

# O Object O Oriented P Programming

S1 = Python with Bianca

S2 = Java with Bryan

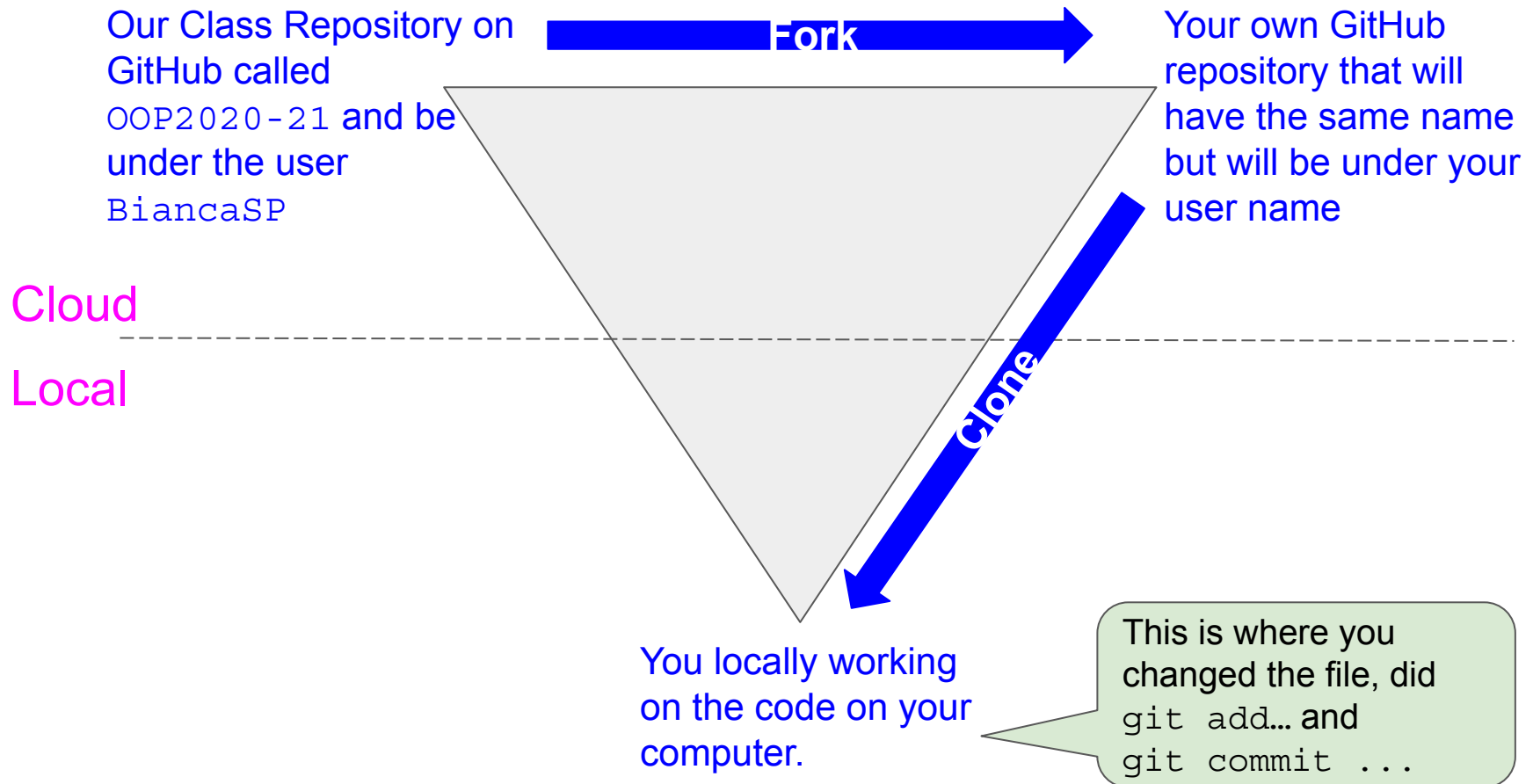
DT228(TU856)/DT282(TU858) - 2

# Revision Lab 2 and Python Foundations

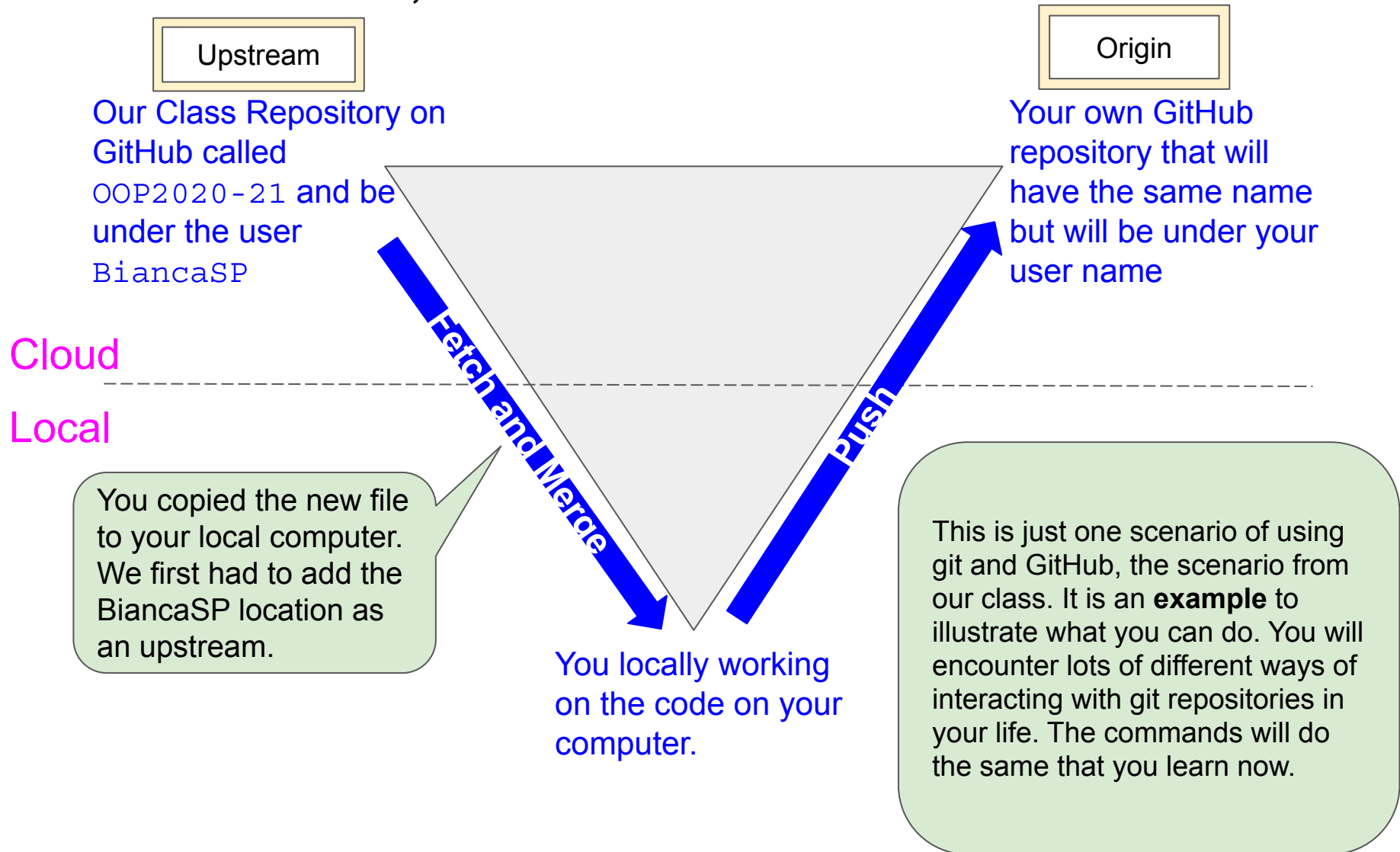
# Objectives

- Revise Lab 2, ask Questions
  - Git and
  - Python
- Rest of Python Foundations

# Git and GitHub, Lab 1

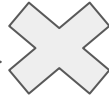


# Git and GitHub, Lab 2



# Git and GitHub, Tutorial

In the tutorial you only had one GitHub repo, your own



Your own GitHub repository

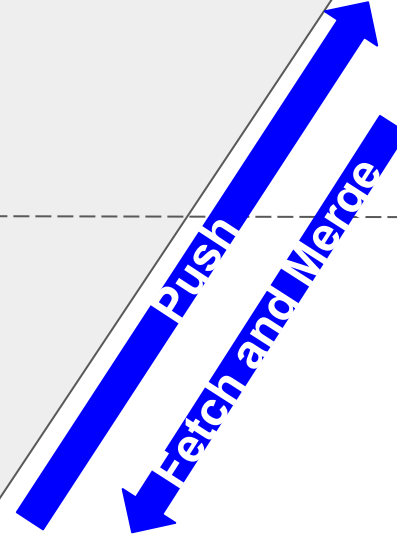
Cloud

Local

In the tutorial we created a folder on the computer, then declared it a git repo by using `git init`, then pushed it up to GitHub, then worked locally with the file and kept pushing up the changes to the online repo, or we made an online change and pulled that change down to our local folder.

You locally working on the code on your computer.

In the tutorial we also worked with different branches apart from master



# Data Types: List

- Zero-indexed
- Guarantees order of items

**List:** collection of data which are normally related, can be different data types

```
list_name = [initial values]
```

```
user_age = [10, 20, 30, 40, 50, 60, 70], or
```

```
user_age = []
```

This is an empty list. Add items using append() method

- Individual items can be accessed by an index that starts with 0 (never with 1)
  - Last item in the list has the index -1, second last -2, etc
  - user\_age[-1] will return 70
- Can be assigned to a variable:

```
user_age2 = user_age
```

```
user_age3 = user_age[1:3]
```

```
user_age4 = user_age[1:6:2]
```

So called **slice** notation:  
Item at the start is always **included**, item at the end is always **excluded**. Third number is a stepper (every 2<sup>nd</sup> item) in range.

# Slices

```
user_age = [10, 20, 30, 40, 50, 60, 70]
```

- Useful defaults:
  - First number is 0, example: `user_age [ :4]` returns values from 0 to 4-1
  - `user_age[1: ]` returns values from index 1 to length-1



# Working with Lists

- Modify by assigning new value to an index

```
user_age[3] = 99 results in
```

```
user_age = [10, 20, 30, 99, 50, 60, 70]
```

- Add an item with append()
- `user_age.append(100)` results in

```
user_age = [10, 20, 30, 99, 50, 60, 70, 100]
```

- Remove an item with del

- `Del user_age[3]` results in

```
user_age = [10, 20, 30, 50, 60, 70, 100]
```

# Working with Lists

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',  
'i', 'j']
```

```
1.del my_list[2]
```

```
print(my_list)
```

```
2.del my_list[1:5]
```

```
print(my_list)
```

```
3.del my_list[:3]
```

```
print(my_list)
```

```
4.del my_list[2:]
```

```
print(my_list)
```

# Extending lists

- `extend()`
  - Combine two lists
- What does the following do:

```
my_list1 = ['a', 'b', 'c']
```

```
my_list2 = [1, 2, 3]
```

```
my_list1.extend(my_list2)
```

```
print(my_list1)
```

# Working with lists

- `in`



No brackets

- True if the parameter is in the list, false otherwise
- `print('c' in my_list1)` returns true

- `insert()`

- Add an item at a particular position

- `len()`

- Number of items

# Working with Lists cont'd

- `pop()`
  - Get an item at an index position and remove it from the list
- Example:

```
my_list1 = ['a', 'b', 'c']  
member = my_list1.pop(2)  
print(member)  
print(my_list1)
```

Removes the last item on the list if no argument is given.

# Working with lists

- `remove()`
  - Removes an item from a list and requires a specific value
- Example:

```
my_list1 = ['a', 'b', 'c']  
my_list1.remove('a')  
print(my_list1)
```

Removes the last item on the list if no argument is given.

# So what's the difference?

- What is the difference between del, pop and remove?
  - **remove**: removes first matching value from the List (not a specific index location)
  - **del**: removes an item at a specific index location
  - **pop**: Removes an item at a specific index location and returns it

# Sorting lists

- `reverse`
- `sort`
- `sorted(...)`
  - Returns a new sorted list without changing the original list



# Operators and Lists

- + operator to concatenate

```
my_list1 = ['a', 'b', 'c']
```

```
my_list2 = [1, 2, 3]
```

```
print(my_list1+my_list2)
```

- \* operator to multiply

```
print(my_list1*3)
```

# Data Types: Tuples

- **Tuples:** like lists, but values cannot be changed

```
tuple_name = (values)
```

```
months_of_the_year = ('Jan', 'Feb', 'Mar')
```

- Individual values can be accessed through indexes, like we did with lists
- Delete a complete tuple with del

```
del months_of_the_year
```

```
print (months_of_the_year)
```

Will produce an error if the variable name is not defined!

- Indexes can be used the same way as with lists

# Working with Tuples

- `del`: deletes the whole tuple
- `in`: “a” in myTuple -> True
- `len()`
- `+` concatenates tuples
- `*` multiplies tuples

The tuple itself does not get modified.

# Data Types: Sets

- Similar and hugely different
- Cannot contain duplicates
- Can contain mixture of types
- Are unordered!
- Example:

```
hello = set('hello')
```

```
print(hello)
```

```
-> {'l', 'h', 'e', 'o'}
```

# Data Types: Dictionary

- **Dictionary:** collection of related data PAIRS

```
dictionary_name = {dictionaryKey : data}
```

```
customers = {"Bianca":30, "Brian":31, "Susan":32}
```

- Dictionary key must be unique, error when you try

```
customers = {"Bianca":30, "Brian":31, "Bianca":21}
```

- Different python versions will behave differently here and ask you to use a constructor instead

- Dictionaries are collections, but they are not sequences
  - There is no order
  - The order might change if the elements in the dictionary change

# Working with Dictionaries

```
print(users_and_age["Bianca"])
```

30

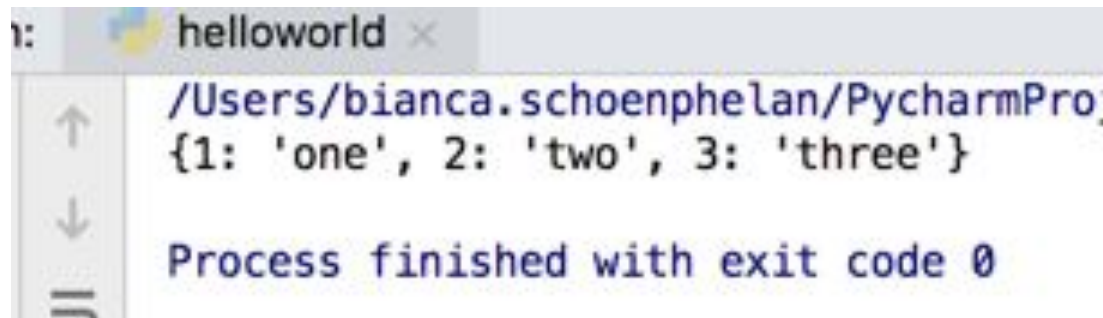
- Modify: `users_and_age["Bianca"] = 51`
- Remove: `del users_and_age["Bianca"]`
- Empty: `users_and_age.clear()`
- `get()`: returns a value for a given key
- `in`: check if an item is in the dictionary
- `items()`: returns a list of dictionary pairs as tuples
- `print(users_and_age.items())`

```
dict_items([('Bianca', 30), ('Brian', 40),  
('Susan', 50), ('Lana', 'Not Available')])
```

# Working with Dictionaries

- `keys()`: returns a list of dictionary keys
- `values()`: returns a list of dictionary values
- `len()`: number of items in dictionary
- `update()`
- Example:

```
dict1 = {1: 'one', 2: 'two'}  
dict2 = {1: 'one', 3: 'three'}  
dict1.update(dict2)  
print(dict1)
```



```
helloworld x  
/Users/bianca.schoenphelan/PycharmPro  
{1: 'one', 2: 'two', 3: 'three'}  
  
Process finished with exit code 0
```

# Type Casting

- When we convert a variable's data type from one data type to another we talk about **type casting**
- Python offers 3 built in functions for type casting:
  - `int()` : takes a float or appropriate string and converts it into an integer
  - `float()` : takes an integer or an appropriate string and converts it into a float
  - `str()` : converts an integer or a float into a string



# Summary

- ★ **Git and GitHub**
- ★ **Lists, Tuples, Sets and Dictionaries**



# References

1. [Kenny Eliason, Difference between OOP and Procedural Programming, https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/](https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/), 2013, Accessed Sep 2020.
2. [The Real Python, 2012-2018, https://realpython.com/switching-to-python/](https://realpython.com/switching-to-python/), Accessed Sep 2020.
3. Learn Python in one day, Jamie Chan, 2014
4. Moutaz Haddara, Introduction to Object-oriented programming, slideshare, 2014.
5. Jamie Chan, Learn Python in one day, 2014.
6. Formatting, [https://www.python-course.eu/python3\\_formatted\\_output.php](https://www.python-course.eu/python3_formatted_output.php), accessed Sep 2020.