]: import matplotlib.pyplot as plt
      s.plot()
      plt.show()
```

```
[22]: s.plot(kind='bar',title='Temperature',figsize=(6,3))
      plt.show()
```



Other options for the `kind=XXX` input argument to the function `Series.plot` are as follows:

kind	description
'line'	line plot (default)
'bar'	vertical bar plot
'barh'	horizontal bar plot
'hist'	histogram
'box'	boxplot
'density'	Smoothed line plot
'area'	area plot
'pie'	pie plot

See `help(pd.Series.plot)` for more information.

You can sort a Series by its values using `.sort_values`.

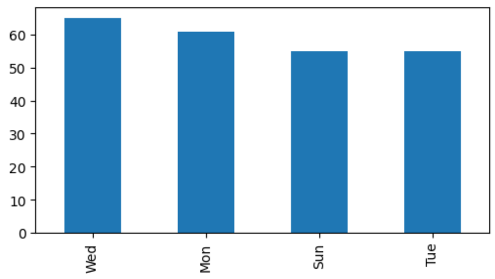
```
[31]: s.sort_values()
```

```
Sun    55
Tue    55
Mon    61
Wed    65
dtype: int64
```

```
[32]: s.sort_values(ascending=False)
```

```
Wed    65
Mon    61
Tue    55
Sun    55
dtype: int64
```

```
[25]: s.sort_values(ascending=False).plot(kind='bar',figsize=(6,3))
      plt.show()
```



Descriptive Statistics

```
[34]: s.describe()
```

```
count     4.000000
mean     59.000000
std       4.898979
min      55.000000
25%      55.000000
50%      58.000000
75%      62.000000
```

```
max      65.000000
dtype: float64
```

```
[35]: s.sum()
```

```
236
```

```
[36]: s.mean()
```

```
59.0
```

```
[37]: s.quantile(0.25)
```

```
55.0
```

```
[38]: # Number of values above 60
      (s>60).sum()
```

```
2
```

```
[39]: # Index of maximum value
      s.idxmax()
```

```
'Wed'
```

```
[40]: # Index of minimum value
      s.idxmin()
```

```
'Sun'
```

Filtering a Series

```
[41]: s
```

```
Sun    55
Mon    61
Tue    55
Wed    65
dtype: int64
```

a) Selecting a single value

You can obtain a specific value by indexing either via its position via `.iloc[]` or its index via `.loc[]`.

```
[42]: s.iloc[0]
```

```
55
```

```
[43]: s.loc['Sun']
```

```
55
```

b) Selecting a consecutive block

As with lists, you can obtain a subsequence of consecutive values using a colon when indexing with `.iloc` or `.loc`

```
[44]: # exclusive of the upper endpoint.
      s.iloc[0:3]
```

```
Sun    55
Mon    61
Tue    55
dtype: int64
```

```
[45]: # inclusive of the upper endpoint.
      s.loc['Sun':'Tue']
```

```
Sun    55
Mon    61
Tue    55
dtype: int64
```

c) Boolean indexing

Another way of filtering is to supply a True/False list of the same length, indicating which elements to include.

```
[46]: s[[False,True,False,True]]
```

```
Mon    61
Wed    65
dtype: int64
```

This is typically done to filter according to a condition, as below.

```
[47]: s>60
```

```
Sun    False
Mon     True
Tue    False
Wed     True
dtype: bool
```

```
[48]: s[s>60]
```

```
Mon    61
Wed    65
dtype: int64
```

More complex logic can be done using the vectorized operators `&` and `|`:

- `&`: the vectorized AND of two Series. (Note that we use "and" for single values, but "&" for Series.)
- `|`: the vectorized OR of two Series.

Note: Parenthesis must be used before and after to ensure the right order of operations.

```
[49]: (s<56) | (s>62)
```

```
Sun     True
Mon    False
Tue     True
Wed     True
dtype: bool
```

```
[50]: s[(s<56) | (s>62)]
```

```
Sun    55
Tue    55
Wed    65
dtype: int64
```

Exercise 1.8: Practice with Series

a) Create a Series named `tuition` with the labels being the year and the values being the cost, as given by the two lists below.

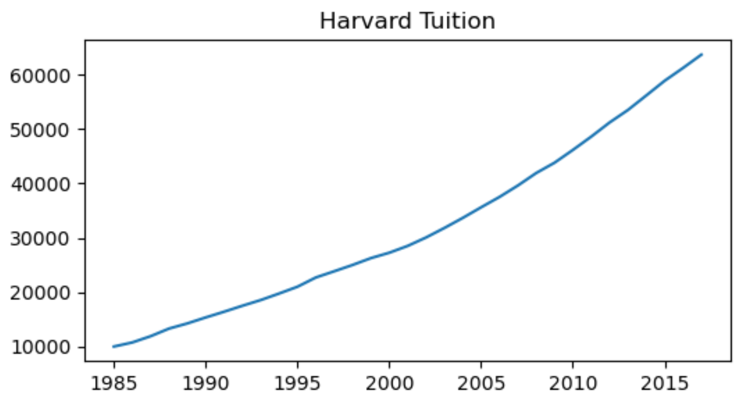
```
[1]: years = [1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000,␣
↪2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017]
     costs = [10000, 10750, 11900, 13300, 14250, 15350, 16400, 17500, 18550, 19750, 21000, 22700, 23840,␣
↪25000, 26260, 27250, 28500, 30050, 31800, 33650, 35600, 37500, 39600, 41900, 43800, 46150, 48600, 51200,␣
↪53500, 56175, 58875, 61225, 63675]
```

```
[2]: import pandas as pd
     tuition=pd.Series(costs,index=years)
```

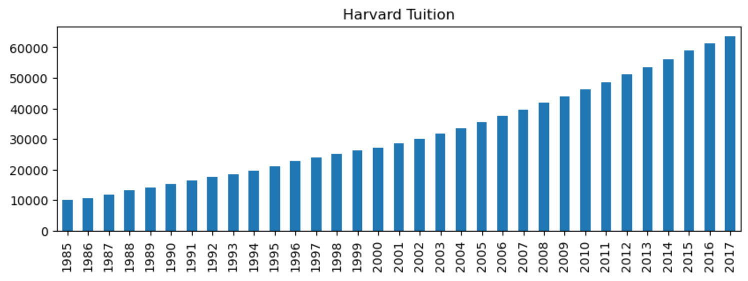
b) Display the first five entries as well as the last seven entries.

c) Make a line plot of the tuition, as well as a vertical bar plot.

[27]:



[26]:



d) Create another Series called `normalized_tuition` which divides Series `tuition` by the tuition in the year 2000.

e) What is the average tuition overall? What is the median tuition? Which year has the maximum tuition?

f) What is the average tuition in the last 10 years of the data? What is the average tuition in the first 10 years? What is the average tuition from the Year 2001 to 2010 inclusive?

g) How many years in the dataset had tuition over between 30000 and 40000 (inclusive)?

h) Filter the Series and include only the years in which tuition is between 30000 and 40000.

1.8 DataFrame

A DataFrame is a data structure in the `pandas` package that represents a table. Whereas Series is used for 1-dimensional data, DataFrame is used for 2-dimensional. Each row or column of a DataFrame is a Series, so manipulating a Dataframe is straightforward if you know how to manipulate a Series.

Creating a DataFrame

The most straightforward way to create a DataFrame is from a list of lists. Each inner list corresponds to a row. The row and column labels can be added using `index=` and `columns=`, as follows.

```
[67]: data=[['Jansen','Yih',4,21],['Alice','Mackinzie',5,25],['Jorge','Lopez',4,23]]
      import pandas as pd
      grades=pd.DataFrame(data,index=['A','B','C']\
                          ,columns=['First','Last','Problem Set 1','Exam'])
      grades
```

```
    First       Last  Problem Set 1  Exam
A  Jansen        Yih              4    21
B   Alice  Mackinzie              5    25
C   Jorge      Lopez              4    23
```

An alternative way is to start with an empty DataFrame and add the entries in later. Note that `.loc` is used since we are referring to row/column labels rather than positions.

```
[68]: grades=pd.DataFrame()
      grades.loc['A','First']='Jansen'
      grades.loc['A','Last']='Yih'
      grades.loc['A','Problem Set 1']=4
      grades.loc['A','Exam']=21
      grades
```

```
    First Last  Problem Set 1  Exam
A  Jansen  Yih            4.0  21.0
```

```
[69]: grades.loc['B','First']='Alice'
      grades.loc['B','Last']='Mackinzie'
      grades.loc['B','Problem Set 1']=5
      grades.loc['B','Exam']=25
      grades
```

```
    First       Last  Problem Set 1  Exam
A  Jansen        Yih            4.0  21.0
B   Alice  Mackinzie            5.0  25.0
```

A third way to create a DataFrame is to give a dictionary of lists. Each key of the dictionary corresponds to a column label.

```
[19]: dic={'First':['Jansen','Alice','Jorge'],\
          'Last':['Yih', 'Mackinzie','Lopez'],\
          'Problem Set 1':[4,5,4],\
          'Exam':[21,25,23]}
      grades=pd.DataFrame(dic,index=['A','B','C'])
      grades
```

```
    First       Last  Problem Set 1  Exam
A  Jansen        Yih              4    21
B   Alice  Mackinzie              5    25
C   Jorge      Lopez              4    23
```

Examining a DataFrame

The following DataFrame methods are analogous to the analogs for Series.

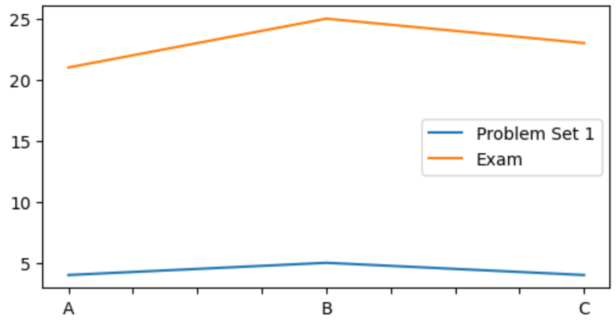
```
[20]: grades.head(2)
```

```
    First       Last  Problem Set 1  Exam
A  Jansen        Yih              4    21
B   Alice  Mackinzie              5    25
```

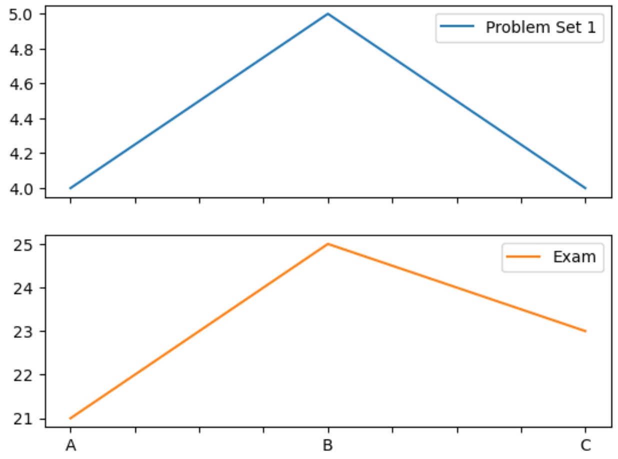
```
[21]: grades.tail()      # displays last 5 rows
```

```
    First       Last  Problem Set 1  Exam
A  Jansen        Yih              4    21
B   Alice  Mackinzie              5    25
C   Jorge      Lopez              4    23
```

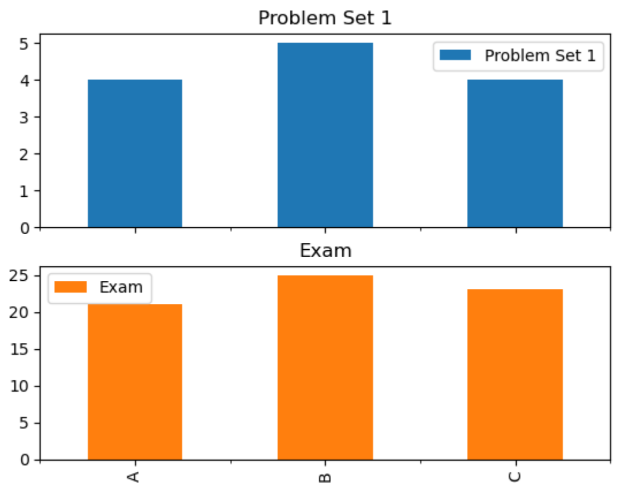
```
[28]: # Plot every numerical column
      grades.plot(figsize=(6,3))
      import matplotlib.pyplot as plt
      plt.show()
```



```
[23]: grades.plot(subplots=True)
      plt.show()
```



```
[24]: grades.plot(subplots=True,kind='bar')
      plt.show()
```



```
[76]: grades.sort_values(by='Exam',ascending=False)
```

```
    First      Last  Problem Set 1  Exam
B   Alice  Mackinzie              5    25
C   Jorge     Lopez               4    23
A  Jansen       Yih               4    21
```

Filtering a DataFrame

a) Selecting a Single Column

Recall that a column of a DataFrame is a Series

```
[77]: grades['Exam']
```

```
A    21
B    25
C    23
Name: Exam, dtype: int64
```

b) Selecting Particular Rows and Columns

- `.loc[rows,columns]`: indexing by row and column labels.
- `.iloc[rows,columns]`: indexing by row and column positions.

```
[78]: grades
```

```
    First      Last  Problem Set 1  Exam
A  Jansen       Yih               4    21
B   Alice  Mackinzie              5    25
C   Jorge     Lopez               4    23
```

Predict what each of the following lines select (before running to check)

```
grades.loc[['B','C'],['Last','Exam']]
grades.iloc[[2,1],:]
grades.loc['B','Exam']
grades.iloc[2,2]
grades.iloc[:,1]
grades.loc['B',:]
grades.loc['B':'C','First':'Problem Set 1']
grades.iloc[:,:3]
grades.loc[:,'Problem Set 1'].iloc[:2]
```

c) Boolean Indexing

As with Series, it is possible to supply a list of True/False entries within square brackets `[]` to indicate which rows to select. The length of the list must equal to the total number of rows.

```
[88]: grades[[True,False,True]]
```

```
    First   Last  Problem Set 1  Exam
A  Jansen    Yih              4    21
C   Jorge  Lopez              4    23
```

This can be used to select rows that meet a certain condition, as below

```
[89]: grades['Exam']>=23
```

```
A    False
B     True
C     True
Name: Exam, dtype: bool
```

```
[90]: grades[grades['Exam']>=23]
```

```
   First       Last  Problem Set 1  Exam
B  Alice  Mackinzie              5    25
C  Jorge      Lopez              4    23
```

```
[91]: grades[(grades['Exam']>=23) & (grades['Problem Set 1']==5)]
```

```
   First       Last  Problem Set 1  Exam
B  Alice  Mackinzie              5    25
```

Exercise 1.9: Practice with DataFrames

This question asks you to perform simple operations based on the following DataFrame.

```
[37]: import pandas as pd
      insurance=pd.DataFrame([[1000,20,0.8],[2000,25,0.9],[1500,30,0.85]],\
                             index=['A','B','C'],\
                             columns=['Deductible','Copay','Coverage'])
      insurance
```

```
   Deductible  Copay  Coverage
A        1000     20      0.80
B        2000     25      0.90
C        1500     30      0.85
```

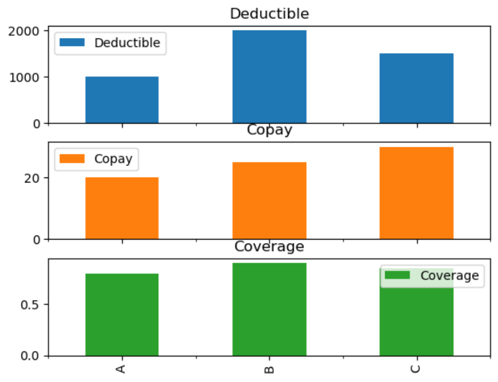
a) Obtain the coverage of plan B in two ways, one using `.loc` and one using `.iloc`.

b) Obtain a DataFrame that sorts the plans by increasing order of deductible, as follows.

c) Obtain a Series which represents the copay column.

d) Obtain a Series which represents the row for plan B.

e) Generate bar plots comparing the three plans, as below.



Exercise 1.10: BMI Calculator v2

This question asks you to package the code from Exercise 1.4 into a function. In general, whenever you are asked to code a function, you should follow the steps in this exercise to first develop a version without the function as in Exercise 1.4, and package the code into a function only once everything is correct. To package a segment of code into a function, copy and paste the code segment into a new cell, highlight everything and press Tab to indent everything, then write the function declaration at the beginning, and make minor edits to make sure that the code segment is using the inputs from the function.

Write a function called `calculateBMI` with two input arguments:

- `weight`: the person's weight in lbs.
- `height`: the person's height in inches.

The function should return two objects:

- `bmi`: the person's BMI (not rounded, unlike in Exercise 1.4).
- `classification`: whether the person is "Underweight", "Normal", "Overweight", or "Obese", as in Exercise 1.4.

Sample input 1:

```
bmi,classification=calculateBMI(145,71)
print('BMI:',bmi)
print('classification:',classification)
```

Correct output:

```
BMI: 20.21964566293356
classification: Normal
```

Sample input 2:

```
calculateBMI(160,60)
```

Correct output:

```
(31.24206399596875, 'Obese')
```

Sample input 3:

```
weight=float(input('Enter weight (lbs): '))
height=float(input('Enter height (inches): '))
bmi,classification=calculateBMI(145,71)
print(f"The person's BMI is {round(bmi,2)}, which is {classification}.")
```

Sample output (as in Exercise 1.4, output depends on what user inputs):

```
Enter weight (lbs): 145
Enter height (inches): 71
The person's BMI is 20.22, which is Normal.
```

Exercise 1.11: Blood Sugar Checker

Write a function called `bloodSugarCheck` with two input arguments:

- `hours`: the number hours the patient has fasted.
- `level`: the patient's blood sugar level.

The function should print a message based on the following logic:

- If they have fasted strictly less than 2 hours, then print "You need to fast at least 2 hours to perform this test."
- If they fasted at least 2 hours but strictly less than 8 hours, then print "Your blood sugar level is high" if `level` is strictly more than 140, and "Your blood sugar level is normal" otherwise.
- If they fasted for at least 8 hours, then print "Your blood sugar level is high" if `level` is strictly more than 100, and "Your blood sugar level is normal" otherwise.

Include the doc-string "Function for triaging a patient's blood sugar level based on hours of fasting and current level."

Sample run 1:

```
bloodSugarCheck(1,100)
```

Correct output:

```
You need to fast at least 2 hours to perform this test.
```

Sample run 2:

```
bloodSugarCheck(2,110)
```

Correct output:

```
Your blood suguar level is normal.
```

Sample run 3:

```
bloodSugarCheck(2,141)
```

Correct output:

```
Your blood sugar level is high.
```

Sample run 4:

```
bloodSugarCheck(8,110)
```

Correct output:

```
Your blood sugar level is high.
```

Sample run 5:

```
help(bloodSugarCheck)
```

Correct output:

```
Help on function bloodSugarCheck in module __main__:

bloodSugarCheck(hours, level)
    Function for triaging a patient's blood sugar level based on hours of fasting and current level.
```