Finishing / Starting

• Intro to user-defined classes
• Programming environments
• Summing up

Final Exam Review – all labs (Q/A)

“When you express your understanding in code, you debug your brain.”

Python User-Defined Classes

A Short Intro

Recall: Python data types (objects):

• type – range of values and operations
• value(s)
• id

A type, or class, is a template for objects.

Objects are instances of a class.

Every Python object is an instance of a class:

```
>>> y = 'abc'
>>> type(y)    # y is an instance of class str
<class 'str'>
>>> y         # returns class info
'abc'
>>> id(y)     # returns value info
4320804568
>>> y.count('a')  # returns memory location info
4298926768
>>> y - 4    # class/type restricts operations
TypeError
```

```
>>> y = str(97403)     # y is an instance of class str
>>> type(y)            # returns class info
<class 'str'>
>>> y                 # returns class/info
'97403'
>>> id(y)              # returns value info
4298926768
>>> y.count('97403')   # returns memory location info
4298926768
>>> y - 4              # class/type restricts operations
TypeError
```

```
class Turtle

>>> t1 = Turtle()
>>> t2 = Turtle()

• Turtle is a constructor method for class Turtle
• Turtle() instantiates a Turtle object
• t1 and t2 are instances of the class Turtle (turtle objects)
```

```
>>> t1                    # y is an instance (object) of class (type) str with value and id
>>> str is a constructor method for class str
>>> str() instantiates a str object
>>> y is an instance (object) of class (type) str with value and id
```

str is a constructor method for class str
str() instantiates a str object
y is an instance (object) of class (type) str with value and id
Recollect Python data types (objects):

- **type**
  - range of values ➝
  - **attributes**
    - object methods (operations)
    - object descriptors
  - **value(s)**
  - **id**

> Primitive objects – single value
> Collections objects – strings, lists, tuples, dicts
> Other Python objects – functions, turtle
> **User-defined objects**

> Recall:
> ```python
>>> t1
<turtle.Turtle object at 0x102692a90>

>>> type(t1)
<class 'turtle.Turtle'>

>>> id(t1)
4335413904
```

> ```python
>>> t1.heading()

>>> t1.pos()

>>> t2.heading()

>>> t2.pos()

>>> t1.color()

>>> t2.shape()

>>> t1.fd(100)

>>> t2.fd(200)
```

> ```python
>>> dir(Turtle)

>>> dir(t1)

>>> t1.heading()

>>> t1.seth(75)

>>> t1.pos()

>>> t2.heading()

>>> t2.pos()

>>> t1.color()

>>> t2.shape()

>>> t1.shape('turtle')

>>> t2.pos()

>>> t2.color('blue')

>>> t1.color()

>>> t2.shape()
```

> Classes and Objects
> Creating our own classes –
> ```python
> Another way to extend the language
> as with functions:
> names
> encapsulates
> ```
> now data types: **Object-oriented programming**
Classes and Objects

Creating our own data types, or classes –
To create our own objects, or instances –

Example: Point

```python
class Point(object):
    '''Represents a point in 2-d space.'''
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

>>> p1 = Point()
>>> p1
<__main__.Point object at 0x15695d0>
```

special constructor method `__init__` initializes class names and values:

```python
class Point():
    '''Represents a point in 2-d space.'''
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

>>> p1 = Point()
>>> p2 = Point(3, 4)
```

Add methods to get and set (mutate) attribute values:

```python
class Point(object):
    '''Represents a point in 2-d space.'''
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def getx(self):
        return self.x
    def setx(self, newx):
        self.x = newx
        return None
    def gety(self):
        return self.y
    def sety(self, newy):
        self.y = newy
        return None

>>> p2 = Point(3, 4)
>>> p2.getx()
3
>>> p2.gety()
4

>>> p1 = Point()
>>> p1.sety(100)
>>> p1.gety()
100
```
Classes and Objects

```python
>>> p1 = Point()
<__main__.Point object at 0x00000280>
```

```python
>>> p2 = Point(3, 4)
```

```python
>>> id(p2)
4389818152
```

```python
>>> p1.getx()

>>> Point.getx(p2)
```

```python
>>> p1.sety(100)
```

```python
>>> p1.gety()
```

### Besides `__init__`, other special (‘magic’) methods:

- `__str__` and `__add__`, for example

```python
>>> ctr = 1
>>> ctr + 1
1
>>> s = 'abc'
>>> s + 'def'
```

```python
>>> s.__add__('def')
```

```python
>>> s + 'def'
```

`__add__` is a special method used by the `+` operator. Its behavior varies per object type (overloaded operators).

### Other special methods:

- `__str__` and `__add__`, for example

```python
class Point(object):
    """Represents a point in 2-d space."""
    def __add__(self, i):
        self.x += i
        self.y += i
        return None
```

```python
>>> p1 = Point()
```

```python
>>> p1 + 99
```

```python
>>> p1.getx()
99
```

```python
>>> p1.gety()
99
```

`__str__` is a special (“magic”) method used to print the object using the `print()` function.

`__str__` should return a string.

```python
class Point(object):
    """Represents a point in 2-d space."
    def __str__(self):
        return f"***{self.x}, {self.y}***"
```

This is what the `print()` function uses.
Classes and Objects

__str__ is a special method used to print the object using print

class Point(object):
    '''Represents a point in 2-d space. '''
    def __str__(self):
        return f'***{self.x}, {self.y}***'

>>> p4 = Point(0, 100)  # execute __init__ method
>>> print(p4)  # execute __str__ method
*** 0, 100 ***

Algorithms ➔ Programs

- Editor for writing programs in a particular language and according to style guidelines

- Translator (interpreter/compiler, linker) for executing program (and reporting results)

- Other tools, e.g., for testing and debugging

An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.

Programming Environments

- Integrated Development Environment (IDE) e.g., IDLE, PyCharm, Jupyter notebooks, Eclipse

Individual tools also exist outside of IDEs:
- Text editor + Command line + Python debugging module (for example)

---

Finishing / Starting

✓ Intro to user-defined classes

- Programming environments
- Summing up

Final Exam Review – all labs (Q/A)

“Everybody should learn how to program a computer, because it teaches you to think.”
Once upon a time...

switches
cards
teletype machines (command-line)
video display terminals (command-line)
Graphical User Interfaces (GUIs – point and click)
command line interface programs (Terminals/cmd.exe)

integrated development environment (IDE) v. command-line

execute this program

 recalling: p22_sqrt_key_F19.py ...

    def main():
        """Square root comparison program driver."""
        sqrt_compare(25, 5)
        sqrt_compare(25, 10)
        ...
        return None

    main()
Integrated Development Environment (IDE) v. Command-line programming

IDLE: <Run Module> to execute .py file open in Editor Window

COMMAND LINE:

sysadmins-air:F19-projects kfreeman$ python3.6 p22_sqrt_key_F19.py
For 25 using 5 iterations:
mysqrt value is: 5.000023178253949
math lib sqrt value is: 5.0
This is a 0.0 percent error.
For 25 using 10 iterations:
mysqrt value is: 5.0
math lib sqrt value is: 5.0
This is a 0.0 percent error.

sysadmins-air:F19-projects kfreeman$

Recall: Python import os

>>> os.getcwd()
/Users/kfreeman

sysadmins-air:F19-projects kfreeman$

>>> os.chdir('...
/Users/kfreeman/Documents/cis210F19/F19-projects
Sysadmins-air:F19-projects kfreeman$

(a)
(b)
c
(d)

def main():
    '''Square root comparison'''
    sqrt_compare(25, 5)
    sqrt_compare(25, 10)
    return None

def main(n, r):
    sqrt_compare(n, r)
    return None

(sysadmins-air:F19-projects kfreeman$ python3.6 ~ kfreeman$ pwd
/Users/kfreeman
(sysadmins-air:F19-projects kfreeman$ cd Documents/cis210F19/F19-projects
(sysadmins-air:F19-projects kfreeman$

import sys

def main():
    '''Use arguments supplied at command line'''
    n = int(sys.argv[1])
    r = int(sys.argv[2])
    sqrt_compare(n, r)
    return None

kfreeman$ python3.6 p22_sqrt_key_F19.py 81 10
81 10

import sys

def main():
    '''Use arguments supplied at command line'''
    print(sys.argv)
    n = int(sys.argv[1])
    r = int(sys.argv[2])
    sqrt_compare(n, r)
    return None

kfreeman$ python3.6 p22_sqrt_key_F19.py 81 10
['p22_sqrt_key_F19.py', '81', '10']
For 81 using 10 iterations:
mysqrt value is: 9.0
math lib sqrt value is: 9.0
This is a 0.0 percent error.
def main():
    '''Use arguments from a file: p22_wk10demo'
    $python3.6 p22_sqrt_key_F19.py < p22_wk10demo.txt
    ...'
    num_tests = int(input())  # first line of file is number of tests
    for i in range(num_tests):
        testlist = input().split(',')  # form the list
        n = int(testlist[0])
        r = int(testlist[1])
        sqrt_compare(n, r)
    return None

Integrated Development Environment (IDE) v. Command-line

Writing — executing — testing and debugging code:

IDE: GUI, integrated, handy tools in one place, overhead to learn (and then learn again). Great for development and testing, especially for large projects.

command line: Shell (text only), not integrated, portable, powerful

built-in editors: e.g., emacs, vim, TextEdit, IDLE

Finishing / Starting

✓ Intro to user-defined classes
✓ Programming environments
• Summing up

Final Exam Review — all labs (Q/A)

Everybody should learn how to program a computer, because it teaches you to think.

Computational problem solving

is an approach to problem solving that is inspired and constrained by the possibilities and limitations of computers and computing

Computational Problem Solving

TASK/PROBLEM ➔ Computational Thinking ➔
PRECISE SPECIFICATION OF PROBLEM/SOLUTION/ALGORITHM

SOLUTION/ALGORITHM ➔ Design/Coding ➔
COMPUTER PROGRAM

COMPUTER PROGRAM ➔ Testing/Debugging ➔
HIGH QUALITY COMPUTER PROGRAM

HIGH QUALITY COMPUTER PROGRAM ➔ Execute ➔
AUTOMATIC, FAST, RELIABLE, REUSABLE SOLUTION
CIS 210 / Welcome

- Understand/revise/refactor/create/implement algorithms and code
- Expand/improve software development skills
  - use structured design and testing methods to develop and implement programs
  - good understanding of a high level programming language

understand, develop, implement, and algorithms for computational problem solving; use structured design and testing methods to develop and implement programs

problem: review, clarify, write examples; test-driven design
algorithm: review, revise, or create algorithm to solve the problem — pseudocode, check against examples
program: design (top down) — determine functions
for each high level function:
  problem, algorithm — review, clarify, ...
  write a brief description
  write function header
  write type contract
  write test cases (include examples)
  write the return statement

CIS 210

understand, develop, implement, and algorithms for computational problem solving; use structured design and testing methods to develop and implement programs, cont’d.

programming: coding (bottom up)
  review Python toolkit
  write Python code (use pseudocode to start)

programming: testing and debugging
  test code using examples of use (automated testing)
  revise and retest as needed
  testing
  • cases for basic, edge, for various expected arguments and results (equivalence classes)
  errors
  • syntax, runtime, logical, documentation
  • assert and try/except

CIS 210

demonstrate good understanding of a high level programming language (Python), for example:

assignment statement
expression
Python repetition — for, while
Python conditional — if
numeric data types (int, float) and operations (e.g., -, **, round, abs)
Boolean data type and operations (e.g., <, and)
string data type and operations (e.g., +, len, count, find, format, formatted strings
Python collections data types and operations — tuples, lists, dictionaries; comprehensions
Python files and file processing
data type "conversion" functions (e.g., str, int, list, float)
NoneType (None)
print/io
Python Standard Library — math, turtle, random modules; import (if __name__ == '__main__')

user-defined functions; function design; docstrings
IDE interactive development environment; intro to programming environments
Python introspection: help, dir, type, id, isinstance, os.getcwd, os.chdir
run-time checking of data and code: assert, try/except

Python is a language (virtual machine) AND
Python is a program (translator/interpreter)

CIS 210 / Welcome
demonstrate robust mental models of data representation and code execution

Python is a language (virtual machine) AND
Python is a program (translator/interpreter)
CIS 210

demonstrate robust mental models of data representation and code execution

- expression evaluation (strong typing, operator overloading)
- flow of control
  - sequential
  - conditionals
  - loops – for, while
  - calling functions
- functions / functional abstraction
  - what happens when a function is called
  - activation record/call stack (Python Visualizer)/namespace
  - parameter passing (call-by-assignment)
  - side effects
  - returned values
- variable assignment
  - memory allocation
  - namespaces and variable scope
  - reference semantics (aliasing and immutable/mutable data types)
  - dynamic typing

Summing up

Programming environments
Intro to user

- variable assignment
- memory allocation
- namespaces and variable scope
- reference semantics (aliasing and immutable/mutable data types)
- dynamic typing

CIS 210

Finishing / Starting

✓ Intro to user-defined classes
✓ Programming environments
✓ Summing up

Final Exam Review – all labs (Q/A)

Everybody should learn how to program a computer, because it teaches you to think.

CIS 210

Programming/Computer Science Concepts

Computational Problem Solving: designing, implementing, checking, revising algorithms/page.

Good programming style: function design: rigor/contract, description (including parameters, returned values, and side effects), parameters of function used, well-defined variables, use of whitespace between operations, avoidance of code, judicious use of inline comments (only not what:

Python is a programming language with Python as an interpreter (program)
Python Shell is a Python build inside print loop
Python primitive elements: Objects, relations, types, memory location (G)
Casting primitive elements: Expressions, expressions evaluate to a value, overloaded operators;
methods, function/operators, short circuit evaluation of boolean expressions

Function: an executable data type, what happens when a function is called
Function design: return a value (return None)
Function side effects
Function as argument
Iterative algorithms, recursive algorithms, Monte Carlo algorithms, data analysis, map and filter patterns

A Structured Approach to Computational Problem-Solving

- review the project specification thoroughly
- write examples of expected results for specific inputs, use review spec, if needed
- develop, rewrite, review a problem-solution approach, use review spec, if needed

- check algorithms using your examples – review spec, if needed
Starting with the lowest level function:
- write the function header
- write the function docstring – type-contract
- write the function docstring – tool description
- write the function docstring – examples of the just concerns developed earlier
- write the return statement
- write the function docstring
- write the function docstring
- write the return statement

- using tools from the Python toolkit, start writing the body of the function
- test algorithms, review spec, use review spec, if needed
- test using examples of the documentation, and your project spec, and then others

- review the project specification thoroughly
- write examples of expected results for specific inputs, use review spec, if needed
- develop, rewrite, review a problem-solution approach, use review spec, if needed
- check algorithms using your examples – review spec, if needed
Starting with the lowest level function:
- write the function header
- write the function docstring – type-contract
- write the function docstring – tool description
- write the function docstring – examples of the just concerns developed earlier
- write the return statement
- write the function docstring
- write the function docstring
- write the return statement
- using tools from the Python toolkit, start writing the body of the function
- test algorithms, review spec, use review spec, if needed
- test using examples of the documentation, and your project spec, and then others
- review the project specification thoroughly
- write examples of expected results for specific inputs, use review spec, if needed
- develop, rewrite, review a problem-solution approach, use review spec, if needed
- check algorithms using your examples – review spec, if needed
Starting with the lowest level function:
- write the function header
- write the function docstring – type-contract
- write the function docstring – tool description
- write the function docstring – examples of the just concerns developed earlier
- write the return statement
- write the function docstring
- write the function docstring
- write the return statement
- using tools from the Python toolkit, start writing the body of the function
- test algorithms, review spec, use review spec, if needed
- test using examples of the documentation, and your project spec, and then others
- review the project specification thoroughly
- write examples of expected results for specific inputs, use review spec, if needed
CIS 210 Learning Outcomes

• understand, develop, implement algorithms for computational problem solving;
• use structured design and testing methods to develop and implement programs;
• read, write, revise, document, test, and debug code;
• demonstrate robust mental models of data representation and code execution;
• demonstrate good understanding of a high level programming language;
• introduce and/or implement a sampling of classic computer science problem domains and algorithms.