ECE326

PROGRAMMING LANGUAGES

Lecture 6 : Review and Python Tidbits

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

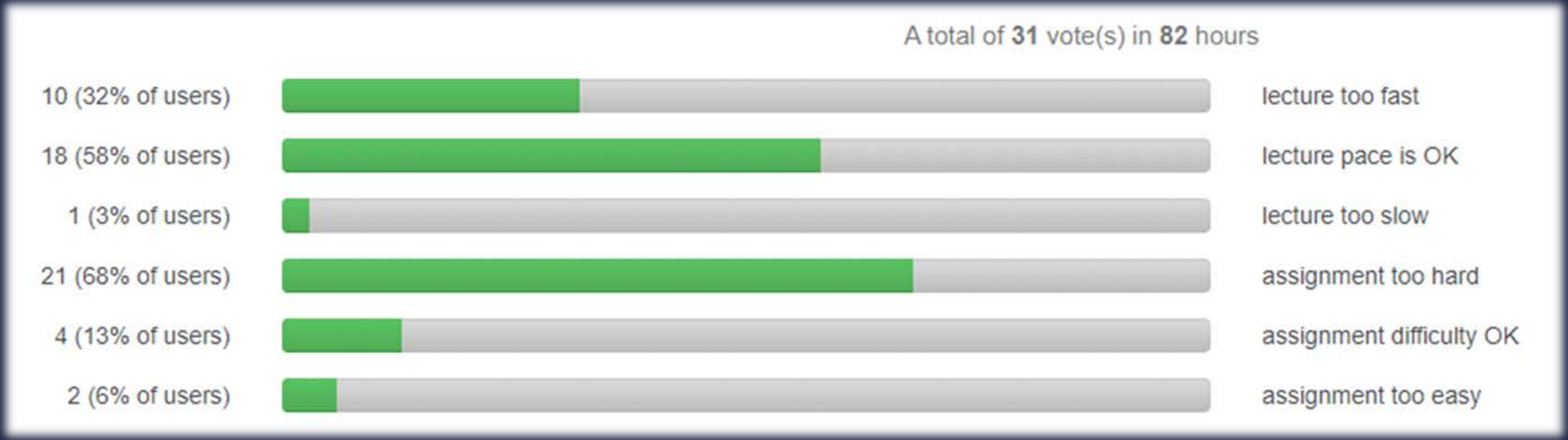
Course Prerequisite

- ECE244: Programming Fundamentals
- ECE297: Design and Communication
- Not a prerequisite
  - ECE302: Probability and Applications
  - ECE345: Algorithms and Data Structures

Assignment 2

- Released next week (postponed by 1 week)
- Will use basic concepts from high school
  - Probability
  - Expected Value
- Dynamic Programming
  - Intent of the course
    - Practical, programmer perspective
    - i.e. save result of function and reuse on same argument
  - Not part of the course
    - Theory: e.g. asymptotic runtime complexity analysis, amortized … zzz

Result of Class Poll



A total of **31** vote(s) in **82** hours

10 (32% of users)	lecture too fast
18 (58% of users)	lecture pace is OK
1 (3% of users)	lecture too slow
21 (68% of users)	assignment too hard
4 (13% of users)	assignment difficulty OK
2 (6% of users)	assignment too easy

Updates to Assignments

	Before			*After*		
Assignment	**Start**	**End**	**Marks**	**Start**	**End**	**Marks**
1	Sept 9	Sept 29	5%	Sept 9	Oct 6	7%
2	Sept 16	Oct 9	6%	Sept 23	Oct 20	8%
3	Oct 7	Nov 10	7%	Oct 14	Nov 17	8%
4 (50% easier)	Nov 4	Dec 8	8%	Nov 11	Dec 8	7%
Quiz			4%			Bonus 5%

Code Organization

- C/C++
  - Files (usually) separated into source and header
  - Header: contains class definition and function *prototypes*
    - Meant to be exported, i.e. used by others
  - Source: contains function definition
    - Sometimes *static* functions and *opaque* user-defined types
- Information Hiding
  - Reduces external complexity
    - Prevents user from making access outside of provided interface
  - Results in better abstraction and a stable interface

Static Function

- Function only available in the file it is defined in
- Similar to private member functions
  - But not even specified in the header file
  - Completely hidden from external user

foo.c

```
static int foo_helper(int a) { /* complex calculation */ }
int foo(int x, int y) {
    return foo_helper(x) + foo_helper(y);
}
```

foo.h

```
int foo(int x, int y); /* no one knows about foo_helper */
```

Opaque Data Type

- Data type only *declared*, not *defined*
  - Concrete representation hidden from its users

list.c

```
struct Node { int value; struct Node * next; };
struct List { struct Node * head; };
struct List * create_list(void) {
    struct List * list = malloc(sizeof(struct List));
    list->head = NULL; return list;
}
```

list.h

```
struct List;
struct List * create_list(void);
struct List * add_to_list(struct List * list, int value);
```

Returns an *opaque pointer*

Code Organization

- Module
  - A Python source file or directory
    - Contains a collection of definitions and statements
  - Prevents name conflict
    - E.g. `math.abs` vs. `bodybuilder.abs`
    - Similar to C++ namespace
- Import
  - To gain access to definitions and functionalities of a module
    - No information hiding, everything is accessible
  - Use name of file (minus the .py)
    - E,g, to import foo.py, use `import foo`
    - File executed when importing

Python Idiom

- Avoid execution if imported as module

foo.py

```
print(“hello world”)
def main():
    pass
# will not run if imported as module
if __name__ == “__main__”:
    main()
```

bar.py (we run this file)

```
import foo              # prints “hello world”
print(type(foo))        # prints <type ‘module’>
print(foo.__name__)     # prints “foo”
print(__name__)         # prints “__main__”
```

Import

- A Python statement
- Can be called anywhere in code
  - Convention: prefer at top of source file
  - Optimization: avoid import until just before use

```
def unlikely_called_function():
    import huge_module
    huge_module.do_something()
```

- Python tracks which module already imported
  - Same module will not be re-imported

Import

- Import functions and types into local namespace
  - Don’t have to prefix with module name

```
from collections import namedtuple, OrderedDict, deque
ordered = OrderedDict(zip(“abcde”, range(5)))

# OrderedDict([('a', 0), ('b', 1), ('c', 2), ('d', 3), ('e', 4)])
print(ordered)

# <class 'collections.OrderedDict'>
print(OrderedDict)
```

Import

- A directory can also be imported (a.k.a *package*)
  - Must have a special file named __init__.py
  - Good way to organize very large code base

mydir/__init__.py

```
from mydir.foo import bar
print(“hello import”)
```

mydir/foo.py

```
def bar():
  print(“hello bar”)
```

```
>> import mydir
hello import
>> mydir.bar()    # imported definition is exported
hello bar
```

Default Argument

- Default value assigned to missing arguments
- In C++, default arguments always recreated

```
struct A {
  int x;
  A() : x(0) {}
  ~A() {
    cout << "destroyed\n";
  }
};

void foo(A a=A()) {
  cout << a.x << endl;
  a.x = 5;
}
```

```
int main() {
    foo();
    foo();
    return 0;
}
```

```
$ ./foo
0
destroyed
0
destroyed
```

Default Argument

- In Python, only evaluated *once*, when defined
  - Beware of mutable default arguments!

```
def add_topping(budget, toppings=list()):
    if budget > 4.99:
        budget -= 4.99
        toppings.append(“chipotle steak”)
    …
    return toppings

>> pizza1 = add_topping(5)
>> pizza2 = add_topping(4)
>> add_topping(3)
[‘chipotle steak’, ‘grilled chicken’, ‘broccoli’]
```

Default Argument

- Workaround
  - Convention: use None
    - Similar to NULL in C/C++

```
def add_topping(budget, toppings=None):
    if toppings is None:
        toppings = list()
    …
    return toppings

>> pizza1 = add_topping(5)
>> pizza2 = add_topping(4)
>> add_topping(3)
[‘broccoli’]
```

Function Scope

- Local variable valid until end of function
  - Can be defined within nested block
    - Usable outside of block where definition occurred
- Global variable
  - Reassignment requires use of *global* statement

```
MUSIC = [ “Pop”, “EDM” ]
def retro():
    global MUSIC
    MUSIC = [ “Classic”, “Jazz” ]
>> retro()
>> print(MUSIC)
[‘Classic’, ‘Jazz’]
```

Function Scope

- Global variable
  - Read and *update* are permitted without *global*

```
MUSIC = [ “Pop”, “EDM” ]
def retro():
    # empties the list and re-populate it
    MUSIC.clear()
    MUSIC.extend([ “Classic”, “Jazz” ])

>> retro()
>> print(MUSIC)
[‘Classic’, ‘Jazz’]
```

Scope and Except

- An “exception” to function scope
    - Exception variable is deleted at end of block

```
def try_fail():
    e = "hello"
    try:
        a = range(2)
        print(a[3])
    except IndexError as e:
        print(e)

    # NameError: name 'e' is not defined
    print(e + " world")
```

Scope and Except

- Why the “exception”?
  - Exception variable interferes with garbage collection
  - Potentially large amount of memory cannot be reclaimed until exception variable is deleted
  - Technical detail in future lecture
- Workaround
  - If you want to keep it, reassign it to another variable

```
keep_me = None
try:
    …
except IndexError as e:
    keep_me = e
# range object index out of range
print(keep_me)
```

List Comprehension

- Creates *sequence* from an *iterable*
  - Form 1:
    - *P*(x) **for** x **in** *iterable*
- map function:
  - Applies function, e.g. *P*(x), to all elements
  - Returns a list of results *in the same order*

```
# function: P(x)
def map(function, iterable):
    output = []
    for element in iterable:
        output.append(function(element))
    return output
```

List Comprehension

- Form 2:
  - x **for** x **in** *iterable* **if** *F*(x)
- `filter` function:
  - From input iterable, returns only elements that satisfies *F*(x)
    - i.e. when *F*(x) returns true
    - Original ordering is preserved

```
def filter(function, iterable):
    output = []
    for element in iterable:
        if function(element): # if F(x) is true
            output.append(element)
    return output
```

List Comprehension

```
>> [ x*x for x in range(1, 6) ]
[1, 4, 9, 16, 25]

>> [ x for x in range(10) if x%2 ]
[1, 3, 5, 7, 9]

def square(x):
    return x * x
>> map(square, range(1, 6))
[1, 4, 9, 16, 25]

def odd(x):
    return x%2 != 0
>> filter(odd, range(10))
[1, 3, 5, 7, 9]
```

Imperative Programming Style

Functional Programming Style