Data Mining 4th year Software Engineering

Duration: 3 hours 2025/2026

TP1 : Introduction to Python

1. The first notebook

```
print('This isnt a string of text')
print("This isn't a string of text")
print(3 + 5)
```

You can assign a numerical value or string of text to an object in Python, and then interact with it.

```
 greeting = 'Hello and welcome to the course '
print(greeting)
```

```
a = 90
b = 12
print(a + b)
```

2. Variable Types in Python

Python has 4 native variable types -- integers, floats, strings, and booleans. You can determine the variable type using the built-in `type()` function.

```
# Print different object types
print(type(3))
print(type(3.5))
print(type('a'))
print(type(True))

# You'll notice this doesn't print everything
type(3)
type(3.5)
```

3. Mathematical Operations

You can perform mathematical operations on each of the different data types here.

```
# Print mathematical operations
print(3 + 5)
print(3.4 + 4.5)
print('a' + 'b' + 'c' + 'd')
print(True + True)
```

You can add, subtract, multiply and divide numerical values in Python. You can also denote an exponent like this one, 3^2 , using two stars `3**2`.

```
# Add, subtract, multiply, divide values

# subtraction
print(3.3 - 1.1)

# multiplication
print(3 * 2)

# division
print(4 / 2)

# exponent
print(3**10)

print(True*True*False)
```

What happens if you multiply **a** and **b** in the following cell? Subtract them?

```
a = 'Hi'
b = 9

print(a*b)
```

What do the following errors mean? How do you begin to investigate the cause?

```
print(This is a course)

print(a**b)

print()
```

4 Data types

- Numbers
- Strings
- Lists
- Dictionaries
- Booleans
- Tuples
- Sets

- Comparison Operators
- if,elif, else Statements
- for Loops
- while Loops

4.1 Numbers and more in Python

4.1.1 Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers.

Python 3 Alert

What just happened? Last time I checked, 3 divided by 2 is equal 1.5 not 1

The reason we get this result is because we are using Python 2. In Python 2, the / symbol performs what is known as "**classic**" division, this means that the decimal points are truncated (cut off). In Python 3 however, a single / performs "**true**" division. So you would get 1.5 if you had inputed 3/2 in Python 3.

So what do we do if we are using Python 2 to avoid this?

Specify one of the numbers to be a float:

```
# Specifying one of the numbers as a float

3.0/2

float(3)/2

# Order of Operation

(2 + 3) * (5 + 5)
```

4.1.2 Variable Assignments

We use a single equals sign to assign labels to variables.

The names you use when creating these labels need to follow a few rules:

```
1. Names can not start with a number.
2. There can be no spaces in the name, use _ instead.
3. Can't use any of these symbols :'",<>/?|\() @#$%^&*~-+
3. It's considered best practice (PEP8) that the names are lowercase.
```

For example:

```
# Can not start with number or special characters
name_of_var = 2

x = 2
y = 3

z = x + y

z

# Use object names to keep better track of what's going on in your code

my_income = 100

tax_rate = 0.1

my_taxes = my_income*tax_rate

# Show my taxes
my_taxes
```

4.2 Strings

Strings are used in Python to record text information, such as name. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

4.2.1 Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

```
# Single word

'hello'

# Entire phrase

'This is also a string'

# We can also use double quote

"String built with double quotes"

 I'm using single quotes, but will create an error'
```

The reason for the error above is because the single quote in I'm stopped the string. You can use combinations of double and single quotes to get the complete statement.

4.2.2 Printing a String

Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

```
x = 'Hello DM Class, This is my first Python Program'

x

print(x)

num = 12
name ='Mo'

print('My number is: {one}, and my name is: {two}'.format(one=num,two=name))

# Use curley Brakcet

print('My number is: {}, and my name is: {}'.format(num,name))

# We can simply declare a string
'Hello World'

# note that we can't output multiple strings this way
'Hello World 1'
'Hello World 2'

print 'Hello World 1'
print 'Hello World 2'
print 'Use \n to print a new line'
print '\n'
print 'See what I mean?'
```

4.3 Lists

Lists in Python have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above). Lists can be thought of the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed.

Lists are constructed with brackets [ ] and commas separating every element in the list.

4.3.1 Creating lists

```
x = [4,2,6,3] #Create a list with values
y = list() # Create an empty list
y = [] #Create an empty list
print(x)
print(y)
```

```
# Assign a list to an variable named my_list
my_list = [1,2,3]

# ust like strings, the len() function will tell you how many items are in
the sequence of the list.

len(my_list)
```

4.3.2 Manipulating Information in Lists

Remember, lists are *mutable*, meaning we can change their length and contents.

- `list.remove()`, which removes elements from a list based on the *value* (not the index).
- `list.pop()`, which removes elements from a list based on the index, (not the value).
- `list.append()`, which adds a new element to the end of a list. If you want to add multiple elements at once, you can use `list.extend()`.
- `list.insert()`, which allows you to insert new values into a list given a specific index. This method takes 2 arguments -- the position (index), and the value you want to insert.

Indexing and Slicing

Indexing and slicing works just like in strings. Let's make a new list to remind ourselves of how this works:

```
my_list = ['a','b','c']

# Add element to the list

my_list.append('d')

# Print the list

my_list

# Grab element at index 0

my_list[0]

# Grab element at index 1

my_list[1]

# Grab everything UP TO index 1  - Using Colon

my_list[:1]



# Reassign
my_list = my_list + ['add new item permanently']
```

```
my_list

# Change an element of our data

my_list[0] = 'NEW'

my_list

# Make the list double
my_list * 2

x=[1,7,2,5,3,5,67,32]
print(len(x))
print(x[3])
print(x[2:5])
print(x[-1])
print(x[::-1])
```

4.3.3 Retrieving information from a `list`

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 'cat', 'dog', 6]

print(a[0])
print(b[2])

# Note that Python leaves out the final number in your index slicing
print(a[1:3])
print(b[1:3])

print(a[-2])
print(b[-2])
```

4.3.4 Adding items to a list

```
x=list()
print(x)
x.append('One') #Adds 'One' to the back of the empty list
print(x)
x.append('Two') #Adds 'Two' to the back of the list ['One']
print(x)
```

```
[]
['One']
['One', 'Two']
```

```
x.insert(0,'Half') #Inserts 'Half' at location 0. Items will shift to make
roomw
print(x)
```

```
['Half', 'One', 'Two']
```

```
x=list()
x.extend([1,2,3]) #Unpacks the list and adds each item to the back of the
```

```
list
print(x)
```

```
[1, 2, 3]
```

4.3.5 Removing items from a list

```
x=[1,7,2,5,3,5,67,32]
x.pop() #Removes the last element from a list
print(x)
x.pop(3) #Removes element at item 3 from a list
print(x)
x.remove(7) #Removes the first 7 from the list
print(x)

x.remove(20)
```

4.3.6 Mutablility of lists

```
y=['a','b']
x = [1,y,3]
print(x)
print(y)
y[1] = 4
print(y)

print(x)

x="Hello"
print(x,id(x))
x+=" You "
print(x,id(x)) #x is not the same object it was
y=["Hello"]
print(y,id(y))
y+=["You "]
print(y,id(y)) #y is still the same object. Lists are mutable. Strings are
immutable

def eggs(item,total=0):
    total+=item
    return total


def spam(elem,some_list=[]):
    some_list.append(elem)
    return some_list

print(eggs(1))
print(eggs(2))

print(spam(1))
print(spam(2))
```

4.3.7 Nesting Lists

A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

```
# Let's make three lists
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]

# Make a list of lists to form a matrix
matrix = [lst_1,lst_2,lst_3]

# Show
matrix
```

4.4 Tuples

In Python tuples are very similar to lists, however, unlike lists they are **immutable** meaning they can not be changed. You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

4.4.1 Constructing Tuples

The construction of a tuples use ( ) with elements separated by commas. For example:

```
# Create a tuple with 1 item, needs a comma after the item
a_tuple = (3,)
print (a_tuple)
# Create a tuple with several items
t = (1, 2, 3, 'cat', 'dog')
print(t)

x = (1,2)

# Print items by index
print(t[3])

alist = [1,2]
new_tuple = tuple(alist)
new_tuple
```

Let's take a look at what it looks like to define a function that returns a `tuple`

```
from statistics import mean, median

# Rather than making two functions, I can return both the mean() and
# the median() to the calling code using a tuple
def summary(a_list):
    return (mean(a_list), median(a_list), min(a_list), max(a_list))
```

```
list_mean, list_median, list_min, list_max = summary([1, 3, 5, 7, 900])

print(list_mean)
print(list_median)
print(list_min)
print(list_max)

# Can create a tuple with mixed types
t = (1,2,3)

# Check len just like a list
len(t)

# Can also mix object types
t = ('one',2)

# Show
t

# Use indexing just like we did in lists
t[0]

# Slicing just like a list
t[1]
```

4.4.2 Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do. Lets look at two of them:

```
# Use .index to enter a value and return the index
t.index('one')

# Use .count to count the number of times a value appears
t.count('one')
```

4.5 Dictionaries

So what are mappings? Mappings are a collection of objects that are stored by a *key* .

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

4.5.1 Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work

The structure of a dictionary is made clear by the syntax for creating one:

```
employee = {
    'name':'Jane Smith',
    'title':'Data Engineer',
```

```
    'building_no': 3,
    'projects':['data warehouse', 'new airflow view', 'webinar']
}
print(employee)
```

Alternatively, your keys could be numeric. Here we store names by employee ID.

```
employees = {
    12342: 'Stanley Kehrer',
    32111: 'Jane Smith'
}
print(employees)
```

Working with dictionaries is straightforward, as long as we remember the syntax to reference a value in a dictionary using `my_dict['my_key']`

```
# Read the value of a dictionary:
print(employee['name'])

# Add a new value to a dictionary (Or update existing one )
employee['status'] = 'employed'
employee['building_no'] = 4
print(employee)
```

Use the **`del`** statement to remove entries from a dictionary.

```
my_dict = { 'a': 15, 'b': 32 }

del my_dict['a']
print(my_dict)

new_dict = dict(name="Kristen Kehrer", age=14)
print(new_dict)

mktcaps = {'AAPL':538.7,'GOOG':68.7,'IONS':4.6}

mktcaps['AAPL'] #Returns the value associated with the key "AAPL"

mktcaps['GS'] #Error because GS is not in mktcaps

mktcaps.get('GS') #Returns None because GS is not in mktcaps

mktcaps['GS'] = 88.65 #Adds GS to the dictionary
print(mktcaps)

del(mktcaps['GOOG']) #Removes GOOG from mktcaps
print(mktcaps)

mktcaps.keys() #Returns all the keys

mktcaps.values() #Returns all the values
```

```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}

# Call values by their key
my_dict['key2']

my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

#Lets call items from the dictionary
my_dict['key3']

# Can call an index on that value
my_dict['key3'][0]

#Can then even call methods on that value
my_dict['key3'][0].upper()
```

4.5.2 A few Dictionary Methods

There are a few methods we can call on a dictionary. Let's get a quick introduction to a few of them:

```
# Create a typical dictionary
d = {'key1':1,'key2':2,'key3':3}

# Method to return a list of all keys
d.keys()

# Method to grab all values
d.values()
```

4.6 Set and Booleans

4.6.1 Sets

Sets are an unordered collection of **unique** elements. We can construct them by using the set( ) function. Let's go ahead and make a set to see how it works

```
x = set()

# We add to sets with the add() method
x.add(2)

#Show
x

 # Add a different element
x.add(2)

#Show
x

# Try to add the same element
x.add(1)
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements We can cast a list with multiple repeat elements to a set to get the unique elements.

4.6.2 Booleans

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None.

```
# Set object to be a boolean
a = True

#Show
a
```

We can also use comparison operators to create booleans. We will go over all the comparison operators later on in the course.

```
# Output is boolean
1 > 2
```

5 Comparison Operators

5.1 Equal

```
2 == 2
```

```
1 == 0
```

5.2 Not Equal

```
2  = 1
```

```
2  = 2
```

5.3 Greater Than

```
2 > 1
```

```
2 > 4
```

5.4 Less Than

```
2 < 4
```

```
2 < 1
```

5.5 Greater Than or Equal to

```
2 >= 2
```

```
2 >= 2
```

5.6 Less than or Equal to

```
2 <= 2
```

2.7 Logic Operators

```
(1 > 2) and (2 < 3)

(1 > 2) or (2 < 3)

(1 == 2) or (2 == 3) or (4 == 4)
```

6 if,elif,else Statements

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action 3
```

6.1 First Example

```
if True:
    print 'It was true '

x = False

if x:
    print 'x was True '
else:
    print 'I will be printed in any case where x is not true'
```

6.2 Multiple Branches

```
loc = 'Bank'

if loc == 'Auto Shop':
    print ('Welcome to the Auto Shop ')
elif loc == 'Bank':
    print ('Welcome to the bank ')
else:
    print ("Where are you?")
```

7. Iteration

7.1 Range iteration

```
#range(len(x)) generates values from 0 to len(x)
x=[1,7,2,5,3,5,67,32]
for index in range(len(x)):
    print(x[index])
list(range(len(x)))
```

7.2 List element iteration

```
x=[1,7,2,5,3,5,67,32]
for element in x: #The for draws elements - sequentially - from the list x
and uses the variable "element" to store values
    print(element)
```

7.3 Loops

Essentially, loops allow you to run Python code multiple times.

```
mylist = [1,2,3,45,123,400]

for e in mylist:
    print("Running:")
    print(e * 1000)

otherlist = []

for element in mylist:
    otherlist.append(element * 100)
print(otherlist)

# Loop from 0 to 10, (excluding 10)
for i in range(10):
    print(i)

# Loop from 10 to 20
for i in range(10, 20):
    print(i, end='')
print()

# Loop from 0 to 20, evens only
for i in range(0, 20, 2):
    print(i, end='')
#print()
```

7.4 While loops

Evaluate the condition and run the code until it's false

```
def mult_to_mil(num):
    if num < 0:
        return -99
    else:
        while num < 1000000:
            num *=10
        return num

#mult_to_mil(5)
mult_to_mil(-5)
```

7.5 for Loops

A **for** loop acts as an iterator in Python, it goes through items that are in a *sequence* or any other iterable item.

```
for item in object:
    statements to do stuff
```

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

for item in seq:
    print(item)

for item in seq:
    print('Yep')

for jelly in seq:
    print(jelly+jelly)
```

7.6 while loops

The general format of a while loop is:

```
while test:
    code statement
else:
    final code statements
```

Let's look at a few simple while loops in action.

```
x = 0

while x < 10:
    print ('x is currently: '),x
    print (' x is still less than 10, adding 1 to x')
    x+=1
```

Notice how many times the print statements occurred and how the while loop kept going until the True condition was met, which occurred once x==10. Its important to note that once this occurred the code stopped. Lets see how we could add an else statement

```
x = 0

while x < 10:
    print ('x is currently: '),x
    print (' x is still less than 10, adding 1 to x')
    x+=1

else:
    print ('All Done ')
```

7.7 break, continue, pass

We can use break, continue, and pass statements in our loops to add additional functionality for various cases. The three statements are defined by:

break: Breaks out of the current closest enclosing loop.
continue: Goes to the top of the closest enclosing loop.
pass: Does nothing at all.

```
while test:
    code statement
    if test:
        break
    if test:
        continue
else:
```

```
x = 0

while x < 10:
    print ('x is currently: '),x
    print (' x is still less than 10, adding 1 to x')
    x+=1
    if x ==3:
        print ('x==3')
    else:
        print ('continuing...')
        continue
```

```
x = 0

while x < 10:
    print ('x is currently: '),x
    print (' x is still less than 10, adding 1 to x')
    x+=1
    if x ==3:
        print ('Breaking because x==3')
        break
    else:
        print ('continuing...')
        continue
```