Finishing / Starting

K-means cluster algorithm; List comprehensions; Return; Depth-first search; Functions as parameters

Python User-Defined Classes

Final Exam Review – all labs (Q/A)

\[ S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \]

T is a list of x such that x is a member of S and x is even

\{x \mid x \text{ in } S \text{ and } x \text{ is even}\}
CIS 210 Introduction to Computer Science  Winter 2017
List Comprehensions

\[ S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \]

\( T \) is a list of \( x \) such that \( x \) is a member of \( S \) and \( x \) is even

\{\( x \mid x \in S \text{ and } x \text{ is even} \}\)

\[
\text{for } x \text{ in } S: \quad \# \text{ procedural}
\]
\[
\text{if even}(x):
\]
\[
\quad \text{T.append}(x)
\]

\[ T = [x \text{ for } x \text{ in } S \text{ if even}(x)] \quad \# \text{ declarative} \]

def checkfunc1():
    """ ""
    mylist = [1, 2, 3, 4, 5]

    for item in mylist:
        if item > 100:
            return True
        else:
            return False

    >>> checkfunc1()
def checkfunc1():
    '''
    mylist = [1, 2, 3, 4, 5]
    for item in mylist:
        if item > 100:
            return True
        else:
            return False

def checkfunc2():
    '''
    mylist = [1, 2, 3, 4, 5]
    for item in mylist:
        if item > 100:
            return True
    return False

def checkfunc3():
    '''
    mylist = [1, 2, 3, 4, 5]; yourlist = []
    for item in mylist:
        yourlist.append(item * 2)
    return yourlist

>>> checkfunc3()
def checkfunc3():
    '''
    mylist = [1, 2, 3, 4, 5]; yourlist = []
    for item in mylist:
        yourlist.append(item * 2)
    return yourlist

def checkfunc4():
    '''
    mylist = [1, 2, 3, 4, 5]; yourlist = []
    for item in mylist:
        yourlist.append(item * 2)
    return yourlist

def checkfunc4():
    '''
    mylist = [1, 2, 3, 4, 5]
    yourlist = [item * 2 for item in mylist]
    return yourlist
CIS 210 Introduction to Computer Science  Winter 2017

t = {0:[1,2],
    1:[5,6],
    2:[3,4,7],
    3:[],
    4:[],
    5:[7],
    6:[7],
    7:[]}

def dfs(tree, start):
    """(dict, int) -> None"
    > dfs(t, 0)
    ""
    print(start)
    for child in tree[start]:
        dfs(tree, child)
    return None

def dfsFind(tree, start, end):
    """(dict, int, int) -> Boolean ""
    #print(start)
    if start == end:  #check for end node
        return True
    for child in tree[start]:
        found = dfsFind(tree, child, end)
        if found:  #stop (only) if node found
            return True
    #non-goal leaf
    return False
def dfsPath(tree, start, end, path):
    '''(dict, int, int, list) -> list

    > dfsPath(t, 0, 7, [])
    [0, 1, 5, 7]
    
    path = path + [start]  # keep track of path
    print(path)
    if start == end:
        print()
        return path

    for child in tree[start]:
        newpath = dfsPath(tree, child, end, path)
        if newpath != []:  # if there, done
            return newpath

    return []

CIS 210 Introduction to Computer Science  Winter 2017

def test_is_member(f):
    '''(function) -> Boolean

    Test function f (one of the isMember functions).
    Return True if it passes all tests, otherwise return False

    >>> test_is_member(isMemberR)
    '''
    if f([1, 3, 5, 7], 4) != False:
        return False

    if f([23, 24, 25, 26, 27], 5) != False:
        return False

    if f([0, 1, 4, 5, 6, 8], 4) != True:
        return False

    ...
    return True
test_cases =
    ([[1, 3, 5, 7], 4, False],
     ([23, 24, 25, 26, 27], 5, False),
     ([0, 1, 4, 5, 6, 8], 4, True),
     ([0, 1, 2, 3, 4, 5, 6], 3, True),
     ([1, 3], 1, True),
     ([2, 10], 10, True),
     ([99, 100], 101, False),
     ([42], 42, True),
     ([43], 44, False),
     ([], 99, False))
for test in test_cases:
    arg1 = test[0]
    arg2 = test[1]
    result = test[2]
    if f(arg1, arg2) != result:
        return False
return True

Python User-Defined Classes
A Short Intro

Recall: Python data types (objects):

type – range of values and operations

value

id
3/17/17

CIS 210 Introduction Computer Science Winter 2017

>>> y = 'abc'
>>> y = str()

>>> type(y)
<class 'str'>

>>> type(y)
<class 'str'>

>>> y
'abc'

>>> y
''

>>> id(y)
4320804568

>>> id(y)
4298926768

>>> y.count('a')
4

>>> y - 4

CIS 210 Introduction Computer Science Winter 2017

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 = str()

>>> type(y)
<class 'str'>

>>> y
''

>>> id(y)
4298926768
>>> y = str()
>>> type(y)
<class 'str'>
>>> y
''
>>> id(y)
4298926768

• str is a **constructor method** for class str
• str() **instantiates** a str object
• y is an **instance (object)** of class str

class Turtle

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

• Turtle is a **constructor method** for class Turtle
• t1 and t2 are **instances of the class** Turtle (turtle objects)
• Turtle() **instantiates** a Turtle object
>>> t1
<turtle.Turtle object at 0x102692a90>

>>> id(t1)
4335413904

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

>>> t1.fd(100)  # for example

>>> t1.fd(100)
>>> t2.fd(200)

>>> t1.heading()
>>> t1.pos()
>>> t2.heading()
>>> t2.pos()
>>> t1.color()
>>> t2.shape()
Recall: Python data types (objects):

- **type**
  - range of values
  - **attributes**
    - object methods (operations)
    - **object descriptors**

- **value(s)**

- **id**

```python
>>> t1.fd(100)
>>> t2.fd(200)

>>> t1.head(heading)
>>> t1.pos()

>>> t2.head(heading)
>>> t2.pos()

>>> t1.color()
>>> t2.shape()
```

```python
>>> t1.seth(75)
>>> t1.setpos(0, 0)

>>> t1.shape('turtle')
>>> t2.color('blue')
```
primitive objects – single value

collections objects – strings, lists, tuples, dicts

other Python objects – functions, turtle

user-defined objects

Classes and Objects

Creating our own classes –

Another way to extend the language

as with functions:

  names
  encapsulates

now data types: **Object-oriented programming**
Classes and Objects

Creating our own classes –

To create our own objects –

Example: Point

```python
>>> class Point(object):
    '''Represents a point in 2-d space.'''

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

Classes and Objects

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

for example, x and y

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)
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

>>> p2 = Point()
>>> p3 = Point(3, 4)
>>> p3
<__main__.Point object at 0x101482a20>
```

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 gety(self):
        return self.y
```
Classes and Objects

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()

>>> p3 = Point(3, 4)

>>> p2.getx()  # >>> Point.getx(p2)
0

>>> p3.sety(100)  # >>> p3.gety()
100
Classes and Objects

Other special methods: __str__ and __add__, for example

ctr = 1
ctr = ctr.__add__(1)

__str__ is a special method where the object returns a string representation of the object

this is what the print function uses

class Point(object):
    """Represents a point in 2-d space."""

    def __str__(self):
        return '***{}, {}***'.format(self.x, self.y)
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 '***{}, {}***'.format(self.x, self.y)

>>> print(p)
***0, 100***

Algorithms → Programs

• writing

• executing (and getting results)

• testing and debugging
Algorithms → Programs
Programming Environments

• writing

• executing (and getting results) – interpret/compile and run program (e.g., Python)

• testing and debugging

Algorithms → Programs
Programming Environments

• Integrated Development Environment (e.g., IDLE, PyCharm, Jupyter notebooks)

• Editor + Command line + Debugging module
Integrated Development Environment (IDE) v. Command-line programming

command line - shell* for operating system

text-based (pre-GUI)

*program that accepts commands as text input and converts [interprets] them to appropriate [operating system] functions.

cicada0:Documents kfreeman$ python3.3
Python 3.3.0 (v3.3.0:bd8af90ebe2, Sep 29 2012, 01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello, world')
hello, world
>>> quit()
cicada0:Documents kfreeman$
Integrated Development Environment (IDE) v. Command-line programming

cicada0:Documents kfreeman
$ python3.3 cis210_p4_1.py
TestResults(failed=0, attempted=9)
$

Command-line programming

cicada0:Documents kfreeman
$ python3.3 cis210_p4_1.py

add to cis210_p4_1.py file:

print(alphapinDecode(‘lohi’))
Command-line programming

cicada0:Documents kfreeman
$ python3.3 cis210_p4_1.py

add to cis210_p4_1.py file:

```python
def main():
    print(alphapinDecode('lohi'))
    return None

main()
```

passing parameters

cicada0:Documents kfreeman
$ python3.3 cis210_p4_1.py lohi

add to cis210_p4_1.py file:

```python
import sys
def main():
    arg1 = sys.argv[1]
    print(alphapinDecode(arg1))
    return None

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

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.

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

CIS 210 Winter 2017 Overarching themes:

understand, develop, implement algorithms (computational problem solving)

• test and documentation-driven program design
• read, write, document, test, debug, revise code
• mental model of code execution
• good understanding of Python language
• familiarity with sampling of computer science concepts and algorithms
Overarching themes:

- test and documentation-driven program design
- read, write, document, test, debug, revise code
  - type contracts
  - function descriptions
  - returned values and side effects
  - example function calls
  - test cases

- errors
  - syntax, runtime, logical, documentation
  - try/except

- testing
  - equivalence classes, automated testing
CIS 210 Introduction Computer Science Winter 2017

- mental model of code execution
  - sequential execution
  - variable assignment
  - expression evaluation
  - functions
    - what happens when a function is called
    - activation records/call stack (Python Visualizer)
    - parameter passing (call-by-assignment)
    - side effects
    - returned values
  - flow of control
    - conditionals
    - loops – for, while

CIS 210 Introduction Computer Science Winter 2017

- good understanding of Python language and programming language concepts

  – objects (value, type, id)
  – assignment (label, memory location, aliasing)
  – dynamic typing (v. static typing)
  – parameter passing (call-by-assignment)
  – data types (mutable, immutable)(built in, user-defined)
  – operators, functions, methods (are all methods)
  – functions (first class citizens) (built in, user-defined)
### CIS 210 Introduction Computer Science Winter 2017

- good understanding of Python language
  - file processing; accessing data from the internet
  - list comprehensions, string formatting
  - procedural and declarative semantics
  - modules: math, random, turtle, doctest, urllib

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CIS 210 Introduction Computer Science Winter 2017</td>
</tr>
</tbody>
</table>

- familiarity with sampling of classic computer science problem domains and algorithms
  - Monte Carlo algorithm
  - encryption and decryption
  - recursion; iterative and recursive algorithms
  - binary search; set membership
  - depth first search
  - loop patterns (accumulator, map, filter)
  - data analysis
  - data mining (k-cluster) and visualization
CIS 210 Introduction to Computer Science  Winter 2017

Python toolkit
numeric data types (int, float) and operations (e.g., +, **, round, abs)
string data type and operations (e.g., +, len, count, find, format)
Boolean data type and operations (e.g., <, and)
collection data types and operations (tuples, lists, dictionaries); list
comprehensions
expressions
assignment
Python repetition – for, while
Python conditionals – if
user-defined functions; function design; docstrings
files and file processing; accessing data from the internet

Python Standard Library – math, turtle, random, doctest, urllib modules;
import
IDLE interactive development environment; help function

CIS 210 Introduction Computer Science Winter 2017

“I think everyone should know how to program a computer, because it teaches you to think. I view computer science as a liberal art, something everyone should learn to do.”
(Steve Jobs)