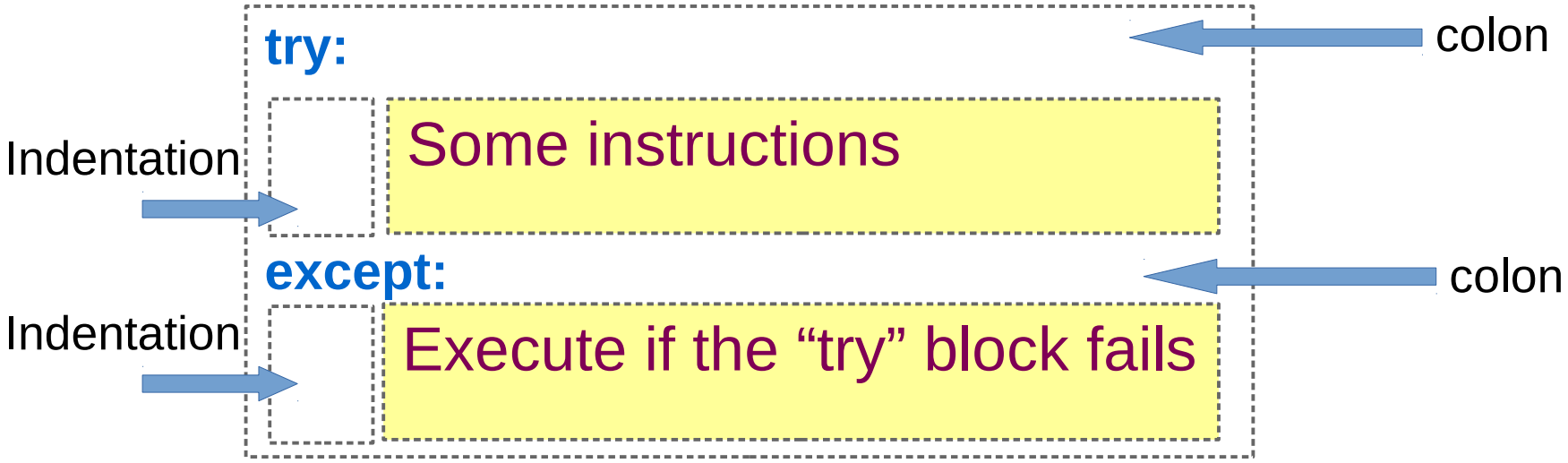
ECE-122
Complement: Exceptions- Discussion about Objects

Catching Exceptions - Definition

- An exception is an error that happens during the execution of a program and that causes the code to crash (bug). For example:
  - division by zero, file open error, wrong type, etc.
- In Python, it is possible to include error handler which can both “catch” and "fix" the problem so the execution of a program can proceed.
- The programmer can then try to anticipate any potential errors using a **try** and **except** instruction block.



Catching Exceptions-Example

```
a,b=map(float,input("Enter a,b to compute a/b: ").split())
print(a/b)
```

```
Enter a,b to compute a/b: 3 2
1.5
```

```
Enter a,b to compute a/b: 3 0
Traceback (most recent call last):
  File "/home/polizzi/122/s19/6.1/test1.py", line 2,
in <module>
    print(a/b)
ZeroDivisionError: float division by zero
```

- Using error handling

```
a,b=map(float,input("Enter a,b to compute a/b: ").split())
try:
   print(a/b)
except:
   print("Your b is 0 and you cannot divide by zero")
```

```
Enter a,b to compute a/b: 3 0
Your b is 0 and you cannot divide by zero
```

Catching Exceptions-Example

- Make it a bit more practical using while and break statement

```
while True:
    try:
        a,b=map(float,input("Enter a,b to compute a/b: ").split())
        print(a/b)
        break
    except:
        print("Your b is 0 and you cannot divide by zero")
```

```
Enter a,b to compute a/b: 3 0
Your b is 0 and you cannot divide by zero
Enter a,b to compute a/b: 4 0
Your b is 0 and you cannot divide by zero
Enter a,b to compute a/b: 3 1
3.0
```

- Now, what happens if we use the wrong type (such as a str that cannot be converted to float)

```
Enter a,b to compute a/b: a 1
Your b is 0 and you cannot divide by zero
...
```

Catching Exceptions-Example

- We can then identify 2 types of errors here: (i) 0 input, (ii)conversion to float
- For example (in python shell)

```
>>>3/0
.....
ZeroDivisionError: division by zero
```

```
>>>float("a")
.....
ValueError: could not convert...
```

- **except** allows to target/recognize/catch the type of errors! so we can take the appropriate action

```
while True:
    try:
        a,b=map(float,input("Enter a,b to compute a/b: ").split())
        print(a/b)
        break
    except ZeroDivisionError:
        print("Your b is 0 and you cannot divide by zero")
    except ValueError:
        print("Enter only number please!")
```

```
Enter a,b to compute a/b: 2 b
Enter only number please!
Enter a,b to compute a/b: 2 0
Your b is 0 and you cannot divide by zero
Enter a,b to compute a/b: 2 1
2.0
```

Catching Exceptions

- Other useful examples for catching error when reading/writing files

```
>>>f=open("myfile.txt","r")
.....
FileNotFoundError: [Errno 2].....
```

```
>>>f=open("/etc/passwd","w")
.....
PermissionError: [Errno 13]
```

not permitted to access

- A large number of built-in exception in Python3
  https://docs.python.org/3/library/exceptions.html
- Catching exceptions allows to:
  - increase your program robustness (prevent crashes)
  - polish your program before releasing a complete/finished software product (however, this will involve a lot of overhead work)
  - be used as a feature (programming style), for example:
    - Try to open a particular file to import some data
    - or, if it fails, generate your own data set

Augmented assignment operators

- In programming, we often need to do either increment or decrement a given value, for example:

```
#suppose a is assigned e.g. a=0
a=a+1 #increment a by 1
a=a+2 #increment a by 2
a=a-3 # decrement a by 3
a=a-0.5 #decrement a by 0.5
```

```
#suppose a is assigned e.g. a=0
a+=1 #increment a by 1
a+=2 #increment a by 2
a-=3 # decrement a by 3
a-=0.5 #decrement a by 0.5
```

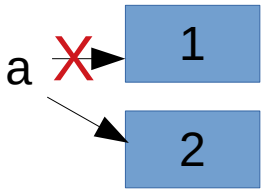
- Like many other languages, Python offers some shortcut notations to perform these increments. Why? Legacy features from older C compiler.
- Unlike Java or C that are using the ++ or -- operators, Python is using the += and -= operators. Why? not sure...
- For each mathematical operator, +, -, /, //, *, **, there is a corresponding augmented assignment operator +=, -=, /=, //=, *=, **=.
- Augmented operators may provide better performance in few cases than using the explicit expression. However, the notation may cause confusions if you switch between languages very often.

Mutable vs. Immutable Objects

- In Python, everything is an object; however, not all objects are created equal
- There are two kinds of objects in Python: Mutable objects and Immutable objects. The value of a mutable object can be modified in place after its creation, while the value of an immutable object cannot be changed.
  - Immutable Object: **int, float, complex, string, tuple, boolean**
  - Mutable Object: **list, dict, set, user-defined classes**
- Let us consider the function **id**() that returns the identity of an object. It returns the memory address of the object which is unique for each object.
  - What is happening for immutable objects?

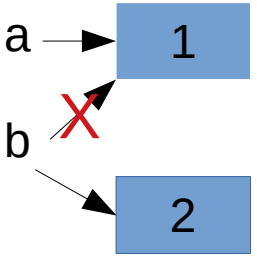
```
a = 1
print(id(a))
a = a+1
print(id(a))
```

10910400
10910432



Rq: 1 and 2 are objects of class **int;** while a is a variable (reference to values/objects). The object 1 will go to a garbage collector and be removed.

```
a = 1
print(id(a))
b = a
print(id(b))
b=2
print(id(b))
```

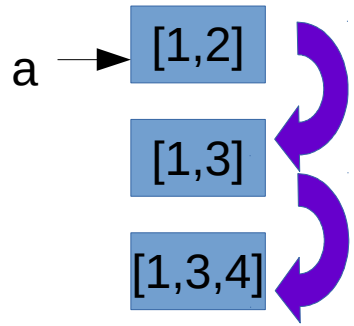


10910400
10910400
10910432

Mutable vs. Immutable Objects

- What is happening for mutable objects?

```
a = [1,2]
print(id(a))
a[1]=3
print(id(a))
a.append(4)
print(id(a))
```




```
22667522436168
22667522436168
22667522436168
```

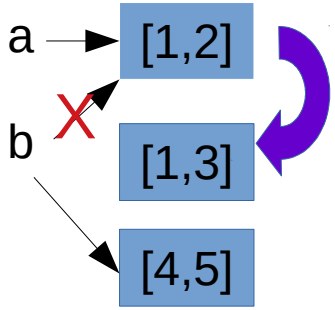
append methods works in-place

```
a = [1,2]
print(id(a))
a.append(3)
print(id(a))
a=a+[4]
print(id(a))
```

Use the __add__ overload operator method that does not work in-place (create a new object)

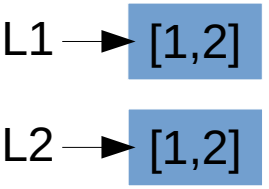
```
22667495999432
22667495999432
22667496205192
```

```
a = [1,2]
print(id(a))
b = a
print(id(b))
b[1]=[3]
print(id(b))
b=[4,5]
print(id(b))
print(id(a))
```




```
22667525956424
22667525956424
22667525956424
22667496551560
22667525956424
```

```
L1 = [1,2]
L2=  [1,2]
print(L1==L2)
print(L1 is L2)
```



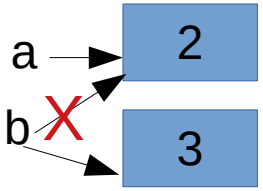

```
True
False
```

Bottom line: Mutable objects just mean that there exists some instance methods in the class that can modify the object in place (no return statement e.g. append for list). The classes (int,string,etc.) do not have such methods. They are then immutable.

Passing to Functions

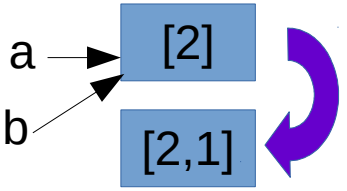
- Passing to functions
  - traditional concepts in C
    - passing by values
    - passing by references (pointers)
  - Python approach
    - using immutable objects: Python can access external variables but not modify them (as soon as it is modified a local variable is created)
    - using mutable objects: Python can modify the object using the instance methods of this object.
    - Also, a function cannot reassign an external variable
- Example: Passing immutable vs mutable objects

```
def func(b):
    b = b+1
a=2
func(a)
print(a)
```




```
# result is 2
```

```
def func(b):
    b.append(1)
a=[2]
func(a)
print(a)
```




```
# result is [2,1]
```

Passing to Functions

- Tricky mistake 1

```
def func1(b):
    b.append(1)

a=[2]
func1(a)
print(a)
```

➡ [2,1]

```
def func2(b):
    b=b+[1]

a=[2]
func2(a)
print(a)
```

➡ [2]

b=b+[1] is equivalent of using the __add__ overload operator method that does not work in-place (create a new object)

```
def func3(b):
    b+=[1]

a=[2]
func3(a)
print(a)
```

➡ [2,1]

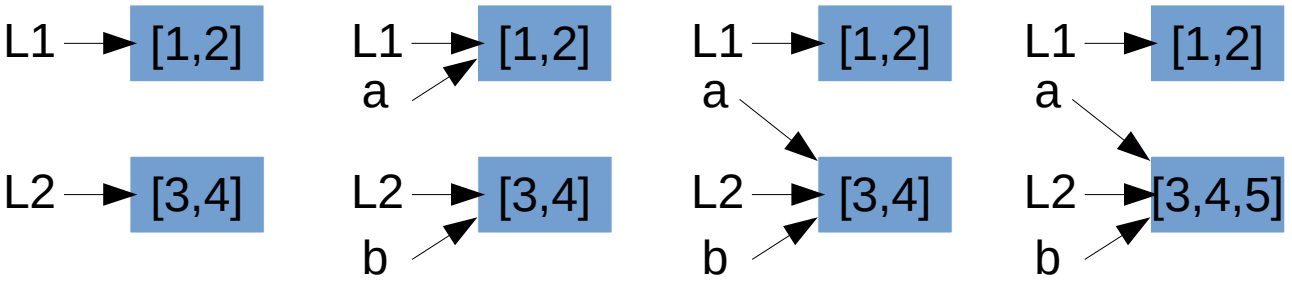
b+=[1] is equivalent of using the __iadd__ overload operator method that works in-place (it is equivalent to the append method)

Passing to Functions

- Tricky mistake 2

```
def func(a,b):
    a=b
    a.append(5) # same a+=[5]

L1=[1,2]
L2=[3,4]
func(L1,L2)
print(L1)
print(L2)
```



➡ [1,2]
[3,4,5]