

Introductions to applications in python for economists

Nhat Luong, Kassel University

Lecture 1 (06/11/2020)

I am an economist, why Python?

*** Statistical tool (comparing to R, Eview, Stata)**

- Experimental economist (Game dev)
- Job perspective
- Beginner friendly and has large community (fashion and trend is important)

A case comparison: R vs Python

Popularity: Python (**BUT**)

Exploratory data analysis: R

Speed: Python (but?)

Graphics & Visualization: R (but?)

Flexibility: Python

=> It depends on your aim

What is this course NOT about?

- Computer Science (Data Structure & Algorithm)
- Statistical/mathematic course
- Certification as (python) developer
- Certification as data analyst

What can I claim after this course? “Course Aim”

- “Yes, I know how to program this in Python”

OR

- “I know how to get this or that done in Python”

Asking questions and self-study

Profile:

Master of Science in Computer Science

December 2016

Focus: Software engineering

Languages: Java, Python, Ruby, Rails, R, SQL, PHP, HTML, JavaScript, CSS

Question posted in 2018:

“...But I get the following error when I reset the database

```
var1 = models.IntegerField(widget=widgets.Slider(attrs={'step': '0.01'}), show_value=FALSE)
```

NameError: name 'FALSE' is not defined”

Questions (at least at this level of this course) should not be aimed to “solve” the problem at hand rather to understand it.

I posted a question in Stack Overflow and the community is quite unfriendly (or even toxic).

>> Learn how to find your own answers with a search first. Most of the time you will find an answer.

Quiz

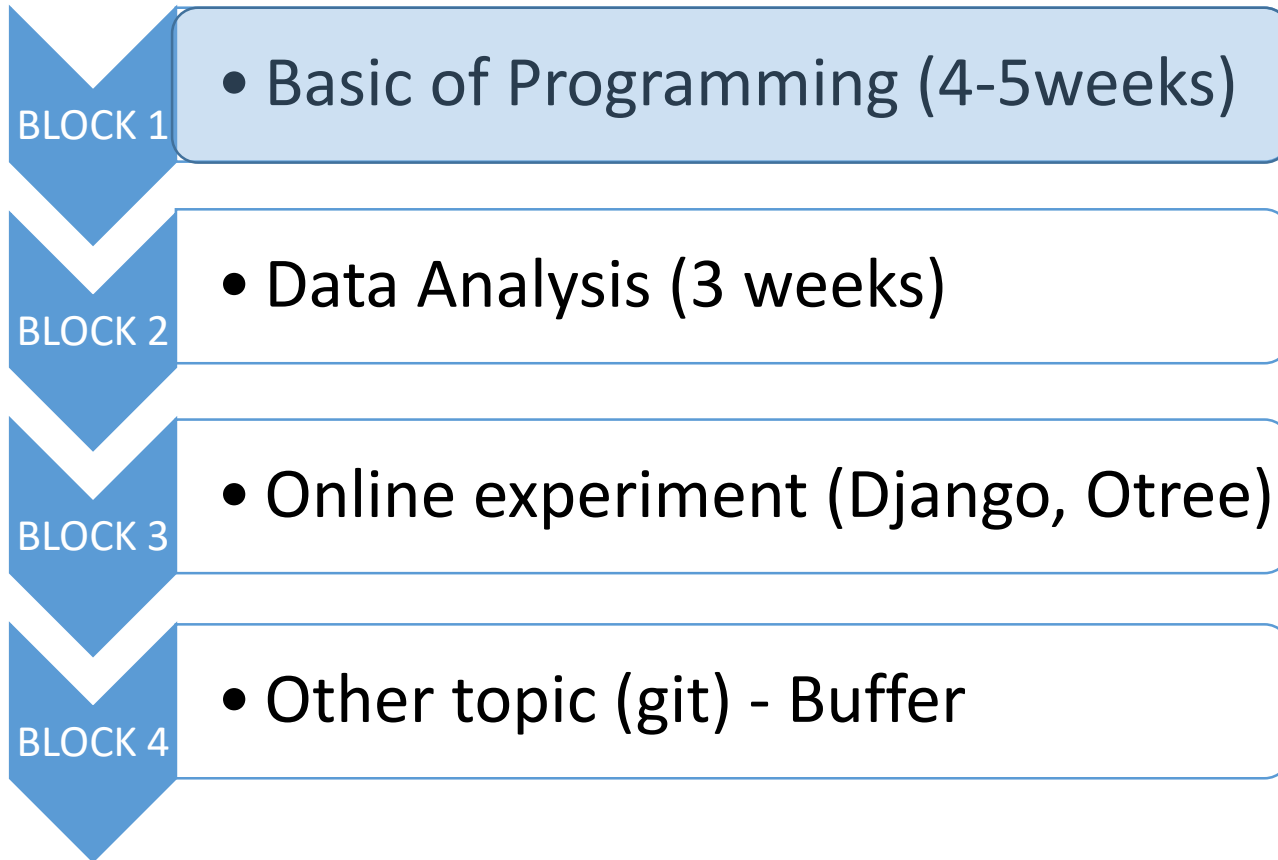
- Throughout the lecture you will see some google form quizzes. The link will be posted in chat. Here is an example:

<https://forms.gle/Yk6kt3W3jUQtv7Pt9>

Projects and Exams

- There will be 3 projects throughout the course.
- You will have around 3 weeks to submit the answers to 2 out of 3 projects in order to gain the right to enter the exam.
- Only 2 projects with the highest scores will be calculated.
- This applies to all students (including those from previous semesters).
- The projects are individualized (there is no two projects that are completely the same).

Course Contents



BLOCK 1: Contents

- What is Python ? Installation and IDE
- Variable and Common Data Types
- List and Working with List
- If Statement
- Dictionary
- Input and While-loop Function
- Classes
- Handling Exceptions and Files
- Code Testing

BLOCK 1.1 What is Python?

- A programming language is a set of rules for giving instructions to a computer.
- Python's programming styles and philosophies are simplicity, readability and getting the job done (and fun)

=> Perfect for wider range of audience that are not computer scientists/enthusiasts

BLOCK 1.1 Some terminology

- ABSTRACTION <we talk about this in more details later>
- Execute & operate
- Algorithm
- Framework
- Convention

BLOCK 1.1 Installation and IDE

- Python version 3.6 and above
- Writing your code (text editor and IDE - integrated development environment):

Sublime text editor

Pycharm

Others (IDLE, Jupiter Notebook, Atom)

BLOCK 1.1 Installation and IDE

- Installing Python on different major platform (windows and mac)
 - Check if python is installed
 - Update version (<https://python.org/>)
 - Installing IDE
- Run your file from terminal
 - Change to the directory using “cd”
 - Check if you can see your file using “dir” (“ls”)
 - Type “python name.py”

BLOCK 1.1 What if nothing works?

- Traceback – Error report.
- Stop and go away.
- Write your code from the beginning again
- Describe your steps and ask your friend to follow through it.
- Ask people who knows programming (not necessarily Python). They will most likely points you to the documents, don't expect that they solve the problem for you.
- Asking online.

Block 1.1 Motivate yourself

Think about 3 programs (anything literally) that you would like to build if you were fluent in programming.

=> Programming is not the solution. It is only a tool and you need a good problem as well.



efficiently



responsively



correctly

Now we actually start programming!



BLOCK 1.2 Variable and common data types

- Create a file named hello_world.py

```
>>> print("Hello World!")
```

- Close the file and run it in window command

⇒ Python interpreter

⇒ Syntax highlighting

BLOCK 1.2 Variables

- Add a variable to your “hello_world.py” file
- Print the variable

⇒ Every variable is connected to a value

- Change value of a variable

=> Only current value is kept track

BLOCK 1.2 Some rules about naming variables

- Variable names can only contain: letters, numbers and underscores (NO :'" ,<>/? |\!@#%^&*~ -+)
- Spaces are NOT allowed
- Avoid using Python keywords and function in naming. Also no: `variable = <something>`
- Name should be short but descriptive
- Special case with l (lower case) and O (upper case), this looks like 1 and 0
- Convention: no Uppercase at beginning!

BLOCK 1.2 A common error in naming variable

```
>>>word = "Hello world!"
```

```
>>>Print(wor)
```

⇒ Traceback is provided.

```
>>>wor = "Hello world!"
```

```
>>>print(wor)
```

⇒ Not spelling error

BLOCK 1.2 Strings (Data type)

- A string is a series of characters
 - Anything inside a quote is a string (‘ ’ or “ ”)
- ⇒ This is very useful as you can put quotes within string
- Case changing with string
 - Variables in string
 - Whitespace, tab, new line
 - Stripping whitespace

BLOCK 1.2.1 Case changing with string

```
>>>book_title = "alice in wonderland"
```

```
>>>print(book_title.title())
```

⇒A method is an action that can be performed on a piece of data. It's start with a dot (.) after a piece of data. For example: *.uppercase()* and *.lowercase()*

⇒Method usually need more parameters, provided inside the parenthesis.

BLOCK 1.2.2 Variables in string

```
>>> pet_1 = "cat"
```

```
>>> pet_2 = "dog"
```

```
>>> sentence_pet = f"I have a {pet_1} and a {pet_2}"
```

```
>>> print(sentence)
```


BLOCK 1.2.3 Whitespace in string

- Whitespace is any nonprinting character (space, tab, end of line)
- Escape characters:

```
>>>print("\tPython") #tab
```

```
>>>print("Animals:\ncat\ndog")
```

BLOCK 1.2.4 Stripping Whitespace

```
>>>pet_1 = ' cat '
```

```
>>>pet_1.rstrip()
```

```
'cat'
```

```
>>>pet_1.lstrip()
```

```
'cat '
```

```
>>>pet_1.strip()
```

```
'cat'
```

BLOCK 1.2.4 Syntax Errors with Strings

- A *syntax error* occurs when section of your program is not seen.
- as a valid (Python) code

BLOCK 1.2 Numbers

Symbol	Functionality
+	Addition
-	Minus
*	Multiplication
/	Division
%	Modulo (remainder after division)
**	Exponentiation
//	Floor division towards negative infinity
abs(a)	Absolute value

Truncate towards zero can be achieved by using `int()` for the result of normal division, for example:

```
print(int(-4/3))
```

BLOCK 1.2 Numbers

- Writing long number:

```
>>> large_num = 20_000_000_000
```

```
>>> print(large_num)
```

```
20000000000
```

- Multiple assignment:

```
>>> x, y, z = 0, 0, 0
```

- Constants:

```
>>> PORT = 9999
```

BLOCK 1.2 Comments

- Denoted using hash mark (#)
- Extremely useful but often underrated and neglected by beginners.
- Save time, make code structures more clear
- Enhance collaboration

=> For now, don't be afraid of long comments. It's easy to delete

BLOCK 1.2 The Zen of Python

```
>>> import this
```

- Those are principles of good Python codes

Beautiful is better than ugly

Simple is better than complex

Complex is better than complicated

Readability counts

There should be one– and preferably only one –obvious way to do it

Now is better than never

BLOCK 1.3 List

- A list is a collection of items in a particular order.
- List is a type of data
- List is *ordered* collection, each items in list has index number
- Items in a list can be accessed using index number.

```
>>> print(students[0])
```

- Individual items from a list can be used similar to a variable

BLOCK 1.3 Modifying List

- Changing/replace items in list

```
>>> students[1] = 'michael'
```

- Adding new item in list

```
>>> students.append('sarah')
```

- Inserting new item in a position

```
>>> students.insert(0, 'jimmy')
```

BLOCK 1.3 Removing items in list

- Removing item in a defined position

```
>>> del students[0] #no access to removed value
```

- Removing item but use that item

```
>>> removed_stud = students.pop()
```

- Removing item using value, only first occurrence!

```
>>> students.remove('sarah')
```

BLOCK 1.3 List sorting and order

```
>>>students.sort()
```

- .sort() will order the items in list in an increasing order. The change in order is permanent.
- .sorted() will order the items in list but only for the purpose of displaying, changes will not be implemented in list.
- .reverse() will simply reverse the list, changes are permanent
- len() function will give the number of items in list

BLOCK 1.3 Common errors when using list

- IndexError: index of the item does not exist
⇒ Of by one mistakes, indexing start with 0
⇒ The list is empty

Solution when can't resolve:

- Print the list or length of the list

Exercises

1. Create a simple equation where *my_taxes* is equal to *my_income* multiply with *tax_rate*. Calculate your taxes by assigning values to *tax_rate* and *my_income*
2. Assign a message to *message_shout* variable and print it. Change the value of the variable and print it again.
3. Assign your name to a variable. Print that name in lowercase, uppercase, and title case.
4. Find a famous quote, print the quote with citation marks.
5. Assign the author name of the quote in exercise 4 to a variable. Then concatenate the name into the quote. Print the result. The result should look like this:
Alan Turing once said, “.....”
6. Make a variable of your name with whitespace using “\t” and “\n”. Print your name, then try stripping whitespace using `lstrip()`, `rstrip()`, and `strip()`.

Excercises

7. Directly print out a result of a addition, subtraction, multiplication and division operations using print without assigning any variables.

8. Assign a variable with a 2 digits number (any number you like, e.g. 44). Use string formatting, print out a phrase like:

Your seat number is 44

9. Store few names of your classmates in a list called *names*. Print each name in a list with by accessing each item in the list one by one using index. The result should look like this:

- *We have....Alisa in our class*
- *We have...Thomas in our class*
- *We have...Ada in our class*

Note that there's a tab before the minus sign in each line

10. Make a list of names you'd like to invite for dinner. Call this list *guests*. Add 3 new guests in the list by:

- i) Adding one to the middle of the list
- ii) Adding one to the end of the list
- iii) Adding one to the beginning of the list

Hint: Use `insert()` and `.append()`

Exercises

11. Using the list in exercise 10. First, print out an apology saying you can only invite 3 to the dinner. Then

- (i) Remove guests from your list one by one. Each time you remove a guest print a sorry message: "Sorry, I can't invite Thomas". Do this till you got 3 guests left in the list.
- (ii) Print message to the 3 people that are still in your list and saying they are still invited
- (iii) After the dinner, you now want to remove the last 3 guests in the list by using `del()`, you should now have an empty list. Check by print out the list



12. Make a list containing names of famous locations.

- (i) Use `sorted()` and print out the result of the list that have been sorted and the original list.
- (i) Use `sort()` two times, print out the result each time.
- (ii) Use `reverse` two times, print out the result each time.

Introductions to applications in python for economists

Nhat Luong, Kassel University
Lecture 2 (13/11/2020)

BLOCK 1: Contents

- What is Python ? Installation and IDE
- Variable and Common Data Types
- List and Working with List 
- If Statement 
- Dictionary
- Input and While-loop Function
- Classes
- Handling Exceptions and Files
- Code Testing

BLOCK 1.3 List Quick Recap of the last lecture (new content starts in page 9)

- Individual items from a list can be used similar to a variable

```
>>> print(f"The last student is {student[-1].title()}.")
```

- Item in list can be swap if assign value in the same line of code. (The assignment happened at the same time)

```
>>> sample_list = [1,5]
```

```
>>> sample_list[0], sample_list[1] = sample_list[1], sample_list[0] #MUST BE same code line
```

```
>>> print(sample_list)
```

Result: [5,1]

BLOCK 1.3 Modifying List

- Changing/replace items in list

```
>>> students[1] = 'michael'
```

- Adding new item in list

```
>>> students.append('sarah')
```

- Instead of .append() method, you can use concatenation, however, you can only concatenate 2 lists together.

```
>>> list1 = ['A','B']
```

```
>>> list2 = ['C']
```

```
>>> list3 = list1 + list2 #list3 will now have value ['A', 'B', 'C']
```

WARNING!: You cannot do this:

```
>>> list3 = list1.append(list2[0]) #This will result in None (undefined)
```

- Inserting new item in a position

```
>>> students.insert(0, 'jimmy')
```

BLOCK 1.3 Removing items in list

- Removing item in a defined position

```
>>> del students[0] #no access to removed value
```

- Removing item but use that item

```
>>> removed_stud = students.pop()
```

- Removing item using value, only first occurrence!

```
>>> students.remove('sarah')
```

QUIZ 1

<https://forms.gle/EtbAHYwpaCsAcZiS7>

BLOCK 1.3 List sorting and order

```
>>>students.sort()
```

- .sort() will order the items in list in an increasing order. The change in order is permanent. Method .sort() has no return value. This means, you would get value “None”, if you do something like:

```
>>>print(students.sort())
```

- sorted(), will order the items in list but only for the purpose of displaying, changes will not be implemented in list as sorted() is not a method.
.sort() or sorted() will sort first item/element in nested list
- .reverse() will simply reverse the list, changes are permanent
- len() function will give the number of items in list

BLOCK 1.3 Common errors when using list

- IndexError: index of the item does not exist
⇒ Of by one mistakes, indexing start with 0
⇒ The list is empty

Solution when can't resolve:

- Print the list or length of the list

BLOCK 1.3 Looping through list

```
>>> shopping_items = [tomatoes, potatoes, eggs]
>>> for item in shopping_items:
    print(item)
```

- For every item in shopping_items list, perform the following actions after the colons “:”. In this case, we only ask python to print out the items.
- “iteration” is a process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met.

BLOCK 1.3 More on looping

- Looping is the most common ways to manage repetitive tasks.

```
>>> for item in shopping_lists: #it is okay to use the word "item" here
```

- for <something> in <list>: means that we start with the first item in the list. Then we assign the value of that item to <something>. For example, the following codes do the same thing:

```
>>> for blah in shopping_list:  
    print(blah)
```

```
>>> for item in shopping_list:  
    print(blah)
```

- This <something> is only a temporal variable. That means it only exists inside the loop. You can't call it outside of the loop.

BLOCK 1.3 More on looping

- The loop will repeat the action(s) defined after the colon (":") the exact number of times as the number of items in the list. If a list contains billions items. The for loop will run billion times.
- Usually, temporal variable is named as a singular form of the list name. For example, if list name is `cats = []` . Then the temporal variable should be "cat".
- You can write as many actions as you want inside a loop. Watch for indentation!

QUIZ 2

<https://forms.gle/zWdyMLSVysm4h2id6>

BLOCK 1.3 More on indentation

- Python use indentation to determine how lines of code should be treated as a group. This makes codes easy to read in Python.
- However, this causes some indentation errors. For examples, indentation where it doesn't need indentation or forget to make indentations.
- Forget to indent

```
for item in shopping_items:  
print(item) #I made an intentional error here!
```

⇒IndentationError

- Forget to indent line of codes supposed to be within the loop

```
for item in shopping_items:  
    print(item)  
print("What's next?)
```

⇒No error shown (logical error)

BLOCK 1.3 More on indentation

- Indentation where it is not needed. This looks like it will work (it does on some IDEs), but the interpreter treats this as an error when you run a .py file.

```
message = "hello world"  
    print(message)
```

⇒IndentationError

```
for item in shopping_items:  
    print(item)  
    message = "hello world"
```

⇒Logical error

- * for-loop needs colon at the end. Otherwise, it will lead to syntaxError.

BLOCK 1.3 More on indentation

There are different practices when it comes to indentation.

PEP8 proposes 4 blank spaces for each level of indentation.

TAB can also be used for indentation.

Again, the rule is to be consistent! If you are using Pycharm or an IDE, you can change the settings to auto-convert TAB into spaces.



BLOCK 1.3 Quick numerical sequence (not list)

- List is usually used to contain set of numbers for economists. It can also be used to keep track of positions, or scores or payments when it comes to game development.
- If you want to generate a series of numbers with certain patterns, use `range([start], stop[, step])`.

```
>>> range(3) #notice that 3 is not included in the result
```

```
[0, 1, 2]
```

```
>>> range(1, 9, 2)
```

```
[1, 3, 5, 7]
```

*WARNING! The results above are not a list in Python 3.x, but it has its own type: “range”. It’s an immutable sequence and often used only as iterable.

BLOCK 1.3 Quick numerical list

- range can be converted to list by using list()

```
>>> number_list = list(range(6))  
>>> print(number_list)  
[ 0, 1, 2, 3, 4, 5]
```

*WARNING: In Python 3.5 and above, you may see:

```
>>> number_list = [* range(6)] #While this works, is it confusing? => Zen of Python
```

⇒ If your code doesn't work, make sure that you run it on the same Python the moment you developed your program successfully, or update your code.

BLOCK 1.3 Customized set of numbers with range iteration

- You can create any set of numbers of your liking using range() and for loop
- For example, a set of numbers containing the result of modulo by 2 of every odd number from 1 to 10.

```
>>> odd_mods = []  
>>> for value in range(1,10,2):  
    mod = value % 2  
    odd_mods.append(value % 2)  
>>> print(odd_mods)  
[ 1 , 1, 1, 1, 1]
```

=> Try to create identity matrix using this (exercise question)

BLOCK 1.3 Min, max, sum with list

```
>>> trees = [23, 4, 52, 43, 21, 5, 9]
>>> min(trees)
>>> max(trees)
>>> sum(trees)
```

- Min and max operations work for nested list also, however, the result is a sub-list:

```
>>> trees_fertilizer = [[23, True], [4, False], [52, True], [43, True],
[21, False], [5, False], [9, False]]
>>> min(trees_fertilizer)
[4, False]
>>> max(trees_fertilizer)
[52, True]
```

BLOCK 1.3 List comprehension

- Use to generate list in 1 line of code (simple list only)
- Works similar to as if you were using for loop with multiple lines

```
>>> for value in range(1,10,2):  
    mod = value % 2  
    odd_mods.append(value % 2)
```

Instead, you can “one-liner” it:

```
>>> odd_mods = [sushi % 2 for sushi in range(1,10,2)]
```

⇒ Not recommended for beginners, but worthwhile to learn and recognize it.

BLOCK 1.3 Slicing, looping and copying list

- Syntax of slicing is as follow: *[start:stop:step]*
- If you want to only work with part of list, you can slice it using index

```
>>> student_names = ['a', 'b', 'c', 'd']  
>>> print(student_names[0:2])  
['a', 'b']
```

- Omitting the first index means that the slicing will start from the beginning *[:2]* will have the same result as above.
- The same syntax applies for the end point.

```
>>> print(student_names[2:])  
['c', 'd']
```

BLOCK 1.3 Slicing, looping and copying list

- Similar with range(), step just add given number to your next index.

```
>>> print(student_names[::-2])  
['a', 'c']
```

- Negative step (without indicating starting index) will start at index -1.

```
>>> student_names = ['a', 'b', 'c', 'd']  
>>> print(student_names[::-2])  
['d', 'b']
```

- This also means [::-1] will reverse/flip the list
- If negative step is used and the [start] and [stop] index is given. The starting index will be the [start] index that was given. From then the next index will be [start] + [negative step]. For example, the following codes do the same thing:

```
>>> student_names = ['a', 'b', 'c', 'd', 'e', 'f']  
>>> print(student_names[4:1:-2])  
>>> print(student_names[-2:-5:-2])  
['e', 'c']
```

Quiz 3

<https://forms.gle/XWjKAN36G5Rs2H169>

BLOCK 1.3 Slicing, looping and copying list

- Instead of looping through the whole list, you can instruct Python to loop through subset of list using slicing.

```
>>> for name in student_names[:2]:  
    print(name)
```

- List can't be copied by assigning new variables. This only chain (give ref to) two list together, any changes in one list will happen to the other.

```
>>> list1 = ['a', 'b']  
>>> list2 = list1  
>>> list2.append('c')  
>>> print(list1, list2)  
['a', 'b', 'c'] ['a', 'b', 'c']
```

BLOCK 1.3 Tuples

- Immutable: Cannot change values
- Tuple is an immutable list

```
>>> screen_sizes = (800, 600)
```

```
>>> screen_sizes[0]
```

```
800
```

```
>>> screen_size[0] = 200
```

```
TypeError
```

- It's generally not recommended to create a tuple with only one element.
However, it might be a result of some codes

```
>>> single_tuple = (10, ) #This is called trailing comma
```


BLOCK 1.3 For loops with tuple

- Similar to list, you can loop over tuple with for list.

```
>>> screen_sizes = (800, 600)
>>> for size in screen_sizes:
    print(size)
```

- Although values in a tuple cannot be modified, a new variable assignment can still replace it!

```
>>> screen_sizes = (800, 600)
>>> screen_sizes = (3, 2)
```

BLOCK 1.4 If statement

- If statement let you check for condition(s) before taking action. For example, let's say you want to automatically check if your lunch food list contains milk and remove it from the list since you are lactose intolerant:

```
>>> lunch_foods = ['tomatoes', 'milk', 'noodles', 'beef']
>>> for food in lunch_foods:
    if food == 'milk': #execute only when condition is True
        lunch_foods.remove('milk')
```

- In this example, we use the equality check. Note that most programming languages define a conditional test for equality as double equal signs “==”, simply to separate it from “=” which is used for variable assignment.

BLOCK 1.4 Other conditional operators

Operator	Meaning
==	Equality check, True if both operands are equal
!=	Inequality check, True if both operands are NOT equal
>	Greater than, True if left operand is larger than right operand
<	Less than, True if left operand is smaller than right operand
>=	Greater than or equal to, True if left operand is greater or equal to right operand
<=	Less than or equal to, True if left operand is less or equal to right operand

Not to be confused with bitwise operators, which is somewhat similar, we will get back to this in Block 2!

BLOCK 1.4 Note on conditional test with string

- Conditional test is case sensitive. For example

```
>>> 'A' == 'a'
```

False

⇒ Use `.lower()` for string when you actually just want to compare meanings of words.

BLOCK 1.4 Multiple conditions

- To check whether two or more conditions are True simultaneously use “and” as a connector

```
>>> pet_1 = 'cat'
```

```
>>> pet_2 = 'dog'
```

```
>>> (pet_1 = 'cat') and (pet_2 = 'cat')
```

False

- The round brackets in the above codes are not required, programmers use them to make the code more readable. PEP8 also recommend it!
- To check whether one of a list of conditions is True, “or” is used as a connector.

```
>>> (pet_1 = 'cat') or (pet_2 = 'cat')
```

True

BLOCK 1.4 Multiple conditions

- “or” connector can be short-circuited. That means, as soon as it sees a condition satisfied, it will stop checking the rest. Code with side-effects could be effected by this Python design.
- De Morgan law: $A \text{ OR } B$ is equal to $\text{NOT} (\text{NOT}(A) \text{ AND } \text{NOT}(B))$

- To check whether an item exists in a list, simply use “in” connector.

```
>>> lunch_foods = ['tomatoes', 'milk', 'noodles', 'beef']  
>>> if 'milk' in lunch_foods:  
    print("Harry flips the table!")
```

- Similarly, to check whether an item does NOT exist in a list, simply use “not in”.

```
>>> if 'milk' not in lunch_foods:  
    print("Harry empties his lunch box!")
```

- A boolean expression like `'game_status = True'` can be very useful to track the state of a program or make a switch to turn it off entirely

BLOCK 1.4 Multiple conditions

This looks simple so far, however, you can make mistakes here very easily!

We have been looking only at 2 simultaneous conditions, things can get complicated easily as the number of conditions increases.

This is also important when you are applying multiple conditions to function with side effects. We will get to side effects in later chapters.

Sometimes, it is helpful to have a truth table. Below is the illustration of XOR:

EX-OR Gate Truth Table

<i>A</i>	<i>B</i>	<i>A \oplus B</i>
<i>0</i>	<i>0</i>	<i>0</i>
<i>0</i>	<i>1</i>	<i>1</i>
<i>1</i>	<i>0</i>	<i>1</i>
<i>1</i>	<i>1</i>	<i>0</i>

BLOCK 1.4 If-else statements and chains

```
>>> customer_age = 14
>>> if customer_age >= 18:
    print("Thank you for your order")
    print("Our product contains alcohol! Do not consume before driving")
```

- Similar to for loop, all the codes (which are indented) after the colon mark will be executed

If-else statements and chains

```
>>> customer_age = 14
>>> if customer_age >= 18:
    print("Thank you for your order")
else:
    print("Sorry, you are not old enough to purchase this item")
```

- When the condition is met (result in True), the first block of codes right after a colon is executed. When the condition is not met (result in False), the second block of code is executed instead.
- Note that if you formulate your code in an if-else chain like this, one of the two blocks will always be executed.

BLOCK 1.4 If-else statements and chains

```
>>> income_monthly = 500
>>> if income_monthly <= 450:
    print("You don't have to pay taxes")
elif income_monthly <= 677:
    print("You have to pay taxes but can get it back at the end of the
year")
else:
    print("You have to pay income taxes")
```

- Conditions are check with the priority from top to bottom. When the condition is met, the code stop executing.

BLOCK 1.4 If-else statements and chains

```
>>> income_monthly = 500
>>> if income_monthly <= 450:
    tax_rate = 0.0
elif income_monthly <= 677:
    tax_rate = 0.0
elif income_monthly > 5000:
    tax_rate = 10%
else:
    tax_rate = 9.45
```

- While this code still works, it makes it hard to read, the elif conditions should be ordered logically with a clear pattern. Coding without clear, stable pattern might lead to logical errors.
- The “else” block can be either: completely omitted or must be put at the end of the if-elif chain (otherwise there will be SyntaxError).
- Often, it makes sense to NOT use “else” at the bottom of the chain to make the condition clearer to read. Else block is a catchall statement, if none of the conditions preceding it has True value, else block will be executed (and sometimes the invalid data can slip through)

BLOCK 1.4 If-else statements and chains

- Consider using multiple if statements instead of if-elif chains when you want all the conditions met.

```
>>> available_toppings = ['pineapple', 'mushroom', 'pepperoni']
>>> if 'pepperoni' in available_toppings:
    print("+ peperoni")
>>> if 'mushroom' in available_toppings:
    print("+ mushroom")
>>> if 'pineapple' in available_toppings:
    print("+ pineapple")
```

BLOCK 1.4 Combining list with if statements

```
>>> available_toppings = ['pineapple', 'mushroom', 'pepperoni']
>>> ordered_toppings = ['mushroom', 'extra cheese', 'spinach', 'pineapple']
>>> for topping in ordered_toppings:
    if topping in available_toppings:
        print(f'We added {topping} on your pizza') #1
    else:
        print('Sorry, that topping is not available today!') #2
```

- The above codes iterate through each item in ordered_toppings list and check if each item is in the available_toppings list. If True, statement 1 is printed out. If False, statement 2 is printed out.

BLOCK 1.4 Combining list with if statements

- An empty list will result in False if checked by If statement, vice versa, a list that is not empty will result in True
- Avoid redundant coding. If the value of the condition is only True or False, you do not need to make an equality check with True or False.

```
>>> ordered_toppings = []  
>>> if ordered_toppings:  
    print("The list is not empty")  
else:  
    print("The list is empty")
```

BLOCK 1.4 Styling, conventions

- **Most important: BE CONSISTENT!**
- Easy to read codes that are following conventions are very useful to know (and follow)
- Python programmers have agreed on a number of rules and conventions to structure and style your codes. Full list of these rules are available in PEP8 (<https://www.python.org/dev/peps/pep-0008/>)
- Few important notes are:
 1. Again, indentation should be 4 spaces per cascading level. Some people use TAB instead, this can be a problem. However, most editors automatically convert TAB into spaces.
 2. Line length is limited to 79 characters. Some editors have a vertical line to indicate this limit
 3. Blank lines should be used to separate codes with different goals.
 4. There should be a space after each comma (e.g [4, 5, 6, 10])
 5. Comparison operators should also be separated with space : (e.g 4 > 3 not 4>3)
 6. Do not put unnecessary blank spaces, this is a bad habit, for example:
for item in item_list: _ <- The green underscore here represent a blank space
print _ (something) _

Introductions to applications in python for economists

Nhat Luong, Kassel University
Lecture 3 (20/11/2020)

Mutability

- Mutable: Can be changed in its place, meaning changes can be implemented on itself, no new variable is created.
- Immutable: Can never be changed.
- Common immutable objects are: int, float, decimal, Boolean, string, tuple, and range.
- Common mutable objects in Python are list, dictionary and set.
- Function `id()` is a good way to find out if an object is mutable or not

```
>>> x = 10
```

```
>>> y = x
```

```
>>> print(id(x) == id(y))
```

Aliasing

- One notable implementation of mutability is aliasing.
- Aliasing means using different names for an object. These different names point to an object in memory. This happens to mutable objects.
- When a change happens to an object in memory, you will see that change happens to all aliases (because they were all pointing to this object) – this is also called “side effect”. Note that “side effect” is a general term, it does not apply to only this case.

```
>>> list1 = [3, 2, 1, 5, 4, 0]
>>> list2 = list1
>>> print(list1, list2)
>>> list1.sort()
>>> print(list1, list2) #changes implemented on both lists, not just list1
```

Aliasing

- When you want to make a copy of a mutable object, make sure to check how to do so correctly.
- For example, you can make a copy of a list by using slicing:

```
>>> list1 = [3, 2, 1, 5, 4, 0]
>>> list2 = list1[:] #this is not referencing (aliasing) but a copy
>>> print(list1, list2)
>>> list1.sort()
>>> print(list1, list2)
```

- Avoid mutating objects as you iterating through it. Python does not update index numbers of a list when iterating through the list.

Aliasing

```
list1 = [1, 2, 3, 4, 5]
list2 = [1, 2]
```

```
for i in list1:
    if i in list2:
        list1.remove(i)
```

```
print(list1)
```

[2, 3, 4, 5] #incorrect

```
list1 = [1, 2, 3, 4, 5]
list2 = [1, 2]
```

```
for i in list1[:]: #copy of the list
    if i in list2:
        list1.remove(i)
```

```
print(list1)
```

[3, 4, 5] #correct

BLOCK 1: Contents

- What is Python ? Installation and IDE
- Variable and Common Data Types
- List and Working with List
- If Statement
- Dictionary
- Input and While-loop Function
- Classes
- Handling Exceptions and Files
- Code Testing



Block 1.5 Dictionary (dict)

- A simple example:

```
>>> student_grades = {'sarah': 1.7, 'ada': 1.0, 'thomas': 2.0}
```

- Each value in dict is associated with a key. Values can be any objects in Python. Keys must be immutable data types.
- In Python > 3.7 similar to list, dict has order, however, values in dict should be accessed only by key. There is also no index number to call a key-value pair in a particular position.
- To access a value in dict, we use key

```
>>> print(student_grades['ada'])
```

- To add new key-value pairs, we also use key

```
>>> student_grades['jim'] = 1.3
```

- Dictionary is an iterable.
- From Python 3.7, the order of value-key pairs are kept as the moment it was created. As a result, you can use index to get, says, the last key of a dict. However, again, dictionary shouldn't be used like a list.

```
>>> list(student_grades)[-1]
```

Block 1.5 Dictionary (dict)

- To modify values in list, simple reassign new values at specific key

```
>>> student_grades['jim'] = 2.0
```

- To remove the key value pair, you can use del.

```
>>> del student_grades['jim']
```

- Again, del is permanent, thus should be used with consideration
- By convention, dictionary should be written in the following style:

```
>>> student_grades = {  
    'sarah' : 1.7,  
    'ada': 1.0,  
    'thomas': 2.0,  
}
```

- By convention, the last pair should be ended with a comma when you style your dictionary this way.

Block 1.5 Dictionary (dict)

- In a large dictionary, you might not be able to keep track of keys. Thus, errors like this can happen:

```
>>> student_grades = {  
    'sarah' : 1.7,  
    'ada': 1.0,  
    'thomas': 2.0,  
}
```

```
>>> print(student_grades['jim'])
```

KeyError

- .get() can help to return a meaningful 'absent of key' value - None.

```
>>> print(student_grades.get('jim'))
```

- .get() can also be used to return any value/message when key is absent from the dictionary.

```
>>> print(student_grades.get('jim', 'No key found in dictionary'))
```


Quiz 1

Will be given during class

Block 1.5 Dictionary (dict)

- Dictionary is an iterable and can be loop through by values, keys or key-value pairs.
- Each key-value pair can be thought of as an item (similar to list). Thus, a way to loop through all items in a dictionary is simply to call `.item()` method.

```
>>> student_grades = {'sarah': 1.7, 'ada': 1.0, 'thomas': 2.0}
>>> for key, value in student_grades.items():
    print(f'Key: {key}')
    print(f'Value: {value}\n')
```

- Similarly, `.key()` is a method to call only keys in a dictionary.

```
>>> for key in student_grades.keys():
    print(f'\n Name: {key}')
```

- Looping through key is also the default when looping a dictionary. Thus, this code is the same as previous:

```
>>> for key in student_grades:
```

```
....
```

- Defining `.keys()` when looping through dictionary may be preferable as it is more explicit (Zen of Python)

Block 1.5 Dictionary (dict)

- .keys() and .items() method will not result in a list but it's own “special” type that can be iterate though and check with “in”. If list is what you want, use list().

```
>>> student_grades = {'sarah': 1.7, 'ada': 1.0, 'thomas': 2.0}
>>> if 'jim' in student_grades.keys():
    print('yes')
else:
    print('no')
```

- Notice that you can always refer the key back to dictionary, instead of using .items() method.

```
>>> for key in student_grades.keys():
    print(student_grades[key])
```

Block 1.5 Dictionary (dict)

- Since dictionary is kept in the order the key-value pairs were added. Sometimes you might want to change this order when looping through. One common way to change the order is to use `sorted()`.

```
>>> student_grades = {'sarah': 1.7, 'ada': 1.0, 'thomas': 2.0}
>>> for key in sorted(student_grades.keys()):
    print(key)
```

What happen when we use `student_grades.keys().sort()`? Why?

What happen when we use `list(student_grades.keys()).sort()`? Why?

Compare to this code:

```
>>> for key in student_grades.keys():
    print(key)
```

Block 1.5 Dictionary (dict)

- Similar to .keys() or .items() method in previous slides, .values() return a special iterable data type.

```
>>> student_grades = {'sarah': 1.7, 'ada': 1.0, 'thomas': 2.0, 'jim': 2.0}
>>> for value in student_grades.values():
    print(value)
```

- Dictionary usually can contain a large number of items. Thus, printing out all values are not so useful. Instead, one may be interested in what type of values the dictionary contains. set() can be used to obtain only unique values.

```
>>> for value in set(student_grades.values()):
    print(value)
```

- Set is mutable and can be created using {} bracket. Unlike list or dictionary, set does not has order (it has a built-in order, e.g. integer will be put in increasing order regardless of the initial arrangement). You can put in set any duplicated values, however, it will only retain unique value.

```
>>> setA = {1, 1, 3, 4, 5, 2}
>>> print(setA)
{1, 2, 3, 4, 5}
```

Block 1.5 Dictionary (dict)

- Similar to list, dictionary can be nested.
- Multiple dictionary can be nested inside a list:

```
>>> dict1 = {'a': 1, 'b': 2, 'c': 3}
>>> dict2 = {'d': 4, 'e': 5, 'f': 6}
>>> dict3 = {'g': 7, 'h': 8, 'i': 9}
>>> list_dicts = [dict1, dict2, dict3]
```

⇒ Very common practice to save dicts in list. Useful if you have similar structure of information for different objects. For example, user on a webpage.

- Multiple lists in a dictionary:

```
>>> applicants = {
    'michael': ['english', 'japanese', 'german'],
    'jen': ['korean', 'english'],
    'ada': ['english', 'latin', 'french'],
}
```

- When you starting to see yourself having a deeply nested list with many layers. The problem might be due to your design. Consider changing it to make your program more readable.

Block 1.5 Dictionary (dict)

- Dictionary can also be nested within dictionary:

```
applicants = {  
    'michael': {  
        'nationality': 'german',  
        'education_level': 'M.Sc',  
        'hobby': 'music',  
    },  
    'jen': {  
        'nationality': 'usa',  
        'education_level': 'B.Sc',  
        'hobby': 'painting',  
    },  
}
```

Block 1.6 Input

- So far, our programs are all one-directional. By asking for input from the user, we will begin to explore programs in a more interactive way.
- `input()` function can be used to record user's input in the form of text.
- whenever, the `input()` function is executed, the entire program pause and listen for the input from the user before continuing the rest of the program
- `input()` should be passed with a prompt:

```
>>> name = input("Please enter your username: ")  
>>> print(name)
```


Block 1.6 Input

- Any prompts that is more than one line should be put in a separate variable:

```
>>> prompt = "Please enter 'yes' in the field below if you agree with our conditions"
```

```
>>> prompt += "If you type 'no', you will be taken back to the home page: "
```

- The sign "+=" is a shortcut for concatenating previous value to the new value.

```
>>> answer = input(prompt)
```

```
>>> print(answer)
```

- input() function output is a string. Thus, answer must be converted to number type (e.g using int() or float()).

```
>>> tempt = input("What is the temperature right now? ")
```

```
>>> tempt = int(tempt)
```

```
>>> if tempt > 18:
```

```
    print("It's not so chilly there!")
```

```
elif tempt <= 18:
```

```
    print("It's chilly, wear warm clothes!")
```

Quiz 2

Will be given during class

Block 1.6 While-loop

- As long as a condition is True, codes wrapped in while-loop will keep repeating.

```
>>> flag = True
>>> i = 0
>>> while flag:
    print("hello!")
    i += 1
    if i == 10:
        print("that's enough!")
        flag = False
```

Block 1.6 While-loop

- Flag is used for complicated programs where multiple conditions can stop the program from running. By delegating all these conditions to a flag you can have a much more readable code.

```
>>> prompt = 'Give me a number: '  
>>> flag = True  
>>> user_input = ""  
>>> while flag:  
    user_input = input(prompt)  
    box = int(user_input)  
    print(box)  
    if box == 4:  
        print("That's a bomb!")  
        flag = False  
    elif box == 16:  
        print("It's a TREASURE!")  
        flag = False  
    else:  
        print("Nothing is here!")
```

Block 1.6 While-loop

- Combining with input, we can create while-loop that keeps asking for input until a specific input is found.

```
>>> basket = []
>>> prompt = "Please enter the item you want to put in your basket: "
>>> user_input = ""
>>> while user_input != 'stop':
    user_input = input(prompt)

    if user_input != 'stop':
        print(f'Added {user_input} to your basket')
        basket.append(user_input)
```

Block 1.6 While-loop

- Sometimes you want to stop everything in a loop without running the remaining codes, `break` can be used for this purpose. This work for any loops, not just while-loop.

```
>>> prompt = "Give me a number: "  
>>> while True:  
    user_input = input(prompt)  
    number = int(user_input)  
  
    if number == 0:  
        break  
  
    else:  
        print(f"Number {number}! I want another one")
```

Block 1.6 While-loop

- Instead of breaking the loop, you can also skip the rest of the codes and start the new cycle of the loop using **continue**:

```
>>> number = 0
>>> while number < 10:
    number += 1
    if number % 2 == 1:
        continue
    print(number)
```

Quiz 3

Will be given during class

Block 1.6 While-loop

- An accidental infinite loop is normally encountered when you start programming.
- Make sure that you test every loop. `print()` might help to troubleshoot your code as you can see the output printed out on the console.
- Each editor or IDE has its own “Stop” button, make sure you know where it is when you need it.

Block 1.6 While-loop

- With this new tool the “while-loop” you can combine it with previous data types that you’ve learned.
- For example, while-loops with list:

```
>>> basket = ['apples', 'spices', 'tomatoes']
```

```
>>> basket_confirmed = []
```

```
>>> while basket:
```

```
    item_in_basket = basket.pop()
```

```
    print(f"Confirming your purchase for item: {item_in_basket} ")
```

```
    basket_confirmed.append(item_in_basket)
```

Block 1.6 While-loop

- While-loop can also be used to remove a specific value from a list (all of them):

```
>>> trail_mix = ['peanut', 'almond', 'cashew', 'soybean', 'peanut', 'peanut']
```

```
>>> while 'peanut' in trail_mix:  
    trail_mix.remove('peanut')
```

```
>>> print(trail_mix)
```

- The combinations are endless. You can try to use a while-loop with dictionary types, list and every other data type that you have learned.

Introductions to applications in python for economists

Nhat Luong, Kassel University
Lecture 4 (8/11/2019)

Block 1: Contents

- What is Python ? Installation and IDE
- Variable and Common Data Types
- List and Working with List
- If Statement
- Dictionary
- Input, While-loop, Function
- Classes
- Handling Exceptions and Files
- Code Testing



Block 1.6 Functions

- Functions are simply a chunk of “named” codes that can be reused by calling it instead of typing all the codes again.
- Here’s the syntax to define a function:

```
>>>def give_list_1():  
    """ Return a list containing an element of value 1"""  
    #any additional codes here  
    return [1]
```

- **def**: is a keyword to signal to Python that I am going to create a function
- **give_list_1()** is a given name of the function, inside the round bracket, you can define variables that the function needs to do its job.
- **""" """** : is a docstring, it is used to describe what the function does.
- **return**: is used to end the function definition, it signals what kind of data the function will output, depending on your goal, this step can be skipped.

Block 1.6 Functions

- Here's an example of functions with arguments and parameters.

```
>>> def cube(length):  
    """ Return volume of a cube by giving its length."""  
    return length**3
```

```
>>> print(cube(10))
```

- The parameter here is `length`, whereas, number `10` is an argument.
- Argument is “passed” to function.

Block 1.6 Functions

- A function can require multiple parameters. There are 2 typical ways to pass arguments in these parameters.

(i) Positional arguments:

- Positions/orders matter.
- The order is determined by the programmer, thus docstring should be used to tell the user in which order the arguments should be passed.

```
>>> def intro_self(your_name, your_major):  
    """ Displaying name and study major"""  
    print(f"\nMy name is {your_name}.")  
    print(f"My major is in {your_major}.")  
>>> intro_self('ada', 'programming')  
>>> intro_self('michael', 'biology')
```


Block 1.6 Functions

- It is quite clear that you can have problems by just mixing the position of the required arguments:

```
>>> intro_self('programming', 'ada')
```

→ My name is programming.

→ My major is in ada.

(ii) Keywords argument:

- You can pass arguments together with its correct parameters in a pair to avoid mixing up positions when you call a function.

```
>>> intro_self(your_name = 'ada', your_major = 'programming')
```

- Make sure you use the correct parameter names when you pass arguments this way.
- A good docstring is even more crucial for keyword arguments.

Block 1.6 Functions

- Sometimes you might want to create default parameters to reduce the number of passing arguments and to show what your function should be typically used for.

```
>>> def intro_self(your_name, your_major = 'economics'):  
    """ Displaying name and study major"""  
    print(f"\nMy name is {your_name}.")  
    print(f"My major is in {your_major}.")  
>>> intro_self('ada')
```

- You can always replace the default arguments with a different value of your choice.

```
>>> intro_self(your_name='ada', your_major='programming')
```

- Non-default parameters must be put before any default parameters when creating a function.

Block 1.6 Functions

- Arguments can also be made optional by creating parameters with default value as “None” and combining it with if-statement, if instead of None, but another value is passed, it becomes a default argument:

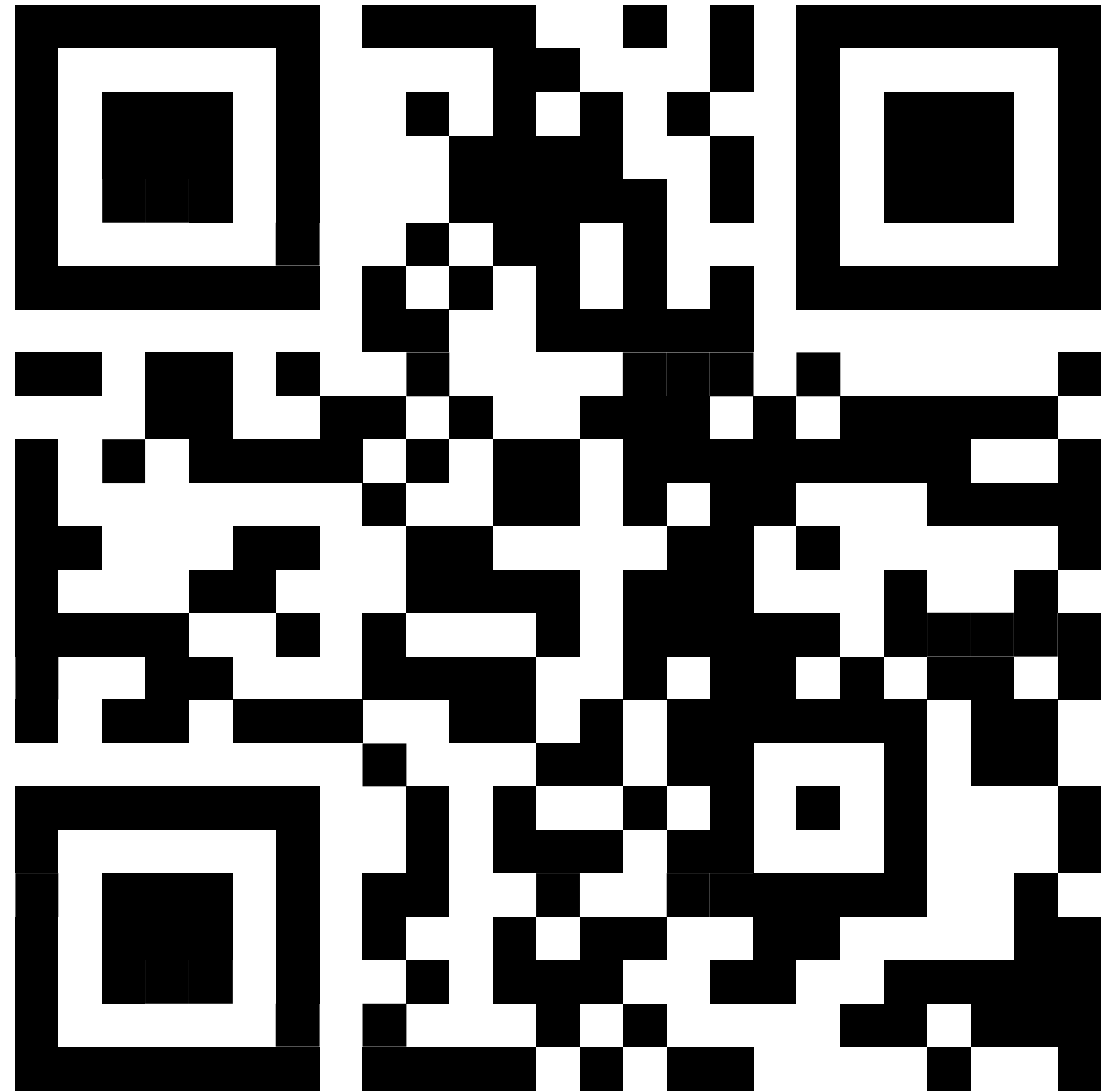
```
>>> def intro_self(your_name, your_major = None):  
#If “programming” is passed instead of None, default major will be programming.
```

```
    """ Displaying name and study major"""  
    if your_major:  
        print(f"\nMy name is {your_name}.")  
        print(f"My major is in {your_major}.")  
    else:  
        print(f"\nMy name is {your_name}.")
```

```
>>> intro_self('ada')
```

Quiz 1

- <https://bit.ly/36JyOgv>



Block 1.6 Functions

- So far, we have looked at functions without return values. With a defined return statement, your function can send back values to the line that called the function.

```
>>> def sumof(num1, num2):  
    """ Return a sum of two given numbers"""  
    sum = num1 + num2  
    return sum  
  
>>> print(sumof(1,1)) # or assign new value then print
```

- Functions can return any type of data you program it to, for example:

```
>>> def create_profile(username, role):  
    profile = {'username': username.lower(), 'role': role}  
    return profile #return value is a dictionary  
  
>>> create_profile('michael', 'member')
```

- When you return more than one variable, these will be put in a tuple follows the order that you define in the **return** statement:

```
def two_variables(a, b):  
    return a, b
```

Block 1.6 Functions

- Here's another example of function when it's used in a while-loop:

```
def sumof(num1, num2):  
    """ Return a sum of two given numbers"""  
    sum = num1 + num2  
    return sum  
  
while True:  
    input1 = input("Give me your first number: ")  
    print("(To quit, enter 'q')")  
    if input1 == 'q':  
        break  
    input2 = input("Give me your second number: ")  
    print("(To quit, enter 'q')")  
    if input2 == 'q':  
        break  
    input1 = int(input1)  
    input2 = int(input2)  
    sum = sumof(input1, input2)  
    print(f"The sum is {sum}")
```

Block 1.6 Functions

- Here's another example of function with list:

```
unregistered = ['adam', 'ada', 'alan']  
registered = []
```

```
def registration(unregistered_list, registered_list):  
    """  
    Print out each element in the unregistered_list  
    Move each element to the registered_list  
    """  
    while unregistered_list:  
        username = unregistered.pop()  
        print(f'Now registering: {username.title()}')  
        registered_list.append(username.title())
```

Block 1.6 Functions

```
def print_registered(list_names):  
    """ Print out all elements in the list """  
    print("\nThe following names have been registered:")  
    for name in list_names:  
        print(name)  
  
registration(unregistered, registered)  
print_registered(registered)
```

- While it's possible to make one single function that process elements in a list (unregistered) and print the processed list (registered). It is a better practice to have different functions performing different tasks. You can also call another function while creating a new function.

Block 1.6 Functions

- Sometimes, you don't know exactly how many arguments a user might pass in a function. Arbitrary notation can be used to collect unlimited arguments
- There are 2 types of arbitrary notation for arguments: (i) arbitrary positional arguments (*args) and (ii) arbitrary keywords argument (*kwargs)
- Here's an example for the first (i):

```
def sumof(*number):  
    result = 0  
    for i in number:  
        result = result + i  
    return result  
print(sumof(2,2,2,2))
```

- The asterisk “*” is used to signal Python that you want to create an arbitrary positional argument.
- After collecting all arguments the user has passed in the function, Python creates a tuple containing all these arguments.

Block 1.6 Functions

- Here's an example for the latter (ii):

```
def concat(**words):  
    result = ''  
    for i in words:  
        result += i  
    return result  
print(concat(a = "hello", b='world')) #as many key-value pairs as you like
```

- The asterisk “**” is used to signal Python that you want to create an arbitrary keyword argument.
- After collecting all arguments (key-value pairs) the user has passed in the function. Python creates a dictionary containing all these arguments.

Quiz 2

- <https://qrgo.page.link/Eo2jN>



Block 1.6 Functions

The order of your parameters when you define a function must be:

1st parameters: Standard arguments (non-default args before default args)

2nd parameter: Arbitrary positional arguments

3rd parameter: Arbitrary keyword arguments

Block 1.6 Recursion

```
>>> def understand_recursion():  
    understand_recursion() #In order to understand recursion you must  
    understand recursion
```

- Recursion is the process of repeating items in a self-similar way.
- Its core implementation idea is to break a problem down to smaller problems. Once you reach the smallest problem where the answer is simple (or given), only then the program will work its way up.
- Any recursion problem can be written iteratively and vice versa. However, sometimes recursion offered a much more elegant and readable solution.

Block 1.6 Recursion

- Here's an example of multiplication done iteratively:

```
def multiplyiter(a,b):  
    i = b  
    result = 0  
    while i > 0:  
        result += a  
        i -= 1  
    return result
```

Block 1.6 Recursion

- Here's an example of multiplication done recursively:

```
def multiplyrecur(a,b):  
    if b == 1:  
        return a  
    else:  
        return a + multiplyrecur(a, b-1)  
  
print(multiplyrecur(2, 3))  
print(multiplyiter(2, 3))
```

Block 1.6 Modules

- Often you want to maintain a clear high-logic (abstraction) level for your program. Putting all functions in one file may take away the readability of your program. Thus, we take them out of the main program and put them into different files called modules.
- Modules are separate files containing reusable functions.
- To reused functions from different files, simply `import` it back to your program.
- A library is a collection of many useful functions or modules.

Block 1.6 Modules

- We created a function in *bookshelf.py* file:

```
def make_bookshelf(num_books, *majors):  
    """  
    Print out a statement about a bookshelf  
    @param num_books: number of books in this bookshelf  
    @param major: what kind of books are in this bookshelf  
    @return: none  
    """  
  
    print(f"Bookshelf contains {num_books}"  
          f"\nBooks in this shelf is about:")  
    for major in majors:  
        print(f"- {major}")
```

Block 1.6 Modules

- In our main.py file, we can:

```
import bookshelf
```

```
bookshelf.make_bookshelf(10, 'economics')
```

⇒ Python looks for the bookshelf.py file in the folder containing our main.py file. Python then copies all the functions in this bookshelf.py file into main.py file.

⇒ To call a function from a module, you must follow this format:

```
module_name.function_name()
```

- To import one specific function, use:

```
from module_name import function_name
```

⇒ Functions imported this way do not required module_name when called.

- Module_name or function_name can be lengthy, sometimes you want to make an alias. E.g:

```
from bookshelf import make_bookshelf as mb
```

```
mb(10, 'economics')
```

- Similarly, you can import the entire module with alias:

```
import bookshelf as bs
```

```
bs.make_bookshelf(10, 'economics')
```

Block 1.6 Modules

- Importing all functions from a module:

```
from module_name import *
```

⇒ No need to use `module_name.function_name()` notation, you can call a function without the `module_name`.

⇒ This approach should NOT be used! (clashes between function you have written with function that have a similar name in the module)

Block 1.6 Styling functions, modules

- Use a descriptive name, similar to variable names: lowercase, with underscores for functions and modules. You may use mixedCase, however, try to be consistent.
- Docstring is create using triple quotation marks. In an ideal case, docstring should explain what are the parameters (type, what is it for) and what the function will return. This ideal format can be automatically created in Pycharm with Python Integrated Tools -> Epytext.
- Write `import` at the beginning of the file. The only exception is the overall description of your program (`#` written as comments)
- Sometimes, a function can be lengthy, you can use ENTER to jump to another line:

```
def function_name(  
    parameter0, parameter1, parameter2, parameter4,  
    parameter5, parameter6, parameter7):  
    .....
```

Block 1.7 Object-Oriented Programming (OO)

- Object-oriented programming is the most effective programming approach for game, software, library or packages development. It is not being used often used by economists whose focus is on data analysis.
- Object-oriented programming does not entirely mean faster, easier coding since it often focuses on development, expandability, and readability. For economists, most of the time functional programming is a more direct (better) approach to solve a problem.
- However, understanding object-oriented programming is highly beneficial to understand the library you are using for data analysis. This programming approach is also used for game development, which is relevant for experimental economists.
- Everything in Python is an object, datatypes that you have learned previously are all objects.

Block 1.7 Classes

- A Class can be thought of as a model of an object (blueprint).

```
class Cat:
```

```
    """Model of a cat"""
```

```
    def __init__(self, name, color, age):
```

#1

```
        """ Initialize name, color, age attributes"""
```

```
        self.name = name
```

```
        self.color = color
```

```
        self.age = age
```

```
    def meow(self):
```

#2

```
        """ Simulate cat meow """
```

```
        print('Meow..')
```

```
    def knock_item(self, item):
```

#3

```
        """ Simulate cat knocks item off the table """
```

```
        print(f"Your cat {self.name.title()} just knocks the {item} off your table")
```

```
cat1 = Cat('Oscar', 'orange', 5)
```

Block 1.7 Classes

- Functions that are placed within a class definition are called methods.
- The first definition (at **#1**) `def __init__()` is a special method. Anything within this method will be run automatically whenever we create an instance of the class.
- An instance is a realization of a class (e.g class is a blueprint of a house, and an actual house built using this blueprint is an instance of that blueprint).
- Other special methods are defined here:
<https://docs.python.org/3/reference/datamodel.html#basic-customization>
- All special methods start and end with double underscores to prevent you from naming a conflicting name.
- `self` is a reference to the instance itself, **this is required in all methods definition**. `self` is passed automatically in all methods call.
- When `self.` is attached before the variable (e.g `self.age`), this variable can be accessed by any other methods in the class. This will also enable use to access the variable through the instance (e.g `cat1.age`, however, accessing attribute this way is not recommended)
- Variables with `self.` attached before it are called attributes.

Introductions to applications in python for economists

Nhat Luong, Kassel University
Lecture 5 (03/12/2020)

Block 1.7 Object-Oriented Programming (OO)

- Object-oriented programming is the most effective programming "approach" (philosophy) for game, software, library or packages development. It is not being used often used by economists whose focus is on data analysis.
- Object-oriented programming does not entirely mean faster, easier coding since it often focuses on development, maintainability, expandability, and readability. For economists, most of the time functional programming is a more direct (better) approach to solve a problem.
- However, understanding the concept of OO programming is highly beneficial to understand the libraries you are using for data analysis. This programming approach is also used for game development, which is relevant for experimental economists.
- Everything in Python is an object, datatypes that you have learned previously are all objects.

Block 1.7 Classes

- A Class can be thought of as a model of an object (blueprint).

```
class Cat:
```

```
    """Model of a cat"""
```

```
    def __init__(self, name, color, age):
```

#1

```
        """ Initialize name, color, age attributes """
```

```
        self.name = name
```

```
        self.color = color
```

```
        self.age = age
```

```
    def meow(self):
```

#2

```
        """ Simulate cat meow """
```

```
        print('Meow..')
```

```
    def knock_item(self, item):
```

#3

```
        """ Simulate cat knocks item off the table """
```

```
        print(f"Your cat {self.name.title()} just knocks the {item} off your  
table")
```

```
cat1 = Cat('Oscar', 'orange', 5)
```

Block 1.7 Classes

- Functions that are placed within a class definition are called methods.
- The first definition (at **#1**) `def __init__()` is a special method. Anything within this method will be run automatically whenever we create an instance of the class. (this is often referred as a “constructor” in Python and many other programming languages)
- An instance is a realization of a class (e.g class is a blueprint of a house, and an actual house built using this blueprint is an instance of that blueprint).
- Other special methods are defined here:
<https://docs.python.org/3/reference/datamodel.html#basic-customization>
- All special methods start and end with double underscores to prevent you from naming a conflicting name.
- `self` is a reference to the instance itself, **this is required in all methods definition**. However, `self` is passed automatically in all methods call.
- When `self.` is attached before the variable (e.g `self.age`), this variable can be accessed by any other methods in the class. This will also enable use to access the variable through the instance (e.g `cat1.age`, however, accessing attribute this way is not recommended)
- Variables with `self.` attached before it are called attributes.

Block 1.7 Classes

- The method at #2 and #3 (slide 3) is used to just print out a statement (e.g “Meow..”).
- To make an instance from a class, variable name must be given, then arguments defined in the `__init__` method must be passed:

```
cat1 = Cat('Oscar', 'orange', 5)
```

- To access an attribute from the instance of a class, you can use dot notation.

```
print(cat1.age)
```

⇒ However, this is not at all recommended, just don't do it!

- To access an attribute from the instance of a class, you should write a method to do so, we add this method in the class defined on slide 3 as follow:

```
def get_age(self):  
    return self.age
```

⇒ From now on, when you want to get the `age` of your cat, use a method call: `cat1.get_age()`

- Python allows you to reassign attribute outside of a class definition by directly calling it and assign a new value:

```
cat1.age = 10
```

⇒ Again, this is not at all recommend, to reassign a new value to an attribute, you should write a method specifically used for this purpose:

```
def set_age(self, new_age):  
    self.age = new_age
```

Then: `cat1.set_age(10)` or `cat1.set_age(new_age = 10)`

Block 1.7 Classes

- Methods use for getting an attribute or setting a new value to an attribute are called setters and getters. These are used outside of class definition to access/modify your class attributes.
- Here are a few things you should NOT do with your class, even though Python allows you to do so. Just don't do it:
 - access data from outside class definition: `print(cat1.age)`
 - write to data from outside class definition: `cat1.age = 'unknown'`
 - create data attributes for an instance from outside class definition: `cat1.size = "tiny"`
- You can create infinite numbers of instances from a class.
- Instead of setting a new value you can also increment new value to an attribute

```
def add_to_age(self, increment_age):  
    self.age += increment_age
```

⇒ The possibility is endless. You can create any methods using the tools and principles that we have learned.

Quiz 1

Will be given in class

Block 1.7 Concept of OO Programming:

Inheritance

```
class Feline:
    """A template for feline animals
    @name: string
    @age: int
    @female: 1 if true, 0 otherwise"""

    def __init__(self, name, age, female):
        self.name = name
        self.age = age
        self.female = female

    def speak(self):
        print("ROAARR!")

class Cat(Feline): # 1
    def __init__(self, name, age, female):
        super().__init__(name, age, female) # 2

cat1 = Cat('Nora', 3, 1)
```

- When a class inherits from another, we called the inherited class a child class and the original the parent class (sometimes called superclass (parent) and subclass (child)).
- A child class can have some or all of the attributes and methods of the parent class. A child class can also freely redefine methods or attributes inherited from the parent class.

Block 1.7 Concept of OO Programming: Inheritance

- Parent class must appear before the child class in a file (unlike functions)
- To signal inheritance, you call a class with another class name in parenthesis **#1 (slide 8)**
- The `super()` function is a special built-in function that helps you to call a method from the parent class. Usually, you won't need this.
- At **#2 (slide 8)** `super()` tells Python to get the `init` method from the parent class (which is `Feline`). This gives `Cat` all the attributes that were defined in `Feline` `init` method. Your code at #2 will still work without the `def __init__(self, name, age, female):` (the constructor)
- After successfully inherited from the parent class, you can freely add new or modify attributes or methods of the child class. This won't have any effect on the parent class. Here are examples, using the code in slide 8:

Block 1.7 Concept of OO Programming: Inheritance

```
class Cat(Feline):  
  
    def __init__(self, name, age, female):  
        super().__init__(name, age, female)  
        self.domesticated = True #3  
  
    def speak(self): # 1  
        print("MEOW!")  
  
    def sit(self): # 2  
        print("Your cat is now sitting")
```

- ⇒ At #1, we redefine the `.speak()` method for our class Cat. Instead of "ROAARR!", instances created from Cat will print out "MEOW!" when you call the method `.speak()`. This is called polymorphism.
- ⇒ At #2, we create a new method that applies for only Cat class, if you call this method for an instance of Feline, an error will occur.
- ⇒ At #3, we add another attribute to our Cat class, again this attribute only exists for this child class (i.e. Cat).

Quiz 2

Will be given in class

Block 1.7 Import classes from modules

- Similar to functions, you can separate your class definition codes into another file and import it back to the main file.
- For example the class definition for Feline on page 29, can be put in a file name feline.py . Then in the main file (main.py) you can:

```
from feline import Feline
```

- Note that this import only Feline class. Other classes defined in this feline.py won't be imported.
- To import multiple classes, simply use comma to separate them

```
from feline import Feline, Cat
```

- Or similar to importing functions, you can import everything in a module:

```
import feline
```

- However, this means the file name must be called every time you use codes in this file (e.g. feline.Feline(...))
- Again, very similar to functions, you can import all class without calling filename by:

```
from feline import *
```

⇒ This is not recommended for the same reasons as in functions importing

- You can also use aliases, similar to functions:

```
from feline import Feline as Fel
```

Block 1.7 Python standard library

- Python standard library includes many useful modules that are included when you install Python. (It's ready there, no need to download anything)
- The most frequently used module from Python standard library is the random module. Here're a few examples:

```
import random
```

```
print(random.randint(1,6)) #randomly select a number from 1 to 6  
(including 6)
```

```
ticket = [1, 3, 5, 6]
```

```
print(random.choice(ticket)) #randomly choose an element from  
collectibles such as list or tuple
```

Block 1.7 Styling for Classes

- Use CamelCase for class name
- Docstring should be formatted the same way as functions (what the class is trying to do, what are the parameters, what arguments should be passed to parameters, .etc)
- There should be a blank line between methods defined in a class and 2 blank lines between 2 different classes in a file.
- Important: import statement for modules from the standard library should come before modules that you wrote. These two types of modules should be separated with a blank line. This helps other programmers distinguish where the modules come from. For example:

```
import random # Standard library module
```

```
import feline # Module that you wrote yourself
```

Block 1: Contents

- What is Python ? Installation and IDE
- Variable and Common Data Types
- List and Working with List
- If Statement
- Dictionary
- Input, While-loop, Function
- Classes
- Handling Exceptions and Files
- Code Testing



Block 1.8 Handling exceptions

- Whenever there is an error occurred, Python creates an exception object.
- Program can continue if these exceptions are properly managed.
Otherwise, a traceback will appear on the screen/console and the program stops.
- You need to know the name of the Error you are expecting in order to handle it if said Error occurs.
- For example, when you divide a number by zero, this error will show up on your console:

```
Traceback (most recent call last):
```

```
File "<input>", line 1, in <module>
```

```
ZeroDivisionError: division by zero
```

- In this example, the name of this error (the exception object) is
"ZeroDivisionError"

Block 1.8 Handling exceptions

- Exceptions are handled using try-except block, for example:

try:

```
    print(10/0)
```

except ZeroDivisionError:

```
    print("Cannot divide by zero")
```

- This syntax means, whenever a ZeroDivisionError occur in the try block, print “Cannot divide by zero” on the screen. The rest of your program will continue to execute without stopping.
- Here’s another example:

try:

```
    print(10/0)
```

except ZeroDivisionError:

```
    pass
```

- You don’t always have to print out an error message to the user, sometimes it’s best to let the program “failing silently” as if nothing wrong had happened.

Block 1.8 Handling exceptions

- Remember in the input lecture, we assume that our user will type exactly a number in input. However, if they don't, a ValueError will be printed out on the screen/console. One way to handle this is using the try-except block:

```
prompt1 = "Give me your first number"
prompt2 = "Give me your second number"

try:
    num1 = int(input(prompt1))
    num2 = int(input(prompt2))
except ValueError:
    print("You did not enter a number!")
else:
    sum = num1 + num2
    print(sum)
```

- In this example, whenever the codes in try block can be executed without the ValueError, Python will execute the else block.

Block 1.8 Handling exceptions

- It is not always a good idea to use try-except block. If you find too much exception blocks in your program, this means your codes are not yet well-designed and can be fixed.
- Depending on situations, try-except block can consume much more processing power, which slows down your program significantly.
- For example, if you only want to check if user-input is an integer, you can:

```
input_number = input('Give me an integer')
if input_number.isdigit():
    print(f"The number you just enter is {input_number}")
else:
    print("You did not enter a number")
```

- Built-in methods like `.isdigit()` often used when you're expecting a lot of incorrect inputs (asking for number but a lot of users input string). If you were expecting a lot of correct inputs (most users will give you a number when you ask for it), try-exception block is a much better choice.
- Try-Except block often used when you are expecting external inputs (user input, files, network connections). For internal uses, try-except blocks should be avoided. This also means most of economic uses (e.g. analyzing data) will not require try-except blocks.

Quiz 3

Will be given in class

Block 1.8 Files

- Assume you have a `my_text.txt` file in the folder your `main.py` file is located, to import the content from this file to Python, we follow this syntax:

In `main.py`

```
with open('my_text.txt') as mytext:  
    contents = mytext.read()
```

```
print(contents)
```

- `open()` require at least 1 argument, that is the name of the file you want to open
- `my_text.txt` file is an object and we name it as `mytext` in Python using the syntax `with...as`. The `with` block also automatically close the file once we open it with `open()`.
- You can also `open()` the file then `close()` it manually. However, this is not recommended.
- We use `.read()` method to collect all contents of `mytext` as a long string in `contents`.
- `.read()` will create a blank line at the end of the file automatically. This blank line can be remove using `.rstrip()`. E.g. `contents.rstrip()`

Block 1.8 Files

- If you want to organize your files in different folder, when `open()` the file, just give the directory using forward slashes (/). For example:

```
with open('C:/dialogue/my_text.txt') as mytext:  
    contents = mytext.read()
```

- Backslashes (\) will result in an error because it is reserved for escape characters. If you insist on using backslashes, double backslashes can be used instead. (`'C:\\dialogue\\my_text.txt'`)
- The above examples use absolute path. You can also use 'relative path' to access a file when that file is in a subfolder of your current program file. For example, `'dialogue/my_text.txt'`
- If you want a more complex directory (path) handling, try the `pathlib` or `os` library. However, it is not recommended to have a too complex path system. It is most efficient to have a `main.py` file in a parent folder, and subfolders in this parent folder if you want to organize your files.
- If your file path is too long, simply assign it to a variable, for example:

```
file_path = 'C:/dialogue/my_text.txt'
```

- By convention, the name of the file you want to open should also be stored in a variable (often simply called `filename`). This prevents mistake thinking the file is an actual file object instead of a name.

Block 1.8 Files

- Sometimes, you want to read a text file line by line. This can be done with for loop:

```
filename = 'test_text.txt'
with open(filename) as mytext:
    for line in mytext:
        print(line.rstrip()) #without .rstrip() each line will be separated by a blank line
```

- You can also output lines as a list, then work with this list outside of the with block:

```
filename = 'test_text.txt'
with open(filename) as mytext:
    lines = mytext.readlines()
```

```
for line in lines:
    print(line.rstrip())
```

- Once you have extract the content of the file into a string or list of string, you can perform any operations that we have learned with it.

Block 1.8 Files

- To open a file, you simply need to pass on an extra argument in the `open()` function.

```
filename = 'hello.txt'
```

```
with open(filename, 'w') as file_object:
```

```
    file_object.write("Hello world!")
```

- the argument 'w' stands for write. Python will automatically create the file that you want to write in if it does not exist in your current folder. It will also replace the file that has the same name. Therefore, use it with caution.
- Python writes only string type to a text file, therefore, if you want to write numbers into a text file, use `str()` function.
- Other modes of the `open()` function are: 'r' for read-only (this is the default), 'w' for write, 'a' for append and several others such as 'r+' for both read and write.
- The formatting of the string type when Python writes on a file is similar to `print()` function. This means, if you want a tab or a new line, you must use escape character.

Block 1.8 Files

- Here's an example of the append mode, with multiple `.write()` methods, you can see that the lines are squished together (without breaking to a new line).

```
filename = 'hello.txt'  
with open(filename, 'a') as file_object:  
    file_object.write("Hello World!")  
    file_object.write("Hola Mundo!")
```

- When writing using append mode ('a'), the file won't be erased (compare to 'w' mode). These strings are simply added at the end of the last character in the previous file.

Block 1.8 Files

- Aside from .txt files, Python often use .json file as data storage. To use json file, you need to import json module.
- JSON: JavaScript Object Notation. Json is used not only for Python but many other programming languages, it's easy to use and learn.
- To save data as json:

```
import json
```

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

```
filename = 'data.json'
```

```
with open(filename, 'w') as f:
```

```
    json.dump(mydata, f)
```

- json.dump() function takes in 2 arguments, the first argument is the data you want to export and the second argument is the file object.
- If you now open the data.json file you will see that it looks just like Python list

Block 1.8 Files

- To load data back into Python:

```
import json
filename = 'data.json'
with open(filename) as f:
    mydata = json.load(f)
```

- `json.load()` takes in 1 argument, the file object. This function collects data from the object files, we then assign this data to variable `mydata`.
- Json file is useful to store data from users (whether from a game, or a website or from different programs) – it's widely used and very flexible. Here's an example to share data between 2 different programs:

In `prompt.py`

```
import json

guest_username = input("Give me your name")

filename = "guests.json"
with open(filename, 'w') as f:
    json.dump(guest_username, f)
    print("Your username has been recorded")
```

Block 1.8 Files

In greetings.py

```
import json
filename = 'guests.json'

with open(filename) as f:
    guest_name = json.load(f)
print(f"Hello {guest_name}! Welcome back!")
```

- **Now if we combine 2 of these programs into one file (in the prompt.py file):**

```
import json
filename = "guests.json"
try:
    with open(filename) as f:
        name = json.load(f)
except FileNotFoundError:
    new_name = input("Give me your name: ")
    with open(filename, 'w') as f:
        json.dump(new_name, f)
    print(f"First time here {new_name}? Welcome")
else:
    print(f"Hello {name}! Welcome back!")
```

This program simply asks for your name and creates a file with your name in it. If you run this code a second time, it should know your name and give you a greeting!

Block 1.8 Refactoring

- Refactoring is the process of breaking down your codes into smaller reusable functions that perform specific tasks. While the codes in the previous slide work, it's much cleaner, easier to understand and expand if you break it down into different functions. Here is one way:

```
import json

def get_username():
    """
    Check if username is available in file
    :return: username
    """
    filename = 'guests.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        return None
    else:
        return username
```

Block 1.8 Refactoring

```
def greeting():
    """
    Print out a greeting
    :return:
    """
    username = get_username()
    if username:
        print(f"Hello {username}! Welcome back!")
    else:
        username = input("Give me your name: ")
        filename = 'guests.json'
        with open(filename, 'w') as f:
            json.dump(username, f)
            print("Your username has been recorded")

greeting()
```

Block 1.8 Refactoring

- Although in the 2 previous slides, we have refactored our program into 2 separate functions that serve a clear purpose. However, function greeting() could still be broken down:

```
def set_new_username():  
    """  
    Prompt new username  
    :return:  
    """  
    newname = input("Give me your name: ")  
    filename = 'guests.json'  
    with open(filename, 'w') as f:  
        json.dump(newname, f)  
    return newname
```

Block 1.8 Refactoring

```
def greeting():  
    """  
    Print out a greeting to user  
    :return:  
    """  
    username = get_username()  
    if username:  
        print(f"Hello {username}! Welcome back!")  
    else:  
        username = set_new_username()  
        print("Your username has been recorded")
```

`greeting()`

- Our final program now comprises of 3 functions: `get_username()` in slide 26, `set_new_username()` in page 28 and the above code. We also see that the program is somewhat lengthy comparing to codes in slide 25, although they perform the same task.
- This is somewhat similar to object-oriented programming. Refactoring or object-oriented programming does not mean easier coding, it simply makes it easier to expand and debug. Economists perhaps enjoy a more direct approach similar to codes in slide 25.

END NOTE

- We have covered a lot for basic programming in Python, however, this is still only a very small fraction. You will always have to keep searching online using your favorite search engine, reading and trying new things. Technology will keep changing and this process is unavoidable. The true message to take from this block is the concept and principles of programming.
- There are a lot of useful commands, functions, methods which can be found here <https://docs.python.org/3/library/>
- These might be hard to read and understand, however, it is the best (and official) documentation there is. You can always try other sources, however, take it with a grain of salt!